2 Komponenten für die Benutzeroberfläche

Dieses Kapitel gibt einen Überblick über die wichtigsten **Komponenten**, die in der Tool-Palette **für die Benutzeroberfläche** von Programmen zur Verfügung gestellt werden. Die meisten von ihnen entsprechen **Steuerelementen** (Bedienelementen), die man in vielen Windows-Programmen findet: Sie zeigen dem Anwender Informationen an oder nehmen Anweisungen und Informationen von ihm entgegen.

Angesichts der großen Anzahl von Eigenschaften, Methoden und Ereignissen der Komponenten ist keine Vollständigkeit beabsichtigt: Die Dateien der Online-Hilfe haben einen Umfang von vielen Megabytes und sollen hier nicht wiederholt werden. Bei der Auswahl der vorgestellten Elemente wurde darauf geachtet, dass sie für viele Anwendungen nützlich und auch für andere Elemente typisch sind.

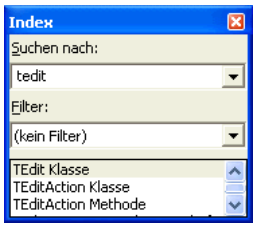
Zusammen mit den Komponenten und ihren Elementen werden auch Konzepte wie Datentypen, Anweisungen, Funktionen und Klassen vorgestellt. So gibt dieses Kapitel gleichzeitig auch einen kleinen Überblick über die Themen, die ab Kapitel 3 ausführlich behandelt werden.

Weitere Komponenten der Tool-Palette werden in Kapitel 10 behandelt. Viele dieser Komponenten sind nicht schwieriger als die hier vorgestellten und können auch ohne ein Studium der Kapitel 3 bis 9 verwendet werden.

2.1 Die Online-Hilfe zu den Komponenten

Da in diesem Kapitel nicht alle Elemente der Komponenten ausführlich behandelt werden, soll zuerst gezeigt werden, wie man mit der Online-Hilfe weitere Informationen zu den einzelnen Komponenten bekommen kann.

Die Komponenten der Tool-Palette sind in der Online-Hilfe unter ihrem Datentyp eingetragen. Diese Namen beginnen meist mit einem vorangestellten „T“ (also z.B. „TEdit“ für die Edit-Komponente). Über *Hilfe\Borland Hilfe* erhält man dann unter *Hilfe\Index* eine Seite, die etwa folgendermaßen aufgebaut ist.



TEdit Klasse

Eigenschaften | Methoden | Ereignisse | Alle Elemente

TEdit kapselt die einzeiligen Eingabefelder von Windows.

Unit

StdCtrls

Syntax

```
class TEdit : public Stdctrls::TCustomEdit
```

Beschreibung

Mit einem TEdit-Objekt können Sie eines der unter Windows üblichen Standard-Eingabefelder in ein Formular einfügen. Eingabefelder dienen dazu, von Benutzern eingegebenen Text zu erfassen. Außerdem kann mit ihnen Text für den Benutzer angezeigt werden.

Wenn Sie Text lediglich anzeigen wollen, hat der Benutzer bei einem Eingabefeld die Möglichkeit, den Text zu markieren und in die Zwischenablage zu kopieren. Wenn diese Bearbeitungsmöglichkeiten nicht benötigt werden, verwenden Sie ein Label-Objekt.

TEdit implementiert das allgemeine Verhalten von TCustomEdit . TEdit deklariert viele der von TCustomEdit geerbten Eigenschaften als published, führt jedoch kein neues Verhalten ein. Für spezielle Eingabefelder können Sie andere von TCustomEdit abgeleitete Klassen verwenden oder selbst welche ableiten.

Hierarchie

System::TObject
Classes::TPersistent
Classes::TComponent
Controls::TControl
Controls::TWinControl
Stdctrls::TCustomEdit

Alle Seiten zu den Komponenten der Tool Palette enthalten Verweise auf Seiten mit Eigenschaften, Methoden und Ereignissen

– Über **Eigenschaften** werden alle Eigenschaften der Komponente angezeigt.

TEdit Eigenschaften

Klasse | Methoden | Ereignisse | Alle Elemente

Von Stdctrls::TCustomEdit geerbte Eigenschaften

Eigenschaft	Beschreibung
AutoSelect (public)	Bestimmt, ob der gesamte Text im Eingabefeld automatisch markiert wird, wenn das Steuerelement den Fokus erhält.
AutoSize (public)	Legt fest, ob die Höhe des Eingabefeldes automatisch an den Text angepasst wird.
BorderStyle (public)	Legt fest, ob der Client-Bereich im Eingabefeld eine einzelne Linie als Rahmen verwendet.
CanUndo (public)	Gibt an, ob das Memo-Steuerelement Änderungen enthält, die gespeichert werden können.
CharCase (public)	Bestimmt die Schreibweise des Textes im Eingabefeld.
HideSelection (public)	Legt fest, ob die visuelle Kennzeichnung des markierten Textes erhalten bleibt, wenn der Fokus an ein anderes Steuerelement übergeht.
MaxLength (public)	Legt die maximale Anzahl der Zeichen fest, die der Benutzer in das Eingabefeld eingeben kann.
Modified (public)	Gibt an, ob der Benutzer den Text im Eingabefeld bearbeitet hat.
OEMConvert (public)	Bestimmt, ob die in das Eingabefeld eingegebenen Zeichen von ANSI in OEM und wieder zurück in ANSI konvertiert werden sollen.
PasswordChar (public)	Gibt das Zeichen an (falls vorhanden), das anstelle der tatsächlich in das Steuerelement eingegebenen Zeichen angezeigt werden soll.
ReadOnly (public)	Legt fest, ob der Benutzer den Text im Eingabefeld bearbeiten kann.
SelLength (public)	Gibt die Anzahl der markierten Zeichen (Bytes) an.
SelStart (public)	Legt die Position des ersten markierten Zeichens im Text fest.
SelText (public)	Gibt den markierten Textbereich des Eingabefeldes an.

Von Controls::TWinControl geerbte Eigenschaften

Eigenschaft	Beschreibung
AlignDisabled (public)	Signalisiert, ob die Neuausrichtung des untergeordneten Steuerelements deaktiviert ist.
BevelEdges (public)	Legt fest, welche Seiten des Steuerelements mit einem Rahmen angezeigt werden.

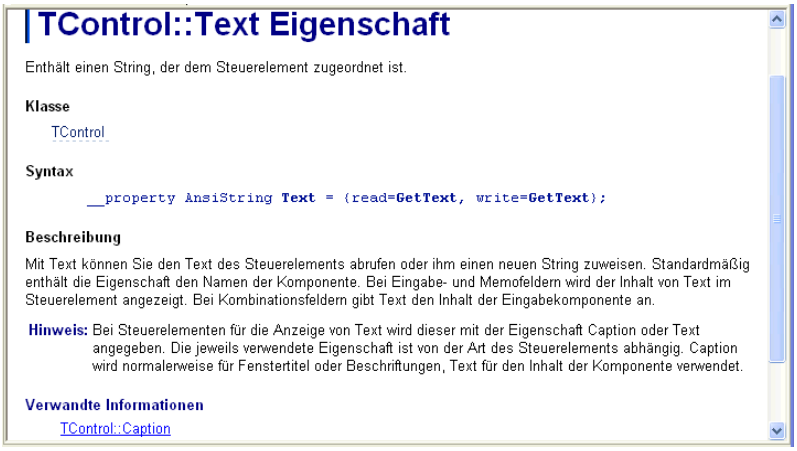
Sie sind in Gruppen mit Überschriften wie „Von … geerbte Eigenschaften“ zusammengefasst. Diese Anordnung spiegelt ein Charakteristikum der objektorientierten Programmierung wider: Klassen können von sogenannten Basisklassen abgeleitet werden. Sie enthalten dann alle Elemente (Eigenschaften, Methoden und Ereignisse) der Basisklassen. Solche geerbten Elemente sind

keineswegs weniger wichtig als die Elemente der ausgewählten Klasse, selbst wenn sie weit unten in der Liste und der Vererbungshierarchie stehen. Wenn Sie nach einem Element suchen, dürfen Sie deshalb nicht erwarten, es gleich am Anfang der Liste zu finden. So findet sich z.B. die Eigenschaft *Color* relativ weit unten im Abschnitt „Von Controls::TControl geerbte Eigenschaften“.

Die meisten Elemente findet man hier direkt. Manche sind aber auch hinter anderen versteckt (wie z.B. *Text* unter *WindowText*). Diese Seite erhält man auch, indem man die Eigenschaft *Text* im **Objektinspektor** anklickt und dann die Taste *F1* drückt.

In vielen Anwendungen sind nur die als *public* gekennzeichneten Elemente von Bedeutung. Die *protected* Elemente (z.B. einer Edit-Komponente) können in einer Funktion wie *Button1Click* nicht angesprochen werden.

Klickt man ein Element an, erhält man weitere Informationen:



TControl::Text Eigenschaft

Enthält einen String, der dem Steuerelement zugeordnet ist.

Klasse

TControl

Syntax

```
__property AnsiString Text = {read=GetText, write=GetText};
```

Beschreibung

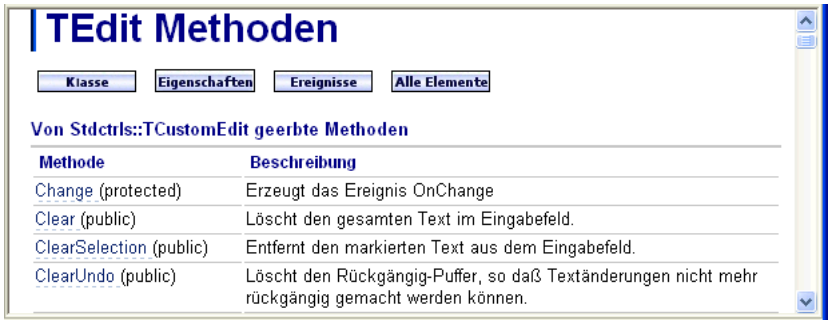
Mit Text können Sie den Text des Steuerelements abrufen oder ihm einen neuen String zuweisen. Standardmäßig enthält die Eigenschaft den Namen der Komponente. Bei Eingabe- und Memofeldern wird der Inhalt von Text im Steuerelement angezeigt. Bei Kombinationsfeldern gibt Text den Inhalt der Eingabekomponente an.

Hinweis: Bei Steuerelementen für die Anzeige von Text wird dieser mit der Eigenschaft Caption oder Text angegeben. Die jeweils verwendete Eigenschaft ist von der Art des Steuerelements abhängig. Caption wird normalerweise für Fenstertitel oder Beschriftungen, Text für den Inhalt der Komponente verwendet.

Verwandte Informationen

TControl::Caption

– Auch die **Methoden** sind nach ihrer Ableitungshierarchie angeordnet.



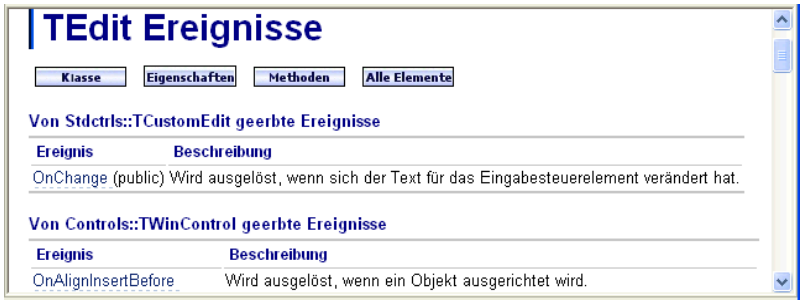
TEdit Methoden

Klasse | Eigenschaften | Ereignisse | Alle Elemente

Von Stdctrls::TCustomEdit geerbte Methoden

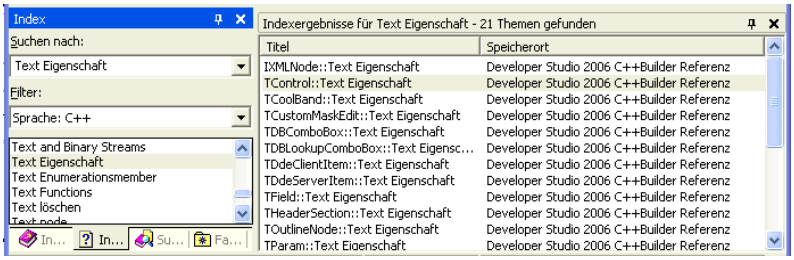
Methode	Beschreibung
Change (protected)	Erzeugt das Ereignis OnChange
Clear (public)	Löscht den gesamten Text im Eingabefeld.
ClearSelection (public)	Entfernt den markierten Text aus dem Eingabefeld.
ClearUndo (public)	Löscht den Rückgängig-Puffer, so daß Textänderungen nicht mehr rückgängig gemacht werden können.

– Nach dem Anklicken von **Ereignisse** werden die Ereignisse angezeigt.



Die soeben beschriebene Vorgehensweise (zuerst die Seite mit der Klasse und dann auf den Seiten mit den Eigenschaften, Methoden oder Ereignissen nach einem Element einer Komponente zu suchen) mag auf den ersten Blick etwas umständlich erscheinen, da man nach dem Element auch direkt im Index suchen kann. Allerdings erhält man so oft sehr viele Treffer, da derselbe Name in verschiedenen Zusammenhängen verwendet wird, und hat dann Schwierigkeiten, den richtigen auszuwählen.

Beispiel: Gibt man „Text Eigenschaft“ im *Index* ein, erhält man viele Treffer, selbst wenn man als Filter „Sprache C++“ wählt. Dabei ist es meist nicht offensichtlich, welcher dieser Treffer der gesuchte ist:



Deshalb ist die Suche über den Namen der Klasse oft doch einfacher.

Bei Eigenschaften und Ereignissen, die im Objektinspektor angezeigt werden, kann man diese auch im **Objektinspektor** anklicken und dann ***F1*** drücken. Da im Objektinspektor aber nicht alle Eigenschaften und keine Methoden angezeigt werden, steht diese Möglichkeit nicht für alle Elemente zur Verfügung.

Im C++Builder 6 konnte man die Online-Hilfe außerdem mit der *F1*-Taste folgendermaßen aufrufen, nachdem man

– im Editor den Namen einer Komponente angeklickt hat, oder
– eine Komponente in der Tool-Palette oder auf dem Formular angeklickt hat.

Diese Möglichkeiten waren sehr hilfreich und werden nur in der Hoffnung erwähnt, dass sie in zukünftigen Versionen des C++Builders wieder verfügbar sind.

Aufgabe:

Zeigen Sie mit jeder der in diesem Abschnitt beschriebenen Vorgehensweisen die Online-Hilfe an zu

– der Eigenschaft *Caption* eines Formulars,
– der Eigenschaft *Cursor* in einem Label und einem Button.

2.2 Namen

Wenn man Komponenten aus der Tool-Palette auf ein Formular setzt, werden ihre Namen vom C++Builder der Reihe nach durchnummeriert: Das erste Edit-Fenster erhält den Namen *Edit1*, das zweite *Edit2* usw. Entsprechend auch für andere Komponententypen: Label erhalten die Namen *Label1*, *Label2* usw.

Über diese Namen können dann nicht nur die Komponenten als Ganzes, sondern auch ihre Eigenschaften und Methoden angesprochen werden:

1. In einer Funktion, die wie *Button1Click* zu einem Formular gehört, spricht man eine Methode oder Eigenschaft *x* einer Komponente *k* des Formulars mit *k->x* an.

   Beispiel: In der Funktion *Button1Click* ist *Edit1->Color* die Eigenschaft *Color* der Komponente *Edit1* dieses Formulars:

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

   Die Höhe *Height* eines Labels *Label2* spricht man mit *Label2->Height* an.

2. In einer Funktion, die zu einem Formular *f* gehört, spricht man eine Methode oder Eigenschaft *x* des Formulars einfach mit *x* an oder mit *this->x* an.

   Beispiele: In der nächsten Funktion ist *Color* die Farbe des Formulars und nicht etwa die des Buttons. Durch den Aufruf der Methode *Close* wird das Formular geschlossen und nicht der Button.

   ```
   void __fastcall TForm1::Button1Click(TObject
                                                *Sender)
   {
   Color=clGreen; // oder this->Color=clGreen
   Close();       // oder this->Close();
   }
   ```

Die Bedeutung von x und *this->x* ist hier gleich. Die Schreibweise *this->x* hat lediglich den Vorteil, dass die Programmierhilfe (Code Insight) nach dem Eintippen von „this->“ die Elemente des Formulars auflistet.

3. In einer Funktion, die nicht zu einem Formular *f* gehört, spricht man die Eigenschaft oder Methode *x* einer Komponente *k* des Formulars *f* mit *f->k->x* an.

   Beispiel: Eine solche Funktion kann zu einem anderen Formular oder zu keinem Formular gehören. Die Breite *Width* des Labels *Label3* von *Form1* spricht man dann so an:

```
void f()
{
Form1->Label3->Width=17;
}
```

Die Anmerkungen nach „//“ (siehe 2.) sind übrigens ein sogenannter **Kommentar** (siehe Abschnitt 3.16). Ein solcher Text zwischen „//“ und dem Zeilenende wird vom Compiler nicht übersetzt und dient vor allem der Erläuterung des Quelltextes.

In diesem Zusammenhang stellt sich die Frage, welche Funktionen zu einem Formular gehören. Die Antwort ergibt sich einfach aus ihrer ersten Zeile: Enthält diese „TForm1::“, gehört sie zum Formular *Form1*. Deshalb gehört in den letzten Beispielen die Funktion *Button1Click* zum Formular *Form1*, nicht jedoch die Funktion *f*.

Die vom C++Builder vergebenen Namen können allerdings leicht zu Unklarheiten führen: Steht in *Edit1* der Vorname und in *Edit2* der Nachname, oder war es gerade umgekehrt? Um solche Unklarheiten zu vermeiden, sollte man den Komponenten aussagekräftige Namen geben wie z.B. *Vorname* oder *Nachname*.

Eine solche **Namensänderung** muss **immer im Objektinspektor** durchgeführt werden, indem man den neuen Namen als Wert der Eigenschaft *Name* einträgt. Zulässig sind alle Namen, die mit einem Buchstaben „A..Z“ oder einem Unterstrichzeichen „_“ beginnen und von Buchstaben, Ziffern oder Unterstrichzeichen gefolgt werden. Groß- und Kleinbuchstaben werden dabei unterschieden, und Umlaute sind nicht zulässig.

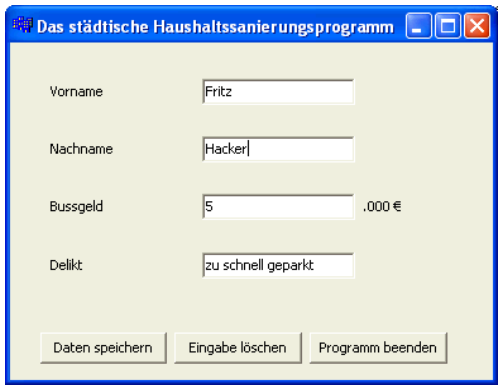
```
Beispiele: Vorname      // zulässig
           123vier      // nicht zulässig, beginnt nicht mit einem Buchstaben
           Preis_in_$   // nicht zulässig wegen $
           Zähler       // nicht zulässig wegen Umlaut
```

Der C++Builder ersetzt den Namen dann in allen Dateien des Projekts (z.B. der Unit des Formulars) an all den Stellen, an denen er den Namen eingefügt hat.

– Falls man diese Dateien nicht kennt, sollte man eine solche Namensänderung nie direkt im Quelltext durchführen; die Folge sind nur mühsam zu behebende Programmfehler.
– An den Stellen, an denen der Name manuell eingefügt wurde, wird er aber nicht geändert. Hier muss er dann auch manuell geändert werden.

Aufgaben 2.2

1. Schreiben Sie ein Programm, das ein Fenster mit folgenden Elementen anzeigt:



   Verwenden Sie dazu die Komponenten *Label*, *Edit* und *Button* aus dem Abschnitt *Standard* der Tool Palette.

2. Ersetzen Sie alle vom C++Builder vergebenen Namen durch aussagekräftige Namen. Da in diesem Beispiel sowohl ein Label als auch ein Edit-Fenster für den Vornamen, Nachnamen usw. verwendet wird, kann es sinnvoll sein, den Typ der Komponente im Namen zu berücksichtigen, z.B. *LVorname* und *LNachname* für die Label und *EVorname* und *ENachname* für die Edit-Fenster.

3. Als Reaktion auf ein Anklicken des Buttons *Eingabe löschen* soll jedes Eingabefeld mit der Methode *Clear* gelöscht werden. Für den Button *Daten speichern* soll keine weitere Reaktion vorgesehen werden. Beim Anklicken des Buttons *Programm beenden* soll das Formular durch den Aufruf der Methode *Close* geschlossen werden.

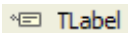
4. Alle Labels sollen in derselben Spaltenposition beginnen, ebenso die TextBoxen. Die Buttons sollen gleich groß sein und denselben Abstand haben.

2.3 Labels, Datentypen und Compiler-Fehlermeldungen

Der **Datentyp** einer Eigenschaft legt fest,

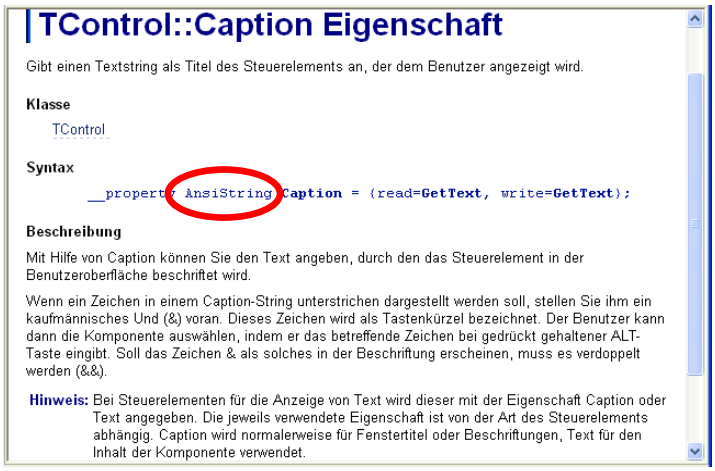
- welche Werte sie annehmen kann und
- welche Operationen mit ihr möglich sind.

In diesem Abschnitt werden einige der wichtigsten Datentypen am Beispiel einiger Eigenschaften eines Labels vorgestellt. Da sich viele dieser Eigenschaften und Datentypen auch bei anderen Komponenten finden, sind diese Ausführungen aber nicht auf Label beschränkt.



Mit einem Label kann man Text auf einem Formular anzeigen. Der angezeigte Text ist der Wert der Eigenschaft *Caption*, die sowohl während der Entwurfszeit im Objektinspektor als auch während der Laufzeit gesetzt werden kann. Anders als bei der Edit-Komponente kann ein Programmbenutzer den auf einem Label angezeigten Text nicht ändern.

Eine Eigenschaft kann einen Text als Wert haben, wenn sie den Datentyp ***AnsiString*** hat. Der Datentyp einer Eigenschaft steht in der Online-Hilfe vor dem Namen der Eigenschaft:



TControl::Caption Eigenschaft

Gibt einen Textstring als Titel des Steuerelements an, der dem Benutzer angezeigt wird.

Klasse

TControl

Syntax

```
__property AnsiString Caption = {read=GetText, write=GetText};
```

Beschreibung

Mit Hilfe von Caption können Sie den Text angeben, durch den das Steuerelement in der Benutzeroberfläche beschriftet wird.

Wenn ein Zeichen in einem Caption-String unterstrichen dargestellt werden soll, stellen Sie ihm ein kaufmännisches Und (&) voran. Dieses Zeichen wird als Tastenkürzel bezeichnet. Der Benutzer kann dann die Komponente auswählen, indem er das betreffende Zeichen bei gedrückt gehaltener ALT-Taste eingibt. Soll das Zeichen & als solches in der Beschriftung erscheinen, muss es verdoppelt werden (&&).

Hinweis: Bei Steuerelementen für die Anzeige von Text wird dieser mit der Eigenschaft Caption oder Text angegeben. Die jeweils verwendete Eigenschaft ist von der Art des Steuerelements abhängig. Caption wird normalerweise für Fenstertitel oder Beschriftungen, Text für den Inhalt der Komponente verwendet.

Da ein String beliebige Zeichen enthalten kann, muss er durch ein besonderes Zeichen begrenzt werden. Dieses Begrenzungszeichen ist in C++ das Anführungszeichen:

```
Label1->Caption="Anführungszeichen begrenzen einen String";
```

Der **Datentyp *int*** ist ein weiterer Datentyp, der häufig vorkommt. Er kann im C++Builder ganzzahlige Werte zwischen -2147483648 und 2147483647

($-2^{31}..2^{31}-1$) darstellen und wird z.B. bei den folgenden Eigenschaften für die **Position und Größe einer Komponente** verwendet:

```
Left   // Abstand zum linken Rand des Formulars in Pixeln
Top    // Abstand zum oberen Rand des Formular
Width  // Breite der Komponente
Height // Höhe der Komponente
```

Alle diese Werte sind in Pixeln angegeben. **Pixel** (Picture Element) sind die Bildpunkte auf dem Bildschirm, aus denen sich das Bild zusammensetzt. Ihre Anzahl ergibt sich aus den Möglichkeiten der Grafikkarte und des Bildschirms und kann unter Windows eingestellt werden. Üblich sind die Auflösungen 1024×768 mit 1024 horizontalen und 768 vertikalen Bildpunkten, 1280×1024 oder 1600×1280.

Will man einer Eigenschaft des Datentyps *int* einen Wert zuweisen, gibt man ihn einfach nach dem Zuweisungsoperator „=“ an. Man muss ihn nicht wie bei einem *AnsiString* durch Anführungszeichen begrenzen:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Label1->Left=10;
Label1->Top=20;
Label1->Width=30;
Label1->Height=40;
}
```

Vergessen Sie nicht, die einzelnen Anweisungen durch Semikolons abzuschließen. Der Compiler beschimpft Sie sonst mit der Fehlermeldung „In Anweisung fehlt ;“.

Der Datentyp *int* ist ein **arithmetischer Datentyp**. Das heißt, dass man mit Ausdrücken dieses Datentyps auch rechnen kann. Die nächste Anweisung verringert die Breite von *Label1* um ein Pixel:

```
Label1->Width=Label1->Width-1;
```

Bei der Ausführung einer solchen Anweisung wird zuerst der Wert auf der rechten Seite des Zuweisungsoperators berechnet. Dieser Wert wird dann der linken Seite zugewiesen. Wenn also *Label1->Width* vor der ersten Ausführung den Wert 17 hatte, hat es danach den Wert 16, nach der zweiten Ausführung den Wert 15 usw. Führt man diese Anweisung beim Anklicken eines Buttons aus,

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Label1->Width=Label1->Width-1;
}
```

wird das Label (und damit der angezeigte Text) mit jedem Anklicken des Buttons um ein Pixel schmaler. Damit dieser Effekt auch sichtbar wird, muss allerdings vorher die Eigenschaft *Color* auf einen Wert wie *clYellow* und *AutoSize* auf den

Wert *false* gesetzt werden. Falls Sie bisher nicht wussten wie breit ein Pixel ist, können Sie sich mit dieser Funktion eine Vorstellung davon verschaffen.

Neben den Datentypen *AnsiString* und *int* kommen **Aufzählungstypen** bei Eigenschaften von Komponenten häufig vor. Eine Eigenschaft mit einem solchen Datentyp kann einen Wert aus einer vordefinierten Liste annehmen, die im Objektinspektor nach dem Aufklappen des Pulldown-Menüs und in der Online-Hilfe nach ***enum*** angezeigt wird. Ein Beispiel ist die Eigenschaft ***Align***, mit der man die Ausrichtung einer Komponente festlegen kann. Dieser Eigenschaft kann man Werte des Aufzählungstyps ***TAlign*** zuweisen:

TAlign Enum

TAlign legt die Ausrichtung eines Steuerelements innerhalb seiner übergeordneten Komponente fest.

Unit

Controls

Syntax

```
[C++] enum TAlign {alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom};
```

Beschreibung

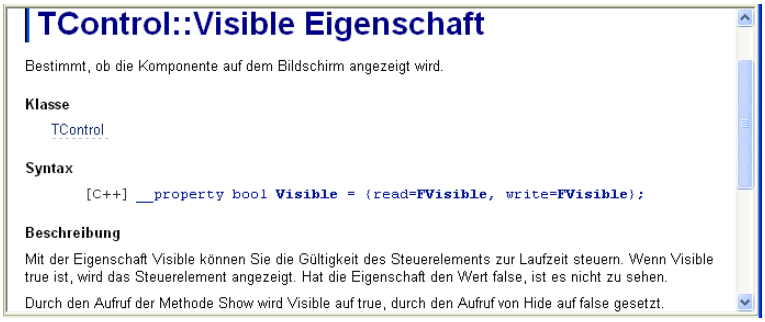
TAlign legt fest, wie ein Steuerelement bezüglich seines übergeordneten Elements platziert wird. Folgende Werte sind möglich:

Wert	Bedeutung
alNone	Das Steuerelement bleibt an der Stelle, an der es platziert wurde. Das ist der Vorgabewert.
alTop	Das Steuerelement wird an den oberen Rand des übergeordneten Elements verschoben und nimmt dessen gesamte Breite ein. Die Höhe des Steuerelements ändert sich nicht.
alBottom	Das Steuerelement wird an den unteren Rand des übergeordneten Elements verschoben und nimmt dessen gesamte Breite ein. Die Höhe des Steuerelements ändert sich nicht.
alLeft	Das Steuerelement wird an den linken Rand des übergeordneten Elements verschoben und nimmt dessen gesamte Höhe ein. Die Breite des Steuerelements ändert sich nicht.
alRight	Das Steuerelement wird an den rechten Rand des übergeordneten Elements verschoben und nimmt dessen gesamte Höhe ein. Die Breite des Steuerelements ändert sich nicht.
alClient	Die Größe des Steuerelements wird so verändert, dass es den Client-Bereich des übergeordneten Elements ausfüllt. Wenn ein Steuerelement bereits einen Teil des Client-Bereichs belegt, wird die Größe des neuen Steuerelements so geändert, dass es den verbleibenden Client-Bereich ausfüllt.
alCustom	Die Position des Steuerelements wird durch Aufrufe der Methoden CustomAlignInsertBefore und CustomAlignPosition der übergeordneten Komponente festgelegt

Diese Werte können z.B. wie in der nächsten Funktion verwendet werden. Insbesondere müssen Werte eines Aufzählungstyps im Unterschied zu einem String nicht durch Anführungszeichen begrenzt werden.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Label1->Align=alClient;
}
```

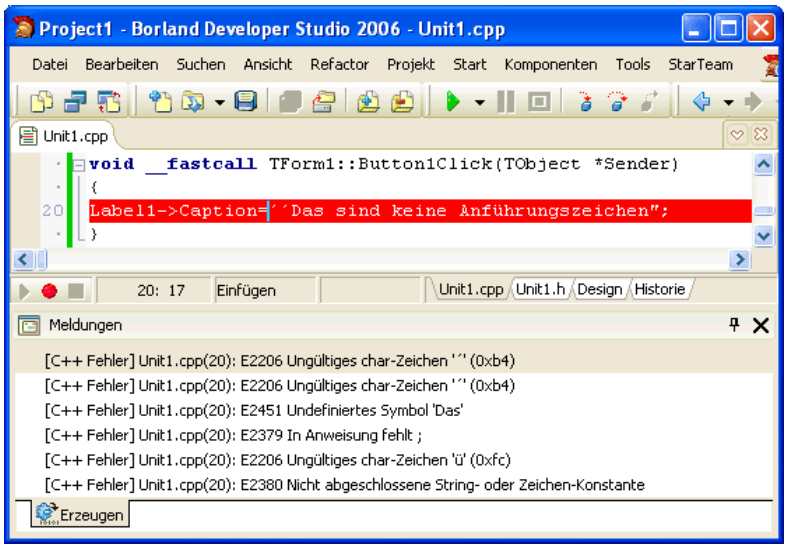
Der Datentyp ***bool*** hat Ähnlichkeiten mit einem Aufzählungstyp. Er kann die beiden Werte *true* und *false* annehmen. Beispielsweise kann man mit der booleschen Eigenschaft *Visible* die Sichtbarkeit einer visuellen Komponente mit *false* aus- und mit *true* anschalten:



Beispiel: Beim Aufruf dieser Funktion wird das Label *Label1* unsichtbar:

```
void __fastcall TForm1::Button1Click(TObject
                                     *Sender)
{
Label1->Visible=false;
}
```

Bei allen Anweisungen muss man die Sprachregeln von C++ genau einhalten. So muss man z.B. als Begrenzungszeichen für einen String das Zeichen " (*Umschalt+2*) verwenden und nicht eines der ähnlich aussehenden Akzentzeichen ` oder ´ bzw. das Hochkomma ' (*Umschalt+#*). Jedes dieser Zeichen führt bei der Übersetzung des Programms zu einer **Fehlermeldung des Compilers** „Ungültiges char-Zeichen "`"“:



Eine solche Fehlermeldung bedeutet, dass der Compiler die rot unterlegte Anweisung nicht verstehen kann, weil sie die Sprachregeln von C++ nicht einhält. Wie dieses Beispiel zeigt, kann ein einziger Fehler eine Reihe von Folgefehlern

nach sich ziehen. Durch einen Doppelklick auf eine Fehlermeldung wird die Zeile angezeigt, die den Fehler verursacht hat.

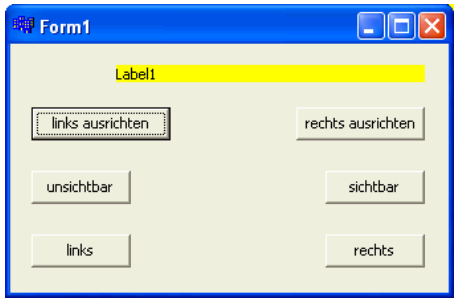
Zu einer solchen Fehlermeldung kann man weitere Informationen erhalten, indem man sie im Fenster „Meldungen“ anklickt und dann die Taste *F1* drückt:

Nach einer solchen Fehlermeldung des Compilers müssen Sie den Fehler im Quelltext beheben. Das kann vor allem für Anfänger eine mühselige Angelegenheit sein, insbesondere wenn die Fehlermeldung nicht so präzise auf den Fehler hinweist wie in diesem Beispiel. Da Fehler Folgefehler nach sich ziehen können, sollte man immer den Fehler zur ersten Fehlermeldung zuerst beheben.

Manchmal sind die **Fehlerdiagnosen** des Compilers sogar eher **irreführend** als hilfreich und schlagen eine falsche Therapie vor. Auch wenn Ihnen das kaum nützt: Betrachten Sie es als kleinen Trost, dass die Fehlermeldungen in anderen Programmiersprachen (z.B. in C) oft noch viel irreführender sind und schon so manchen Anfänger völlig zur Verzweiflung gebracht haben.

Aufgabe 2.3

Schreiben Sie ein Programm, das nach dem Start dieses Fenster anzeigt:



Für die folgenden Ereignisbehandlungsroutinen müssen Sie sich in der Online-Hilfe über einige Eigenschaften informieren, die bisher noch nicht vorgestellt wurden.

Beim Anklicken der Buttons mit der Aufschrift

– *ausrichten* soll der Text im Label mit Hilfe der Eigenschaft *Alignment* (siehe Online-Hilfe, nicht mit *Align* verwechseln) links bzw. rechts ausgerichtet werden.
  Damit das Label sichtbar ist, soll seine Farbe z.B. auf Gelb gesetzt werden.
  Damit die Größe des Labels nicht der Breite des Textes angepasst wird, soll *Autosize* (siehe Online-Hilfe) auf *false* gesetzt werden.
– *sichtbar/unsichtbar* soll das Label sichtbar bzw. unsichtbar gemacht werden,
– *links/rechts* soll das Label so verschoben werden, dass sein linker bzw. rechter Rand auf dem linken bzw. rechten Rand des Formulars liegt. Damit der rechte Rand des Labels genau auf den rechten Rand des Formulars gesetzt wird, verwenden Sie die Eigenschaft *ClientWidth* (siehe Online-Hilfe) eines Formulars.

2.4 Funktionen, Methoden und die Komponente *TEdit*

Eine **Edit-Komponente** kann wie ein Label einen Text des Datentyps *AnsiString* anzeigen. Der angezeigte Text ist der Wert der Eigenschaft *Text*, der wie die Eigenschaft *Caption* bei einem Label im Objektinspektor oder im Programm gesetzt werden kann:

```
Edit1->Text="Hallo";
```

Im Unterschied zu einem Label kann ein Anwender in eine Edit-Komponente auch während der Laufzeit des Programms Text eingeben. Die Eigenschaft *Text* enthält immer den aktuell angezeigten Text und ändert sich mit jeder Eingabe des Anwenders. Dieser Text kann in einem Programm verwendet werden, indem man die Eigenschaft *Text* z.B. auf der rechten Seite einer Zuweisung einsetzt:

```
Label1->Caption = Edit1->Text;
```

Eine Edit-Komponente wird oft als **Eingabefeld** zur Dateneingabe verwendet. Sie übernimmt in einem Windows-Programm oft Aufgaben, die in einem C++-Konsolenprogramm von *cin*>> ... und *cout*<< ... und in einem C-Programm von *printf* und *scanf* übernommen werden.

Die Edit-Komponente liefert eine Eingabe immer als *AnsiString*. Da man oft auch Werte eines anderen Datentyps (z.B. Zahlen) einlesen oder ausgeben will, werden jetzt einige der zahlreichen **Konvertierungsfunktionen** vorgestellt, mit denen man einen *AnsiString* in einen anderen Datentyp umwandeln kann und umgekehrt. In diesem Zusammenhang werden dann auch einige Unterschiede zwischen verschiedenen Arten von Funktionen (globale Funktionen und Methoden) gezeigt.

Für die Umwandlung von *int*-Werten stehen die beiden **globalen Funktionen** *StrToInt* und *IntToStr* zur Verfügung, die in der Online-Hilfe etwa folgendermaßen beschrieben werden:

int ***StrToInt****(AnsiString S);*

Wenn der als Argument übergebenen String eine ganze Zahl darstellt, ist der Funktionswert diese Zahl als *int*-Wert. Stellt der String keine Zahl dar, erfolgt eine Fehlermeldung.

AnsiString ***IntToStr****(int Value);*

Der Funktionswert ist ein String, der die als Argument übergebene Zahl darstellt.

Das Schema, nach dem hier die Funktionen *StrToInt* und *IntToStr* beschrieben sind, wird in C++ üblicherweise zur Beschreibung von Funktionen verwendet. Es wird auch als **Funktionsdeklaration**, Funktions-Prototyp, Funktions-Header oder einfach **Header** bezeichnet. Die einzelnen Elemente bedeuten:

- Der Bezeichner vor den Klammern ist der **Name der Funktion** (hier *StrToInt* und *IntToStr*). Mit diesem Namen wird die Funktion aufgerufen.
- Der Datentyp vor dem Namen der Funktion ist der **Datentyp des Funktionswertes** oder der **Rückgabetyp** (hier *int* bzw. *AnsiString*). Ein Funktionsaufruf ist ein Ausdruck dieses Datentyps. Wenn der Rückgabetyp ***void*** ist, kann dieser Funktionswert nicht in einem Ausdruck, z.B. auf der rechten Seite einer Zuweisung, verwendet werden. Die Funktion kann dann nur aufgerufen werden.
- Der Teil zwischen den Klammern nach dem Funktionsnamen ist die **Parameterliste**. Beim Aufruf einer Funktion muss normalerweise für jeden Parameter ein Argument des Datentyps aus der Parameterliste eingesetzt werden. Falls die Parameterliste leer ist oder nur aus *void* besteht, darf beim Aufruf der Funktion kein Argument angegeben werden.
  Der Unterschied zwischen den Begriffen „**Parameter**“ und „**Argument**“ ist im C++-Standard folgendermaßen definiert: „Parameter“ wird bei einer Funktionsdeklaration verwendet und „Argument“ bei einem Funktionsaufruf.
- Manche Funktionsbeschreibungen enthalten Angaben wie ***__fastcall***, ***extern*** oder ***virtual***. Diese Angaben haben keine Auswirkungen darauf, wie die Funktion aufgerufen werden kann und werden später erklärt.

Deshalb kann man die **Funktionen** *StrToInt* und *IntToStr* folgendermaßen **aufrufen**: Nach dem Namen der Funktion *StrToInt* wird in Klammern der umzuwandelnde String angegeben. Dieser Ausdruck hat den Datentyp *int*. Entsprechend wird nach dem Namen *IntToStr* in Klammern ein *int*-Ausdruck angegeben, der in einen String umgewandelt werden soll. Dieser Ausdruck hat den Datentyp *AnsiString*.

Beispiel: In einem Formular mit zwei Edit-Fenstern haben die beiden Ausdrücke

StrToInt(Edit1->Text) und
StrToInt(Edit2->Text)

den Datentyp *int*. Mit ihnen man im Unterschied zu den Strings *Edit1->Text* und *Edit2->Text* auch rechnen:

```
StrToInt(Edit1->Text) + StrToInt(Edit2->Text)
```

ist die Summe der Zahlen in den beiden Edit-Fenstern. Diese Summe kann man nun in einem weiteren Edit-Fenster *Edit3* ausgeben, wenn man sie in einen *AnsiString* umwandelt. Da man Funktionsaufrufe beliebig verschachteln kann, hat man mit

```
Edit3->Text=IntToStr(StrToInt(Edit1->Text) +
                              StrToInt(Edit2->Text));
```

bereits ein einfaches Programm zur Addition von Zahlen geschrieben, wenn man diese Anweisung beim Anklicken eines Buttons ausführt:

```
void __fastcall TForm1::Button1Click(TObject
                                              *Sender)
{
Edit3->Text=IntToStr(StrToInt(Edit1->Text) +
StrToInt(Edit2->Text));
}
```

Eine Funktion, die zu einer Komponente gehört, wird auch als **Methode** bezeichnet. Sie wird aufgerufen, indem man ihren Namen nach dem Namen der Komponente und dem Pfeiloperator -> angibt. Darauf folgen in runden Klammern die Argumente.

Beispiel: Die Methode *Clear* der Komponente *TEdit*

virtual void ***Clear****();* // aus der Online-Hilfe zu *TEdit*

wurde schon in Abschnitt 2.1 vorgestellt. Sie wird über eine Komponente der Klasse *TEdit* (z.B. *Edit1*) aufgerufen:

```
Edit1->Clear();
```

Da die Parameterliste in ihrer Deklaration leer ist, wird sie ohne Argumente aufgerufen.

Wenn eine Funktion **Parameter** hat, muss bei ihrem Aufruf normalerweise für jeden Parameter ein Argument übergeben werden. Der Datentyp des Arguments ist im einfachsten Fall der des Parameters.

Beispiel: Mit der für viele Komponenten definierten Methode

virtual void ***SetBounds****(int ALeft, int ATop, int AWidth,int AHeight);*

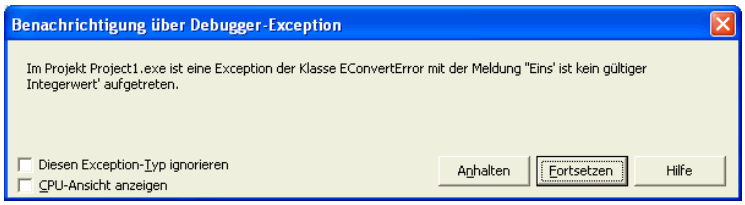
kann man die Eigenschaften *Left*, *Top*, *Width* und *Height* der Komponente mit einer einzigen Anweisung setzen. Die Größe und Position eines Edit-Fensters *Edit1* kann deshalb so gesetzt werden:

```
Edit1->SetBounds(0,0,100,20);
```

Manche Funktionen können mit unzulässigen Argumenten aufgerufen werden. So kann man z.B. die Funktion *StrToInt* mit einem String aufrufen, der wie in

```
StrToInt("Eins"); // das geht schief
```

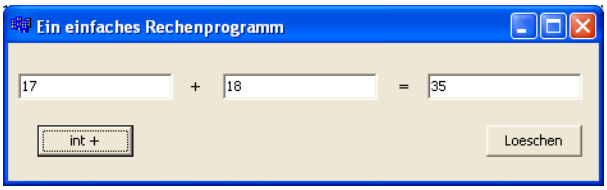
nicht in eine Zahl umgewandelt werden kann. Dann erhält man die **Fehlermeldung:**



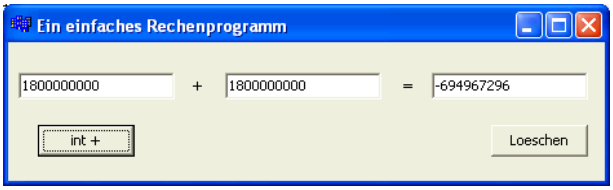
Eine solche Meldung enthält eine Beschreibung der Fehlerursache, hier „'Eins' ist kein gültiger Integerwert". Durch Anklicken des Buttons ***Fortsetzen*** kann man das Programm fortsetzen.

Aufgaben 2.4

1. Schreiben Sie ein einfaches Rechenprogramm, mit dem man zwei Ganzzahlen addieren kann. Durch Anklicken des *Löschen*-Buttons sollen sämtliche Eingabefelder gelöscht werden.



Offensichtlich produziert dieses Programm falsche Ergebnisse, wenn die Summe außerhalb des Bereichs –2147483648 .. 2147483647 ($-2^{31}..2^{31}-1$) liegt:



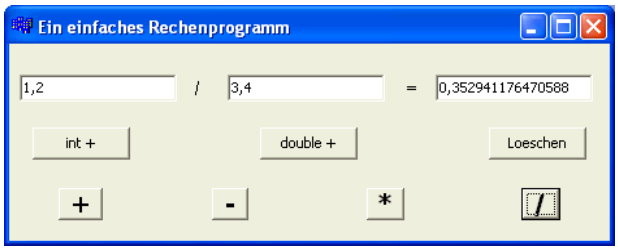
Die Ursache für diese Fehler werden wir später kennen lernen.

2. Ergänzen Sie das Programm aus Aufgabe 1 um einen Button, mit dem auch Zahlen mit Nachkommastellen wie z.B. 3,1415 addiert werden können. Verwenden Sie dazu die Funktionen

   long double ***StrToFloat****(AnsiString S);*
   AnsiString ***FloatToStr****(long double Value);*
   // weitere Informationen dazu in der Online-Hilfe

   Der Datentyp *long double* ist einer der Datentypen, die in C++ Zahlen mit Nachkommastellen darstellen können. Solche Datentypen haben einen wesentlich größeren Wertebereich als der Ganzzahldatentyp *int*. Deshalb treten Bereichsüberschreitungen nicht so schnell auf.

3. Ergänzen Sie das Programm aus Aufgabe 2 um Buttons für die Grundrechenarten +, –, * und /. Die Aufschrift auf den Buttons soll im Objektinspektor über die Eigenschaft ***Font*** auf 14 Punkt und fett (Font|Style *fsBold*) gesetzt werden. Die jeweils gewählte Rechenart soll in einem Label zwischen den beiden Operanden angezeigt werden:



4. Geben Sie im laufenden Programm im ersten Eingabefeld einen Wert ein, der nicht in eine Zahl umgewandelt werden kann, und setzen Sie das Programm anschließend fort.

2.5 Memos, ListBoxen, ComboBoxen und die Klasse *TStrings*

Eine wichtige Kategorie von Datentypen sind **Klassen**. Im Unterschied zu elementaren Datentypen wie *int* können Klassen Eigenschaften, Daten, Methoden und

Ereignisse enthalten. Klassen sind die Grundlage der sogenannten **objektorientierten Programmierung**. Dabei werden Programme aus Bausteinen (Klassen) zusammengesetzt, die wiederum Elemente eines Klassentyps enthalten können.

Wir haben Klassen bisher schon als Datentypen der Komponenten der Tool-Palette kennen gelernt: Alle Komponenten der Tool-Palette haben einen Datentyp, der eine Klasse ist. Die Bibliotheken des C++Builders enthalten zahlreiche weitere Klassen, die oft als Eigenschaften dieser Komponenten verwendet werden. Im Folgenden wird vor allem gezeigt,

– wie man Elemente von Eigenschaften eines Klassentyps anspricht, und
– dass Klassen oft viele Gemeinsamkeiten mit anderen Klassen haben.

In einem **Memo** kann man wie in einer Edit-Komponente Text aus- und eingeben. Im Unterschied zu einer Edit-Komponente kann ein Memo aber nicht nur einzeilige, sondern auch mehrzeilige Texte enthalten. Der im Memo angezeigte Text ist der Wert seiner Eigenschaft *Text*:

```
Memo1->Text="Dieser Text ist breiter als das Memo";
```

Dieser Text wird dann in Abhängigkeit von der Größe des Memos und der Schriftart (über die Eigenschaft *Font*) in Zeilen aufgeteilt. Die einzelnen Zeilen können über die Eigenschaft *Lines->Strings*[0] (die erste Zeile), *Lines->Strings*[1] usw. angesprochen werden:

```
Edit1->Text = Memo1->Lines->Strings[0];
```

Die Eigenschaft ***Lines*** hat den Datentyp ***TStrings****:

__property TStrings **Lines** …;*

Der Datentyp *TStrings* ist eine **Klasse**, die unter anderem die folgenden Elemente enthält:

virtual int ***Add****(AnsiString S);* // fügt das Argument für S am Ende ein
virtual void ***Insert****(int Index, AnsiString S);* // fügt das Argument für S an der Position *index* ein

Ein Element einer Eigenschaft eines Klassentyps wird wie das Element einer Komponente angesprochen: Nach dem Namen der Eigenschaft (z.B. *Memo1->Lines*) gibt man den Pfeiloperator „–>“ und den Namen des Elements (z.B. der Methode *Add*) an.

Beispiel: Ein Aufruf von *Add* fügt das Argument am Ende des Memos als neue Zeile ein:

```
Memo1->Lines->Add("Neue Zeile am Ende von Memo1");
```

Ein Aufruf von *Insert* mit dem Argument 0 für *Index* fügt eine Zeile am Anfang des Memos ein:

```
Memo1->Lines->Insert(0,"Neue Zeile vorne");
```

Memos und die Methode *Add* werden oft zur Anzeige der Ergebnisse eines Programms verwendet.

Die Eigenschaft *Count* von *Lines* enthält die Anzahl der Zeilen des Memos:

__property int ***Count****;*

Mit der zu *TStrings* gehörenden Methode

virtual void ***LoadFromFile****(AnsiString FileName);*

kann man einen Text, der als Datei vorliegt, in ein Memo einlesen. Diese Datei sollte keine Steuerzeichen enthalten (wie z.B. *.doc-Dokumente, die mit Microsoft Word erzeugt wurden) und darf maximal 32 KB groß sein. Der Text eines Memos kann mit

virtual void ***SaveToFile****(AnsiString FileName);*

als Datei gespeichert werden.

Beispiel: Bei Strings, die das Zeichen „\" enthalten, ist zu beachten, dass dieses Zeichen in C++ eine besondere Bedeutung hat (Escape-Sequenz). Deswegen muss es immer doppelt angegeben werden, wenn es wie bei einem Pfadnamen in einem String enthalten sein soll:

```
void __fastcall TForm1::Button1Click(
                                      TObject *Sender)
{
Memo1->Lines->LoadFromFile("c:\\config.sys");
}
```

Da man ein Memo wie einen Editor verwenden und den Text verändern kann, sollte man den Namen „c:\config.sys" nicht als *FileName* beim Speichern verwenden. Nach einer Veränderung dieser Datei könnte sonst der nächste Start Ihres Rechners mit einer unangenehmen Überraschung verbunden sein.

Die Strings in eine Eigenschaft des Typs *TStrings* können nicht zur Laufzeit mit Funktionen wie *Add* eingegeben werden, sondern auch zur Entwurfszeit nach einem Doppelklick auf die Eigenschaft im Objektinspektor. Dann öffnet sich der **String-Listen-Editor**, mit dem man die Zeilen einfach eintragen kann. Damit kann man auch den Standardtext „Memo1" aus einem Memo entfernen.

Eine **ListBox** zeigt wie ein Memo Textzeilen an. Diese können aber im Unterschied zu einem Memo vom Anwender nicht verändert werden. ListBoxen werden vor allem dazu verwendet, eine Liste von Optionen anzuzeigen, aus denen der Anwender eine auswählen kann.

Die angezeigten Zeilen sind die Zeilen der Eigenschaft ***Items***, die ebenfalls den Datentyp *TStrings* hat. Deshalb hat die Eigenschaft *Items* einer ListBox dieselben Elemente (Eigenschaften, Methoden und Ereignisse) wie die Eigenschaft *Lines* eines Memos.

Beispiel: Die Beispiele mit *Memo1->Lines* lassen sich auf eine ListBox *ListBox1* übertragen, indem man *Memo1->Lines* durch *ListBox1->Items* ersetzt.

Eine **ComboBox** besteht im Wesentlichen aus einer ListBox und einem Eingabefeld. Die ListBox wird nach dem Anklicken des rechten Dreiecks aufgeklappt. Aus ihr kann ein Eintrag ausgewählt werden, der dann in das Eingabefeld übernommen wird und da editiert werden kann.

Die Zeilen der ComboBox sind wie bei einer ListBox der Wert der Eigenschaft *Items* vom Typ *TStrings*. Der Text im Eingabefeld der ComboBox wird durch die Eigenschaft *Text* des Datentyps *AnsiString* dargestellt.

Beispiel: Den Text *ComboBox1->Text* im Eingabefeld der ComboBox kann man wie die Eigenschaft *Text* einer Edit-Komponente verwenden, und die *TStrings*-Eigenschaft *ComboBox1->Items* wie die einer ListBox:

Da sich die mit einer Eigenschaft zulässigen Operationen allein aus ihrem **Datentyp** ergeben, kann man zwei verschiedene Eigenschaften desselben Datentyps auf dieselbe Art verwenden. Ist dieser Datentyp eine Klasse, haben beide Eigenschaften dieselben Elemente (Eigenschaften, Methode und Ereignisse), die ebenfalls auf dieselbe Art verwendet werden können. Wenn eine Eigenschaft, die Sie noch nicht kennen, denselben Datentyp hat wie eine Ihnen schon bekannte Eigenschaft, können Sie die neue Eigenschaft genauso verwenden wie die bereits bekannte, ohne dass Sie irgendwelche Besonderheiten lernen müssen.

Beispiel: Wenn Sie die Klasse *TStrings* aus der Arbeit mit der Eigenschaft *Lines* der Memo-Komponente kennen, können Sie mit einer Eigenschaft dieses Typs in jeder anderen Komponente genauso arbeiten. Wenn Sie also z.B. in der Online-Hilfe sehen, dass die Eigenschaft *Text* einer *TSendMail*-Komponente den Datentyp *TStrings* hat, können Sie mit dieser Komponente genauso arbeiten.

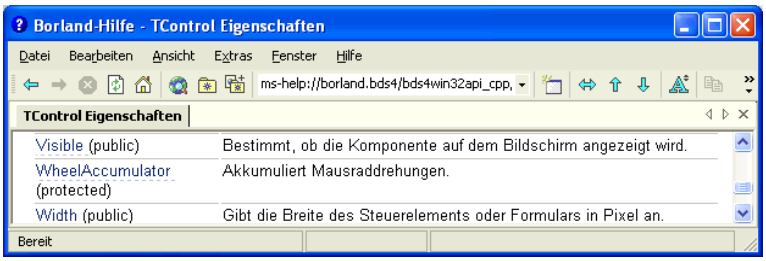
Das gilt nicht nur für die Operationen mit einer Komponente im Programm, sondern auch für die Operationen im Objekt Inspektor: Nach dem Anklicken einer Eigenschaft des Typs *TStrings* im Objekt Inspektor wird der String-Listen-Editor geöffnet.

Oft stellt eine Klasse Gemeinsamkeiten verschiedener Klassen dar. Dann enthält sie die gemeinsamen Elemente der spezielleren Klassen. In der objektorientierten Programmierung werden solche Gemeinsamkeiten durch **Vererbung** zum Ausdruck gebracht. Vererbung bedeutet, dass eine abgeleitete Klasse (die Klasse, die erbt) alle Elemente einer Basisklasse übernimmt. Die Online-Hilfe zeigt die Vererbungshierarchie für jede Klasse im Abschnitt „Hierarchie“ an.

Beispiel: Die Klasse *TComboBox* erbt von der Klasse *TCustomComboBox*, diese wiederum von *TCustomCombo* usw.:

Die Klasse *TComboBox* enthält deshalb alle Elemente von *TCustomListControl*. Da eine *TListBox* ebenfalls von *TCustomListControl* erbt, kann man die gemeinsamen Elemente einer ListBox und einer ComboBox auf dieselbe Art verwenden.

TCustomListControl erbt wiederum von *TControl*. Diese Klasse enthält die gemeinsamen Eigenschaften, Methoden und Ereignisse aller Steuerelemente (Controls). Dazu gehören z.B. Eigenschaften wie *Visible*, die schon in Abschnitt 2.3 vorgestellt wurde.



Deshalb gilt alles, was für die Eigenschaft *Visible* in Abschnitt 2.3 gesagt wurde, auch für die Eigenschaft *Visible* jeder anderen Klasse, die diese Eigenschaft von der Klasse *TControl* erbt.

Die Klasse ***TObject*** ist die Basisklasse aller Komponenten.

Eine abgeleitete Klasse enthält meist noch zusätzliche Elemente, die die Unterschiede zur Basisklasse ausmachen. Einige zusätzliche Elemente einer **ListBox**, die nicht in der Basisklasse *TWinControl* enthalten sind:

Falls ein Benutzer einen Eintrag ausgewählt hat, steht der Index dieses Eintrags unter der *int*-Eigenschaft ***ItemIndex*** zur Verfügung (0 für den ersten Eintrag). Falls kein Eintrag ausgewählt wurde, hat *ItemIndex* den Wert –1. Der ausgewählte Eintrag ist deshalb

```
ListBox1->Items->Strings[ListBox1->ItemIndex]
```

Ob ein Eintrag ausgewählt wurde, kann man auch mit der booleschen Eigenschaft ***Selected*** prüfen.

Setzt man die boolesche Eigenschaft ***Sorted*** auf *true*, werden die Einträge alphanumerisch sortiert angezeigt.

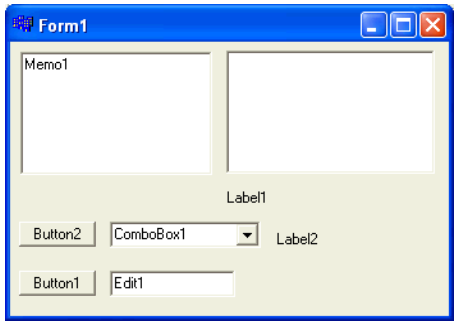
Dieser Abschnitt sollte insbesondere auch zeigen, dass die zahlreichen Komponenten doch nicht so unüberschaubar viele verschiedene Eigenschaften und Methoden haben, wie man das auf den ersten Blick vielleicht befürchtet. In der **objektorientierten Programmierung** werden Programme aus Bausteinen (Klassen) zusammengesetzt. Mit einer geschickt konstruierten Klassenbibliothek kann man aus relativ wenigen Klassen Programme für eine Vielzahl von Anwendungen entwickeln. Die Wiederverwendung der Klassen erleichtert den Überblick und den Umgang mit den Komponenten beträchtlich.

Anmerkungen für Delphi-Programmierer: In Delphi kann man die einzelnen Zeilen eines Memos auch ohne die *TStrings*-Eigenschaft ansprechen:

```
Edit1.Text := Memo1.Lines[0]; // Delphi
Edit1->Text = Memo1->Lines->Strings[0]; // C++Builder
```

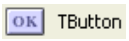
Aufgabe 2.5

Schreiben Sie ein Programm mit einem Memo, einer ListBox, einer ComboBox, einem Edit-Fenster, zwei Buttons und zwei Labels:



a) Beim Anklicken von *Button1* soll der aktuelle Text des Edit-Fensters als neue Zeile zu jeder der drei *TStrings*-Listen hinzugefügt werden.
b) Wenn ein *ListBox*-Eintrag angeklickt wird, soll er auf Label1 angezeigt werden.
c) Beim Anklicken von *Button2* soll der in der ComboBox ausgewählte Text auf dem *Label2* angezeigt werden.

2.6 Buttons und Ereignisse



Ein Button ermöglicht einem Anwender, die Ausführung von Anweisungen zu starten. Durch einen einfachen Mausklick auf den Button werden die für das Ereignis *OnClick* definierten Anweisungen ausgeführt.

Buttons werden oft in **Dialogfenstern** (wie z.B. *Datei|Öffnen* oder *Datei|Speichern unter*) verwendet, über die ein Programm Informationen mit einem Benutzer austauscht. Solche Fenster enthalten meist einen „Abbrechen“-Button, mit dem das Fensters ohne weitere Aktionen geschlossen werden kann, und einen „OK“-Button, mit dem die Eingaben bestätigt und weitere Anweisungen ausgelöst werden.

Ein Button kann auf die folgenden Ereignisse reagieren:

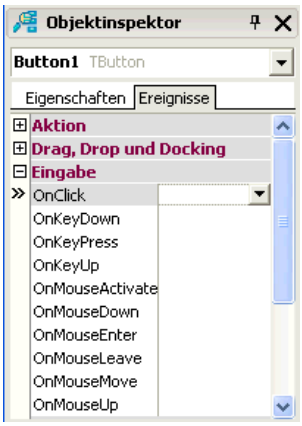
Ereignis	Ereignis tritt ein
OnClick	wenn der Anwender die Komponente mit der Maus anklickt (d.h. die linke Maustaste drückt und wieder loslässt), oder wenn der Button den Fokus hat (siehe Abschnitt 2.6.2) und die Leertaste, *Return*- oder *Enter*-Taste gedrückt wird.
OnMouseDown *OnMouseUp*	wenn eine Maustaste gedrückt bzw. wieder losgelassen wird, während der Mauszeiger über der Komponente ist.
OnMouseMove	wenn der Mauszeiger über die Komponente bewegt wird.
OnKeyPress	wenn eine Taste auf der Tastatur gedrückt wird, während die Komponente den Fokus hat.

Ereignis	Ereignis tritt ein
	Dieses Ereignis tritt im Unterschied zu den nächsten beiden nicht ein, wenn eine Taste gedrückt wird, die keinem ASCII-Zeichen entspricht, wie z.B. eine Funktionstaste (F1 usw.), die Strg-Taste, die Umschalttaste (für Großschreibung) usw.
OnKeyUp *OnKeyDown*	wenn eine beliebige Taste auf der Tastatur gedrückt wird, während die Komponente den Fokus hat. Diese Ereignisse treten auch dann ein, wenn man die Alt-, AltGr-, Shift-, Strg- oder Funktionstasten allein oder zusammen mit anderen Tasten drückt.
OnEnter	wenn die Komponente den Fokus erhält.
OnExit	wenn die Komponente den Fokus verliert.
OnStartDrag *OnDragOver* *OnDragDrop*	wenn der Anwender – damit beginnt, das Objekt zu ziehen, – über die Komponente zieht, – ein gezogenes Objekt abgelegt

Diese Ereignisse werden von der Klasse *TWinControl* geerbt. Da diese Klasse eine gemeinsame Basisklasse zahlreicher Steuerelemente ist, stehen sie nicht nur für einen Button, sondern auch für zahlreiche andere Komponenten zur Verfügung.

2.6.1 Parameter der Ereignisbehandlungsroutinen

Die Ereignisse, die für die auf dem Formular ausgewählte Komponente eintreten können, werden im Objektinspektor nach dem Anklicken des **Registers Ereignisse** angezeigt.



Mit einem Doppelklick auf die rechte Spalte eines Ereignisses erzeugt der C++Builder die Funktion (Ereignisbehandlungsroutine, engl. „event handler“), die bei diesem Ereignis aufgerufen wird. Sie wird dann im **Quelltexteditor** angezeigt, wobei der Cursor am Anfang der Funktion steht.

Für das Ereignis *OnKeyPress* von *Button1* erhält man diese Ereignisbehandlungsroutine:

```
void __fastcall TForm1::Button1KeyPress(TObject *Sender,
                                        char &Key)
{
}
```

An eine Ereignisbehandlungsroutine werden über die **Parameter** (zwischen den runden Klammern) Daten übergeben, die in Zusammenhang mit dem Ereignis zur Verfügung stehen. Vor jedem Parameter steht sein Datentyp.

Beispiel: In der Funktion *Button1KeyPress* hat der Parameter *Sender* (den wir allerdings vorläufig nicht verwenden) den Datentyp *TObject** und der Parameter *Key* den Datentyp *char*.

Der Parameter *Key* von *Button1KeyPress* enthält das Zeichen der Taste, die auf der Tastatur gedrückt wurde und so das Ereignis *OnKeyPress* ausgelöst hat. Dieses Zeichen kann in der Funktion unter dem Namen *Key* verwendet werden:

```
void __fastcall TForm1::Button1KeyPress(
                        TObject *Sender, char &Key)
{ // gibt das Zeichen Key in Edit1->Text aus
Edit1->Text=Key;
}
```

Bei den Ereignissen ***OnKeyDown*** und ***OnKeyUp*** werden die Werte der Zeichen im Parameter *Key* als sogenannte virtuelle Tastencodes übergeben. Ihre Bedeutung findet man in der Online-Hilfe des C++Builders unter dem Stichwort „virtual-key codes“ (beim C++Builder 6 unter „virtuelle Tastencodes“).

Um Anweisungen beim **Erzeugen** eines Formulars auszuführen, hat man in der Version 1 des C++Builders die *OnCreate* Ereignisbehandlungsroutine verwendet. Für neuere Versionen empfiehlt die Online-Hilfe, stattdessen den Konstruktor des Formulars zu verwenden. Diese Funktion findet man am Anfang einer Unit. Setzt man hier irgendwelche Eigenschaften, hat das im Wesentlichen denselben Effekt wie wenn man sie im Objektinspektor setzt.

```
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
Edit1->Color=clRed;
}
```

Beim **Schließen** eines Formulars (sowohl nach dem Aufruf der Methode *Close* als auch nach dem Anklicken des Schließen-Buttons ) wird die Ereignis ***OnClose*** ausgelöst, in dessen Ereignisbehandlungsroutine man z.B. fragen kann, ob man Änderungen speichern will. Um das Schließen des Formulars abzubrechen, setzt man den zweiten Parameter *Action* in der Ereignisbehandlungsroutine auf *caNone*.

2.6.2 Der Fokus und die Tabulatorreihenfolge

Ereignisse, die durch die Maus ausgelöst werden (z.B. *OnClick* oder *OnMouseMove*), werden immer dem Steuerelement zugeordnet, über dem sich der Mauszeiger gerade befindet. Bei Tastaturereignissen (wie z.B. *OnKeyPress*) ist diese Zuordnung nicht möglich. Sie werden dem Steuerelement zugeordnet, das gerade den **Fokus** hat. Ein Steuerelement erhält z.B. durch Anklicken oder durch wiederholtes Drücken der Tab-Taste den Fokus. In jedem Formular hat immer nur ein Steuerelement den Fokus. Es wird auch als das gerade **aktive** Steuerelement be-

zeichnet. Ein Button, der den Fokus hat, wird durch einen schwarzen Rand optisch hervorgehoben.

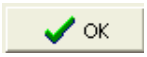
Wurde während der Laufzeit eines Programms noch kein Steuerelement als aktives Steuerelement ausgewählt, hat das erste in der **Tabulatorreihenfolge** den Fokus. Die Tabulatorreihenfolge ist die Reihenfolge, in der die einzelnen Steuerelemente durch Drücken der Tab-Taste den Fokus erhalten. Falls diese Reihenfolge nicht explizit (zum Beispiel über die Eigenschaft *TabOrder* bzw. über das Kontextmenü) gesetzt wurde, entspricht sie der Reihenfolge, in der die Steuerelemente während der Entwurfszeit auf das Formular gesetzt wurden.

Von der Aktivierung über die Tab-Taste sind die Steuerelemente ausgenommen,

– die deaktiviert sind (die Eigenschaft *Enabled* hat den Wert *false*),
– die nicht sichtbar sind (die Eigenschaft *Visible* hat den Wert *false*),
– bei denen die Eigenschaft *TabStop* den Wert *false* hat.

2.6.3 BitButtons und einige weitere Eigenschaften von Buttons

Falls man einen mit einem Bild verzierten Button möchte, muss man einen **BitBtn** (Bitmap-Button, aus der Kategorie „Zusätzlich“ der Tool-Palette) verwenden. Er unterscheidet sich von einem Button im Wesentlichen nur durch die zusätzliche Grafik und kann wie ein gewöhnlicher Button verwendet werden. Über die Eigenschaft *Kind* kann einige gebräuchliche Kombinationen von Grafik/Text-Kombinationen auswählen wie



Weitere Bitmaps können während der Entwurfszeit durch einen Doppelklick auf die rechte Spalte der Eigenschaft *Glyph* im Objektinspektor ausgewählt werden oder während der Laufzeit des Programms mit der Methode *LoadFromFile*:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
BitBtn1->Glyph->LoadFromFile("C:\\Programme\\Gemeinsame
Dateien\\Borland Shared\\Images\\Buttons\\alarm.bmp");
}
```

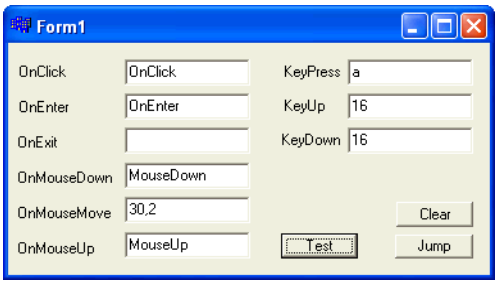
Das Verzeichnis „C:\Programme\Gemeinsame Dateien\Borland Shared\Images\-Buttons“ enthält zahlreiche Bitmaps, die oft für Buttons verwendet werden.

Bei manchen Formularen soll das Drücken der ESC-Taste bzw. der *Return*- oder *Enter*-Taste denselben Effekt wie das Anklicken eines Abbrechen-Buttons oder OK-Buttons haben (wie z.B. bei einem Datei-Öffnen Dialog). Das erreicht man über die Eigenschaften *Default* und *Cancel* eines Buttons:

– Wenn ***Default*** den Wert *true* hat, tritt bei diesem Button das Ereignis *OnClick* auf, wenn die *Return*- oder *Enter*-Taste gedrückt wird, auch ohne dass der Button den Fokus hat. Ein solcher Button wird auch als **Accept-Button** bezeichnet.
– Wenn ***Cancel*** den Wert *true* hat, tritt bei diesem Button das Ereignis *OnClick* auf, wenn die *ESC-Taste* gedrückt wird. Ein solcher Button wird auch als **Cancel-Button** bezeichnet.

Aufgabe 2.6

Schreiben Sie ein Programm, das etwa folgendermaßen aussieht:



a) Wenn für den „Test“-Button eines der Ereignisse *OnClick*, *OnEnter* usw. eintritt, soll ein Text in das zugehörige Edit-Fenster geschrieben werden. Für die Ereignisse, für die hier keine weiteren Anforderungen gestellt werden, reicht ein einfacher Text, der das Ereignis identifiziert (z.B. „OnClick“).

Bei den Key-Ereignissen soll der Wert des Parameters *Key* angezeigt werden. Beachten Sie dabei, dass dieser Parameter bei der Funktion

```
void __fastcall TForm1::Button1KeyPress(TObject *Sender,
                                        char &Key)
```

den Datentyp *char* hat und der Eigenschaft *Text* des Edit-Fensters direkt zugewiesen werden kann, während er bei den Funktionen *KeyDown* und *KeyUp* den Datentyp *WORD* hat:

```
void __fastcall TForm1::Button1KeyDown(TObject *Sender,
                                WORD &Key, TShiftState Shift)
```

Dieser Datentyp kann mit *IntToStr* in einen String umgewandelt und dann der Eigenschaft *Text* des Edit-Fensters zugewiesen werden.

Beim Ereignis *MouseMove* sollen die Mauskoordinaten angezeigt werden, die als Parameter X und Y übergeben werden. Diese können mit *IntToStr* in einen String umgewandelt und mit + zu einem String zusammengefügt werden. Als Trennzeichen soll dazwischen noch mit + ein String "," eingefügt werden.

b) Mit dem Button *Clear* sollen alle Anzeigen gelöscht werden können.

c) Beobachten Sie, welche Ereignisse eintreten, wenn der Test-Button
   – angeklickt wird
   – den Fokus hat und eine Taste auf der Tastatur gedrückt wird
   – den Fokus hat und eine Funktionstaste (z.B. F1, Strg, Alt) gedrückt wird
   – mit der Tab-Taste den Fokus bekommt.

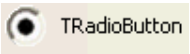
d) Die Tabulatorreihenfolge der Edit-Fenster soll ihrer Reihenfolge auf dem Formular entsprechen (zuerst links von oben nach unten, dann rechts).

e) Der Text der Titelzeile „Events“ des Formulars soll im Konstruktor des Formulars zugewiesen werden.

f) Der „Jump“-Button soll immer an eine andere Position springen (z.B. an die gegenüberliegende Seite des Formulars), wenn er vom Mauszeiger berührt wird. Dazu ist keine *if*-Anweisung notwendig. Falls er angeklickt wird, soll seine Aufschrift auf „getroffen“ geändert werden.

2.7 CheckBoxen, RadioButtons und einfache *if*-Anweisungen

Eine **CheckBox** besteht im Wesentlichen aus einer Aufschrift (Eigenschaft *Caption*) und einem Markierungsfeld, dessen Markierung ein Anwender durch einen Mausklick oder mit der Leertaste (wenn sie den Fokus hat) setzen oder aufheben kann. Ihre boolesche Eigenschaft *Checked* ist *true*, wenn sie markiert ist, und sonst *false*. Falls sich auf einem Formular oder in einem Container (siehe Abschnitt 2.8) mehrere CheckBoxen befinden, können diese unabhängig voneinander markiert werden.



Ein **RadioButton** hat viele Gemeinsamkeiten mit einer CheckBox. Er besitzt ebenfalls eine Aufschrift (Eigenschaft *Caption*) und ein Markierungsfeld (Eigenschaft *Checked*), das mit der Maus markiert werden kann. Der entscheidende Unterschied zu CheckBoxen zeigt sich, sobald ein Formular oder ein Container mehr als einen RadioButton enthält: Wie bei den Sender-Stationstasten eines Radios kann immer nur einer der RadioButtons markiert sein. Markiert man einen anderen, wird bei dem bisher markierten die Markierung aufgehoben. Befindet sich nur ein einziger RadioButton auf einem Formular, kann dessen Markierung nicht zurückgenommen werden.

Beispiel: CheckBoxen und RadioButtons unter *Tools*|*Optionen*|*Tool-Palette*:

CheckBoxen und RadioButtons werden vor allem dazu verwendet, einem Anwender Optionen zur Auswahl anzubieten. Falls **mehrere** Optionen **gleichzeitig** ausgewählt werden können, verwendet man eine CheckBox. RadioButtons verwendet man dagegen bei Optionen, die sich **gegenseitig ausschließen**. Ein einziger RadioButton auf einem Formular macht im Unterschied zu einer einzigen CheckBox wenig Sinn.

Sie werden außerdem zur Anzeige von Daten mit zwei Werten (z.B. „schreibgeschützt ja/nein“) verwendet. Falls die Daten nur angezeigt werden sollen, ohne veränderbar zu sein, setzt man die boolesche Eigenschaft ***Enabled*** auf *false*. Der zugehörige Text wird dann grau dargestellt.

Die Auswahl der Optionen soll meist den Programmablauf steuern. Falls eine Option markiert ist, sollen z.B. beim Anklicken eines Buttons bestimmte Anweisungen ausgeführt werden, und andernfalls andere. Anders als bei einem Button werden beim Anklicken einer CheckBox meist keine Anweisungen ausgeführt, obwohl das mit entsprechenden Anweisungen beim Ereignis *OnClick* auch durchaus möglich ist.

Zur Steuerung des Programmablaufs steht die ***if*-Anweisung** zur Verfügung. Bei ihr gibt man nach *if* in runden Klammern einen booleschen Ausdruck an:

```
if (RadioButton1->Checked) Label1->Caption="Glückwunsch";
else Label1->Caption = "Pech gehabt";
```

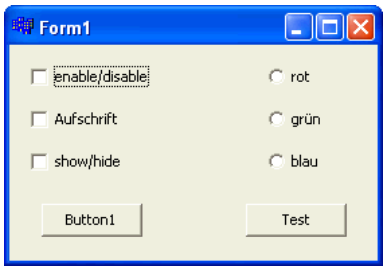
Bei der Ausführung dieser *if*-Anweisung wird zuerst geprüft, ob *RadioButton1->Checked* den Wert *true* hat. Trifft dies zu, wird die folgende Anweisung ausgeführt und andernfalls die auf *else* folgende. Bei einer *if*-Anweisung ohne *else*-Zweig wird nichts gemacht, wenn die Bedingung nicht erfüllt ist.

Falls mehrere Anweisungen in Abhängigkeit von einer Bedingung ausgeführt werden sollen, fasst man diese mit geschweiften Klammern { } zusammen. Prüfungen auf Gleichheit erfolgen mit dem Operator „==“ (der nicht mit dem Zuweisungsoperator „=“ verwechselt werden darf) und liefern ebenfalls einen booleschen Wert:

```
if (Edit1->Text == "xyz")
  {
    Label1->Caption="Na so was! ";
    Label2->Caption="Sie haben das Passwort erraten. ";
  }
```

Aufgabe 2.7

Schreiben Sie ein Programm mit drei CheckBoxen, zwei RadioButtons und zwei gewöhnlichen Buttons:



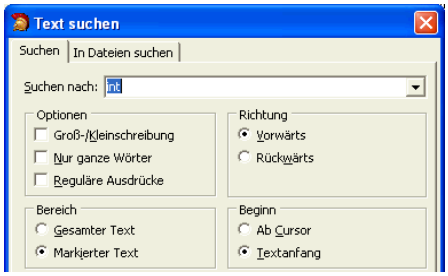
Beim Anklicken des Buttons *Test* sollen in Abhängigkeit von den Markierungen der CheckBoxen und der RadioButtons folgende Aktionen stattfinden:

a) Die Markierung der CheckBox *enable/disable* soll entscheiden, ob bei der zweiten CheckBox, dem ersten RadioButton sowie bei *Button1* die Eigenschaft *Enabled* auf *true* oder *false* gesetzt wird.
b) Die Markierung der CheckBox *Aufschrift* soll entscheiden, welchen von zwei beliebigen Texten *Button1* als Aufschrift erhält.
c) Die Markierung der CheckBox *show/hide* soll entscheiden, ob *Button1* angezeigt wird oder nicht.
d) Die Markierung der RadioButtons soll die Hintergrundfarbe der ersten CheckBox festlegen.
e) Beim Anklicken des ersten RadioButtons soll die Hintergrundfarbe der ersten CheckBox auf rot gesetzt werden.

2.8 Die Container GroupBox, Panel und PageControl

Mit einer **GroupBox** kann man Komponenten auf einem Formular durch einen Rahmen und eine Überschrift (Eigenschaft *Caption*) optisch zu einer Gruppe zusammenfassen. Eine solche Zusammenfassung von Komponenten, die inhaltlich zusammengehören, ermöglicht vor allem die **übersichtliche Gestaltung** von Formularen.

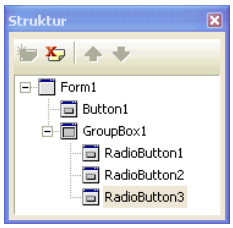
Beispiel: GroupBoxen unter *Suchen|Suchen*:



Die Zugehörigkeit einer Komponente zu einer GroupBox erreicht man am einfachsten, indem man zuerst die GroupBox auf das Formular und dann die Komponente direkt aus der Tool-Palette auf die GroupBox setzt. Dann wird die Komponente automatisch der GroupBox zugeordnet.

Um eine Komponente, die sich bereits auf dem Formular befindet, nachträglich einer GroupBox zuzuordnen, reicht es nicht aus, sie mit der Maus in die GroupBox zu verschieben. Stattdessen muss man eine der folgenden beiden Vorgehensweisen verwenden:

a) Indem man die Komponente in der Struktur-Anzeige (*Ansicht|Struktur*) auf eine GroupBox zieht.



Die Struktur-Anzeige zeigt die Komponenten eines Formulars in ihrer hierarchischen Ordnung an. Hier kann man auch verdeckte Elemente für den Objektinspektor auswählen, verschieben usw.

b) Die unter a) beschriebene Vorgehensweise steht erst seit der Version 6 des C++Builders zur Verfügung. In älteren Versionen muss man die Komponenten zuerst markieren, indem man auf dem Formular einen Punkt anklickt und dann bei gedrückter linker Maustaste ein Rechteck um sie zieht. Dann kann man die so markierten Komponenten mit *Bearbeiten|Kopieren* (bzw. *Ausschneiden, Strg+X*) und *Bearbeiten|Einfügen (Strg+V)* in die GroupBox kopieren.

Ähnlich wie mit einer GroupBox kann man auch mit einem **Panel** Komponenten gruppieren. Im Unterschied zu einer GroupBox verfügt ein Panel über keine Beschriftung. Stattdessen hat es die Eigenschaften *BevelInner* und *BevelOuter*, mit denen ein innerer und äußerer Randbereich so

gestaltet werden kann, dass ein dreidimensionaler Eindruck entsteht, oder dass es überhaupt keinen Rand hat. Diese können die folgenden Werte annehmen:

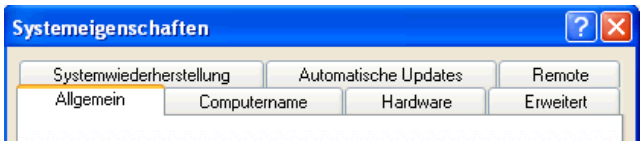
```
bvNone      // kein Effekt der Schräge
bvLowered   // der Rand wirkt abgesenkt
bvRaised    // der Rand wirkt erhöht
```

Die Voreinstellungen für ein Panel sind *bvNone* für *BevelInner* und *bvRaised* für *BevelOuter*, so dass das Panel leicht erhöht wirkt. Setzt man beide auf *bvNone*, ist kein Rand erkennbar. So können Komponenten in einer Gruppe zusammengefasst (und damit gemeinsam verschoben) werden, auch ohne dass die Gruppe als solche erkennbar ist.

Eine **RadioGroup** ist eine GroupBox, die RadioButtons enthalten kann. Sie besitzt die Eigenschaft *Items* des schon bei Memos und ListBoxen vorgestellten Typs *TStrings*: Jedem String von *Items* entspricht ein RadioButton der RadioGroup mit der Aufschrift des Strings. Diese Strings können nach einem Doppelklick auf die Eigenschaft *Items* im Objektinspektor einfach eintragen werden.

Ein **PageControl** (Tool-Palette Kategorie „Win32“) stellt Registerkarten dar, die auch als Seiten bezeichnet werden. Die einzelnen Register sind Komponenten des Datentyps ***TTabSheet***, dessen Eigenschaft *Caption* die Aufschrift auf der Registerlasche ist. Solche Seiten können beliebige Steuerelemente enthalten.

Beispiel: Windows verwendet ein PageControl für die Systemeigenschaften:



Ein PageControl soll oft das gesamte Formular ausfüllen. Das erreicht man, indem man die Eigenschaft *Align* auf *alClient* setzt. Neue Seiten fügt man während der Entwurfszeit über die Option *Neue Seite* im Kontextmenü hinzu.

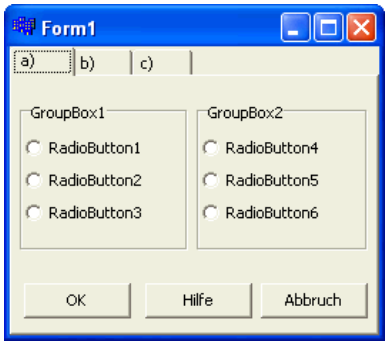
Komponenten, die andere Komponenten enthalten können, nennt man auch **Container-Komponenten**. Von den bisher vorgestellten Komponenten sind Formulare, GroupBox, Panel und RadioGroup solche Container-Komponenten. Die Zugehörigkeit zu einer Container-Komponenten wirkt sich nicht nur optisch aus:

– **Verschiebt** man eine Container-Komponente, werden alle ihre Komponenten mit verschoben (d.h. ihre Position innerhalb des Containers bleibt unverändert). Das kann die Gestaltung von Formularen zur Entwurfszeit erleichtern.

– Bei RadioButtons wirkt sich die gegenseitige Deaktivierung nur auf die RadioButtons in derselben Container-Komponente aus. Das Anklicken eines RadioButtons in einer GroupBox wirkt sich nicht auf die RadioButtons in einer anderen Gruppe aus. So können mehrere Gruppen von sich **gegenseitig ausschließenden Auswahloptionen** auf einem Formular untergebracht werden.
– Die Eigenschaften für die **Position** einer Komponente (*Left* und *Top*) beziehen sich immer auf den Container, in dem sie enthalten sind.
– Die Eigenschaften *Enabled* und *Visible* des Containers wirken sich auf diese Eigenschaften der Elemente aus.

Aufgabe 2.8

Ein Formular soll ein PageControl mit drei Registerkarten enthalten, das das ganze Formular ausfüllt. Die Registerkarten sollen den einzelnen Teilaufgaben dieser Aufgabe entsprechen und die Aufschriften „a)“, „b)“ und „c)“ haben.



a) Die Seite „a)“ soll zwei Gruppen von sich gegenseitig ausschließenden Optionen enthalten.
   Die Buttons *OK*, *Hilfe* und *Abbruch* zu einer Gruppe zusammengefasst werden, die optisch nicht erkennbar ist.
   Reaktionen auf das Anklicken der Buttons brauchen nicht definiert werden.
b) Die Seite „b)“ soll nur einen Button enthalten.
c) Die Seite „c)“ soll leer sein.
d) Verwenden Sie die Struktur-Anzeige, um einen Button von Seite „b)“ auf Seite „c)“ zu verschieben.

2.9 Hauptmenüs und Kontextmenüs

Unter Windows werden einem Anwender die verfügbaren Befehle und Optionen oft in Form von Menüs angeboten. Ein **Menü** wird nach dem Anklicken eines Ein-

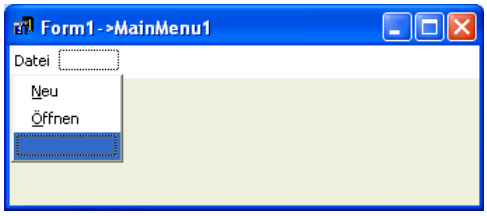
trags in der **Menüleiste** (unterhalb der Titelzeile des Programms, typische Einträge „Datei“, „Bearbeiten“ usw.) aufgeklappt und enthält **Menüeinträge** wie z.B. „Neu“, „Öffnen“ usw. Gut gestaltete Menüs sind übersichtlich gegliedert und ermöglichen dem Anwender, eine gewünschte Funktion schnell zu finden.

2.9.1 Hauptmenüs und der Menüdesigner

Die Komponente **MainMenu** (Tool-Palette Kategorie „Standard“) stellt ein Hauptmenü zur Verfügung, das unter der Titelzeile des Formulars angezeigt wird.

Ein *MainMenu* wird wie jede andere Komponente ausgewählt, d.h. zuerst in der Tool-Palette angeklickt und dann durch einen Klick auf das Formular gesetzt. Dabei ist die Position im Formular ohne Bedeutung: Zur Laufzeit wird das Menü immer unterhalb der Titelzeile des Formulars angezeigt.

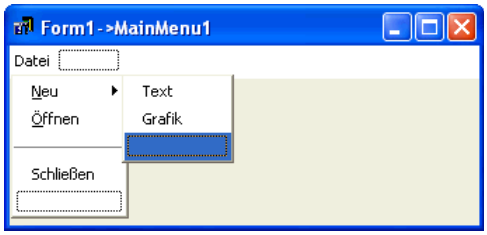
Durch einen Doppelklick auf das Menü im Formular wird dann der **Menüdesigner** aufgerufen, mit dem man das Menü gestalten kann. Dazu trägt man in die blauen Felder die Menüeinträge so ein, wie man sie im laufenden Programm haben möchte. Mit den Pfeiltasten oder der Maus kann man die Menüeinträge anwählen.



Während man diese Einträge macht, kann man im Objektinspektor sehen, dass jeder Menüeintrag den Datentyp T*MenuItem* hat. Der im Menü angezeigte Text ist der Wert der Eigenschaft *Caption*.

Die folgenden Optionen werden in vielen Menüs verwendet:

- Durch das Zeichen & („kaufmännisches Und“, *Umschalt-6*) vor einem Buchstaben der *Caption* wird dieser Buchstabe zu einer **Zugriffstaste**. Er wird dann im Menü unterstrichen angezeigt. Die entsprechende Option kann dann zur Laufzeit durch Drücken der *Alt*-Taste mit diesem Buchstaben aktiviert werden.
- Über die Eigenschaft ***ShortCut*** kann man im Objektinspektor Tastenkürzel definieren, die eine Menüoption auch ohne die *Alt*-Taste aktivieren.
- Die *Caption* „-“ wird im Menü als **Trennlinie** dargestellt.
- **Verschachtelte Untermenüs** erhält man über das Kontextmenü des Menüdesigners (mit der rechten Maustaste einen Menüeintrag anklicken) mit der Option *Untermenü erstellen* bzw. über *Strg+Rechtspfeil*.



Durch einen Doppelklick auf einen Menüeintrag im Menüdesigner (bzw. auf das Ereignis *OnClick* der Menüoption im Objektinspektor) erzeugt der C++Builder die Ereignisbehandlungsroutine für das Ereignis *OnClick* dieses Menüeintrags. Diese Funktion wird zur Laufzeit beim Anklicken dieses Eintrags aufgerufen:

```
void __fastcall TForm1::ffnen1Click(TObject *Sender)
{

}
```

Während der Name einer Ereignisbehandlungsroutine bei allen bisherigen Komponenten aus ihrem Namen abgeleitet wurde (z.B. *Button1Click*), wird er bei einer Menüoption aus dem Wert der Eigenschaft *Caption* erzeugt. Da die *Caption* kein zulässiger C++-Name sein muss, werden unzulässige Zeichen (Umlaute, Leerzeichen usw.) entfernt. Deshalb fehlt im Namen „ffnen1Click“ das „Ö“.

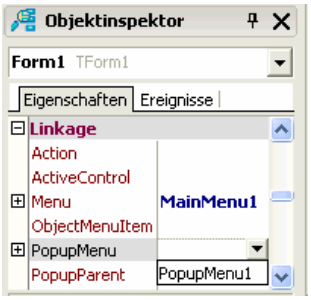
Zwischen die geschweiften Klammern schreibt man dann die Anweisungen, die beim Anklicken des Menüeintrags ausgeführt werden sollen. Das sind oft Aufrufe von Standarddialogen, die im nächsten Abschnitt vorgestellt werden.

2.9.2 Kontextmenüs

Ein Kontextmenü ist ein Menü, das einem Steuerelement zugeordnet ist und angezeigt wird, wenn man das Steuerelement mit der rechten Maustaste anklickt. Kontextmenüs werden auch als „lokale Menüs“ bezeichnet.

Kontextmenüs werden über die Komponente **PopupMenu** zur Verfügung stellt. Ein PopupMenu wird wie ein MainMenu auf ein Formular gesetzt und mit dem Menüdesigner gestaltet.

Die Zuordnung eines Kontextmenüs zu der Komponente, bei der es angezeigt werden soll, erfolgt über die Eigenschaft *PopupMenu* der Komponente. Jede Komponente, der ein Kontextmenü zugeordnet werden kann, hat diese Eigenschaft. Die Zuordnung kann im Objektinspektor erfolgen: Im Pulldown-Menü der Eigenschaft *PopupMenu* kann man alle bisher auf das Formular gesetzten Kontextmenüs auswählen. In der Abbildung rechts wird also dem Formular *Form1* das Kontextmenü *PopupMenu1* zugeordnet.



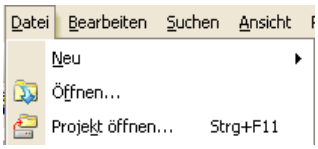
Die Eigenschaft *PopupMenu* kann nicht nur während der Entwurfszeit im Objektinspektor, sondern auch während der Laufzeit des Programms zugewiesen werden:

```
if (CheckBox1->Checked) Form1->PopupMenu = PopupMenu1;
else Form1->PopupMenu = PopupMenu2;
```

Kontextmenüs bieten oft dieselben Funktionen wie Hauptmenüs an. Dann muss die entsprechende Ereignisbehandlungsroutine kein zweites Mal geschrieben werden, sondern kann im Objektinspektor ausgewählt werden (siehe Abbildung rechts).

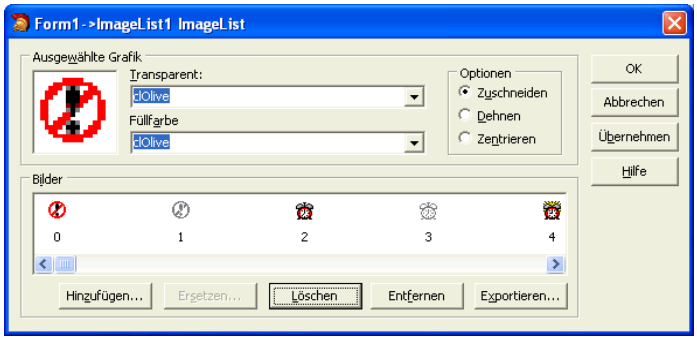
2.9.3 Die Verwaltung von Bildern mit ImageList Θ

Mit einer **ImageList** können Menüeinträgen Grafiken zugeordnet werden, die dann links von den Menüeinträgen angezeigt werden.



Eine **ImageList** (Tool-Palette Kategorie „Win32“) verwendet man zur Speicherung von Bildern, die von anderen Komponenten (MainMenu, PopupMenu, ToolBar, ListView, TreeView usw.) angezeigt werden. Sie ist zur Laufzeit nicht sichtbar.

Nachdem man eine ImageList auf ein Formular gesetzt hat, wird durch einen Doppelklick auf ihr Symbol der Editor für die Bilderliste aufgerufen:



Mit dem Button „Hinzufügen“ kann man Bilder in die ImageList laden. Zahlreiche unter Windows gebräuchliche Bilder findet man im Verzeichnis „C:\Programme\Gemeinsame Dateien\Borland Shared\Images\Buttons“.

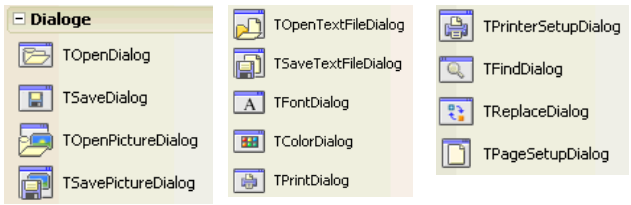
Die Zuordnung der ImageList zu einem Menü erfolgt dann über die Eigenschaft *Images* des Menüs (am einfachsten im Pulldown-Menü auswählen). Den einzelnen Menüeinträgen werden dann die Bilder aus der Bilderliste über die Eigenschaft *ImageIndex* (die Nummer des Bildes) zugeordnet. Diese können ebenfalls im Objektinspektor über ein Pulldown-Menü ausgewählt werden.

2.9.4 Menüvorlagen speichern und laden Θ

Im Kontextmenü des Menüdesigners kann man mit der Option „Als Template speichern“ ein Menü als Vorlage speichern und anderen Programmen zur Verfügung stellen. Der C++Builder verwendet dazu die Datei *bds.dmt* im *bin*-Verzeichnis.

2.10 Standarddialoge

Die Kategorie *Dialoge* der Tool-Palette enthält Komponenten für die **Standarddialoge** unter Windows: („common dialog boxes“).



Diese Dialoge werden von Windows zur Verfügung gestellt, damit häufig wiederkehrende Aufgaben wie die Eingabe eines Dateinamens in verschiedenen Anwendungen auf dieselbe Art erfolgen können.

Beispiel: Durch einen *OpenDialog* erhält man das üblicherweise zum Öffnen von Dateien verwendete Dialogfenster:

Ein OpenDialog wird im Unterschied zu vielen anderen Steuerelementen (z.B. Buttons) nicht automatisch nach dem Start des Programms angezeigt, sondern erst durch einen Aufruf seiner Methode *Execute*:

virtual bool ***Execute****();*

Diese Funktion gibt *false* zurück, wenn der Dialog abgebrochen wurde (z.B. mit dem „Abbrechen“ Button oder der ESC-Taste). Nach einer Bestätigung (z.B. mit dem Button „Öffnen“ oder der ENTER-Taste) ist der Funktionswert dagegen *true*. Dann stehen die Benutzereingaben als Werte von Eigenschaften zur Verfügung. Der ausgewählte Dateiname ist der Wert der Eigenschaft *FileName*:

__property AnsiString ***FileName****;*

Beispiel: Einen OpenDialog ruft man meist nach dem Anklicken der Menüoption *Datei|Öffnen* auf. Die Benutzereingaben werden nur nach einer Bestätigung des Dialogs verwendet:

```
void __fastcall TForm1::Oeffnen1Click(TObject
                                      *Sender)
{
if (OpenDialog1->Execute())
  { // der Dialog wurde bestätigt
    Memo1->Lines->
              LoadFromFile(OpenDialog1->FileName);
  }
}
```

Die Standarddialoge können vor ihrem Aufruf über zahlreiche Eigenschaften konfiguriert werden. Bei einem Open- und SaveDialog sind das unter anderem:

__property AnsiString ***Filter****;* // Maske für Dateinamen
__property AnsiString ***InitialDir****;* // Das beim Aufruf angezeigte Verzeichnis

Bei der Eigenschaft *Filter* gibt man keinen, einen oder mehrere Filter an. Jeder Filter besteht aus Text, der im Dialog nach „Dateityp“ angezeigt wird, einem senkrechten Strich „|“ (*Alt Gr* <) und einem oder mehreren Mustern für die Dateinamen (z.B. „*.txt“), die durch Semikolons „;“getrennt werden. Mehrere Filter können getrennt durch senkrechte Striche „|“ angegeben werden. Diese können dann im Pull-down-Menü nach „Dateityp“ ausgewählt werden.

Bei einem SaveDialog kann man mit der Eigenschaft

__property AnsiString ***DefaultExt***

eine Zeichenfolge (maximal drei Zeichen, ohne einen Punkt „.“) festlegen, die automatisch an den Dateinamen angefügt wird.

Beispiel: Den OpenDialog vom Anfang dieses Abschnitts erhält man mit den folgenden Zuweisungen vor dem Aufruf von *Execute*:

```
OpenDialog1->InitialDir = "c:\\CBuilder";
OpenDialog1->Filter = "C++ Dateien|*.CPP;*.H";
```

Die Standarddialoge der Kategorie „Dialoge“ der Tool-Palette:

OpenDialog	Zeigt Dateien aus einem Verzeichnis an und ermöglicht, eine auszuwählen oder einzugeben (Eigenschaft *FileName*), die geöffnet werden soll.
SaveDialog	Um den Namen auszuwählen oder einzugeben, unter dem eine Datei gespeichert werden soll. Viele Gemeinsamkeiten mit OpenDialog, z.B. die Eigenschaften *FileName*, *InitialDir*, *Filter* usw.
OpenPictureDialog *SavePictureDialog*	Wie ein Open- bzw. SaveDialog, mit einem Filter für die üblichen Grafikformate und einer Bildvorschau.
OpenTextFileDialog *SaveTextFileDialog*	Wie ein Open- bzw. SaveDialog, aber mit einer auswählbaren Textkodierung.
FontDialog	Zeigt die verfügbaren Schriftarten und ihre Attribute an und ermöglicht, eine auszuwählen. Will man die ausgewählte Schriftart (Eigenschaft *Font*) einem Steuerelement zuweisen, sollte man die Methode *Assign* verwenden. Beispiel: `Memo1->Font->Assign(FontDialog1->Font)`
ColorDialog	Zeigt die verfügbaren Farben an und ermöglicht, eine auszuwählen (Eigenschaft *Color*).
PrintDialog	Zur Auswahl eines Druckers, der Anzahl der Exemplare usw.
PrinterSetupDialog	Zur Einrichtung von Druckern.
FindDialog	Zum Suchen von Text.
ReplaceDialog	Zum Suchen und Ersetzen von Text.

Alle Standarddialoge werden wie ein OpenDialog durch den Aufruf der Methode

virtual bool ***Execute****();*

angezeigt. Der Funktionswert ist *false*, wenn der Dialog abgebrochen wurde, und andernfalls *true*. Im letzteren Fall findet man die Benutzereingaben aus dem Dialogfenster in entsprechenden Eigenschaften der Dialogkomponente.

2.10.1 Einfache Meldungen mit *ShowMessage*

Die Funktion

void ***ShowMessage****(AnsiString Msg);*

zeigt ein einfaches Fenster mit der als Argument übergebenen Meldung an (siehe auch Abschnitt 10.2.3).

Aufgabe 2.10

Schreiben Sie ein Programm mit einem Hauptmenü, das die unten aufgeführten Optionen enthält. Falls die geforderten Anweisungen bisher nicht vorgestellt wurden, informieren Sie sich in der Online-Hilfe darüber (z.B. *SelectAll*). Das Formular soll außerdem ein Memo enthalten, das den gesamten Client-Bereich des Formulars ausfüllt (Eigenschaft *Align=alClient*).

- *Datei|Öffnen*: Falls ein Dateiname ausgewählt wird, soll diese Datei mit der Methode *LoadFromFile* in das Memo eingelesen werden. In dem OpenDialog sollen nur die Dateien mit der Endung „.txt“ angezeigt werden.
- *Datei|Speichern*: Falls ein Dateiname ausgewählt wird, soll der Text aus dem Memo mit der Methode *SaveToFile* von *Memo1->Lines* unter diesem Namen gespeichert werden. Diese Option soll außerdem mit dem ShortCut „Strg+S“ verfügbar sein.
- Nach einem Trennstrich.
- *Datei|Schließen*: Beendet die Anwendung

- *Bearbeiten|Suchen*: Ein *FindDialog* ohne jede weitere Aktion.
- *Bearbeiten|Suchen und Ersetzen*: Ein *ReplaceDialog* ohne jede weitere Aktion.
- Nach einem Trennstrich:
- *Bearbeiten|Alles Markieren*: Ein Aufruf von *Memo1->SelectAll*.
- *Bearbeiten|Ausschneiden*: Ein Aufruf von *Memo1->CutToClipboard*.
- *Bearbeiten|Kopieren*: Ein Aufruf von *Memo1->CopyToClipboard*.

- *Drucken|Drucken*. Ein *PrintDialog* ohne weitere Aktion. Mit den bisher vorgestellten Sprachelementen ist es noch nicht möglich, den Inhalt des Memos auszudrucken.
- *Drucken|Drucker einrichten*: Ein *PrinterSetupDialog* ohne weitere Aktion.

Durch Drücken der rechten Maustaste im Memo soll ein Kontextmenü mit den Optionen *Farben* und *Schriftart* aufgerufen werden:

– Popup-Menü|*Farben*: Die ausgewählte Farbe (Eigenschaft *Color* des *Color-Dialogs*) soll der Eigenschaft *Brush->Color* des Memos zugewiesen werden.
– Popup-Menü|*Schriftart*: Die ausgewählte Schriftart (Eigenschaft *Font* des *FontDialogs*) soll der Eigenschaft *Font* des Memos zugewiesen werden.