Displaying and editing data in grids

This topic describes how to use the data-aware *TDBGrid* control to display and edit dataset records in a tabular grid format similar to a spreadsheet or the tabular view found in most desktop database applications. Your application can use the data grid to insert, delete, edit, or simply display data in a dataset.

The TDBGrid component is an indirect descendant of *TWinControl* and can be accessed by selecting the *DBGrid* control from the Data Controls tab of the Component palette. The *TDBGrid* component can be customized to change how a column appears and how the data in the column is displayed. *TDBGrid* components can be completely reconfigured at runtime to hide and show columns and change the order, color, and width of columns.

This topic also discusses the TDBCtrlGrid control that enables you to display other data-aware components as repeating units within a grid-like structure.

The most convenient way to move through data in a grid and to insert, delete,
and edit data is to use the database navigator with the data grid on a form.

Press the >> button to read through topics in sequence.

Other places to look:

Grid controls

Control grids

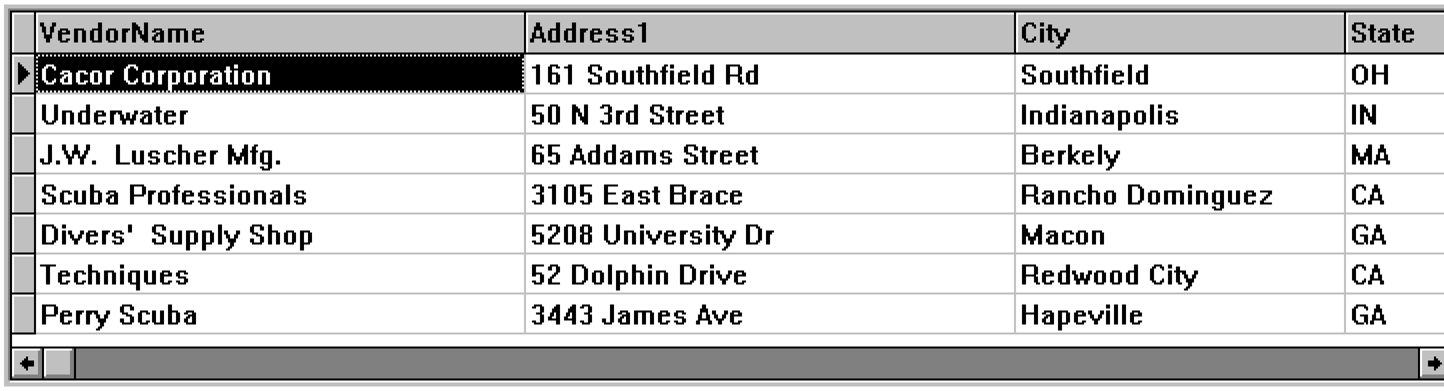
Features common to all data-aware controls

Setting up a data-aware control for displaying and editing data

Viewing and editing data in grids

You can use the *TDBGrid* control to view and edit records in a dataset in a tabular grid format.

The TDBGrid control



VendorName	Address1	City	State
Cacor Corporation	161 Southfield Rd	Southfield	OH
Underwater	50 N 3rd Street	Indianapolis	IN
J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
Divers' Supply Shop	5208 University Dr	Macon	GA
Techniques	52 Dolphin Drive	Redwood City	CA
Perry Scuba	3443 James Ave	Hapeville	GA

Two factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility in setting grid and data appearance.
- Creation of persistent field components for the dataset displayed in the grid.

The *TDBGridColumns* object holds a collection of *TColumn* objects representing all of the columns in a *TDBGrid* component. Every *TDBGrid* owns a *TDBGridColumns* object and exposes it via the *Columns* property. You can use the Columns editor to set up column attributes at design time or the *Columns* property to access *TDBGridColumns* methods and properties, which in turn expose *TColumn* objects representing individual columns.

By default, persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined either by persistent field components for the dataset displayed in the grid, or for datasets without persistent field components, by C++Builder's default set of criteria for data display.

By default, the appearance of records is determined primarily by the properties of the *fields* in the grid's dataset when no persistent column objects are defined. Temporary grid columns are dynamically generated from the fields of the dataset whose *Visible* property is *true*, and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Because a grid that does not have persistent column objects defined gets its display information from the fields of the dataset, all grids without persistent column objects defined that are linked to the same dataset will display data in the same way. To view different fields of the same dataset in two grids, use customized columns.

Using a grid control with no persistent column objects defined is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to display different datasets. A single grid can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's *DataSource* property changes or when the *DataSet* property of the data source itself is changed.

When no persistent column objects are defined, you can still modify a grid's column properties. Column properties are dynamic. They exist only as long as a column is associated with a particular field in a single dataset. Column properties based on the properties of underlying field components modify the underlying field property. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist for the lifetime of the column.

Properties of dynamic columns are retained as long as the associated field component exists. If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, the columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Press the >> button to read through the following topics in sequence.

Changing a grid's default display

Controlling grid behavior and appearance

Editing in the grid

Rearranging column order at runtime

Controlling grid drawing

Handling grid events

Changing a grid's default display

A customized grid control is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. You can use a customized grid to configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also permits users to modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (such as the grid or associated field) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property, which is usually the name of the field as it is stored in the database. To help the user understand the nature of the data in the column, it may be beneficial to change the name of the column to better reflect its contents. If you modify the *DisplayLabel* property, the column title reflects that change immediately. However, once you assign a value to a column property, it no longer changes when its default source changes. For example, if you assign a string to the column title's caption, the title caption is independent of the associated field's *DisplayLabel* property. Changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. You can use a blank column to display bitmaps or bar charts that graphically depict some aspect of a record's data in an otherwise blank cell, for example. To do so, you must override the cells' default drawing method

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

Note: Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is -1.

Persistent columns can be configured to display grid cells as a combo-box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (…) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

At runtime you can test a column's *AssignedValues* property to determine whether a column property gets its values from an associated field component, or has been assigned a separate value. Use a column's properties and methods to change column values. For example, to discard changes made to the title caption of column one and revert to its default value, you can use the statement

```
DBGrid1->Columns->Items[1]->Title->Caption = DBGrid1->Columns->Items[1]->Title->
DefaultCaption();
```

You can reset all default properties for a single column at runtime by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1->Columns->RestoreDefaults();
```

To add a persistent column at runtime, call the *Add* method for the column list:

```
DBGrid1->Columns->Add();
```

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1->Columns->Items[1]->Free();
```

Important:

Use the *Grid* property to see which *TDBGrid* component owns the object. Use the *Items* array property to access individual *TColumn* objects.

Press the >> button to read through topics in sequence.

Creating persistent columns

Removing persistent columns

Ordering persistent columns

Defining a lookup list column

Defining a pick list column

Putting an ellipsis button in a column

Setting column properties

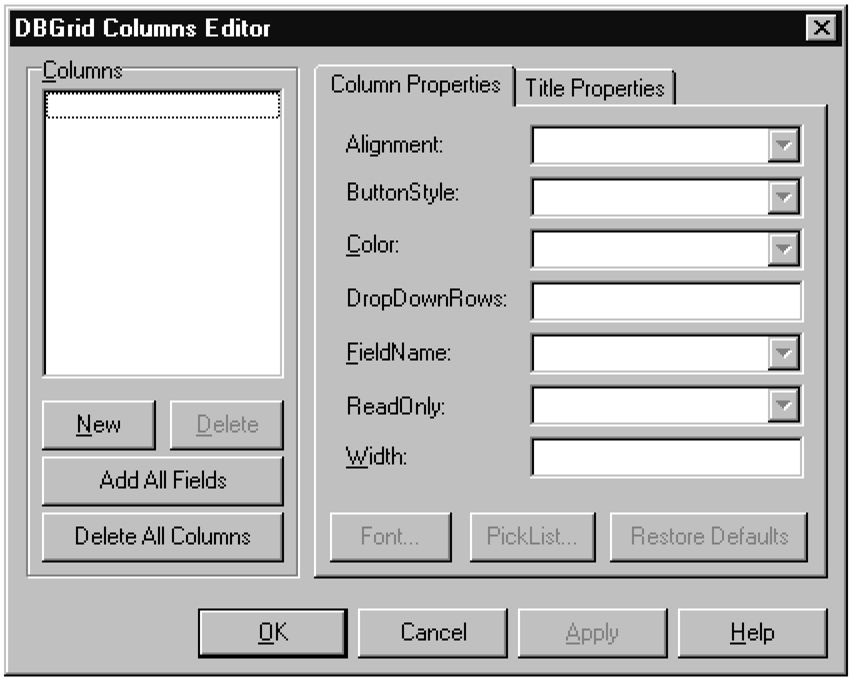
Setting column title properties

Restoring default column properties

Creating persistent columns

To customize the appearance of a grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. To create persistent columns for a grid control,

1. Select the grid component in the form.
2. Double-click on the grid component to invoke the Columns editor.



   The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

3. To create columns for all the fields in the grid's dataset, click the Add All Fields button. The Columns editor will display all persistent fields in the order they were entered and will automatically associate a field name with a column.
4. If the grid already contained persistent columns, a dialog box asks if you want to delete the existing columns. If you answer Yes, any existing persistent field information is removed and all fields in the current dataset are inserted by field name according to their order within the dataset. If you answer No, any existing persistent fields remain intact and all fields in the database are appended to the list of existing persistent fields.
5. To create persistent fields individually, click New in the Columns editor. The new column is given a sequential number and default title (for example, 0 - Untitled), is selected in the list box, and has no field name associated with it. Focus is in the *FieldName* box. If you want to associate a field with this new column, use the *FieldName* combo box to select a field from the grid's dataset. To change the new column's title, set the *Caption* property in the Title properties of the Columns editor.
6. Click OK to copy the dialog settings and close the dialog, or click Apply to copy the settings to the grid component without closing the dialog.

Removing persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

1. Select the field to remove in the Columns list box.
2. Click the Delete button.

Note: If you delete all the columns from a grid, the grid reverts to its default state and automatically builds dynamic columns for each field in the dataset.

Ordering persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

1. Select the column in the Columns list box.
2. Drag it to a new location in the list box.

You can also change the column order by dragging the column in the actual grid, just as you would at runtime.

Defining a lookup list column

To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field object in the dataset

Once the lookup field is defined, set the column's *FieldName* to the lookup field name and make sure the column's *ButtonStyle* is set to *cbsAuto*. The grid automatically displays a combo-box drop-down button when a cell of that column is in edit mode. The drop-down list is populated with lookup values defined by the lookup field.

Defining a pick list column

A pick list column displays a drop-down list of values that is populated with the list of values in the column's *PickList* property. The grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

To define a pick list column

1. Select the column in the Columns list box.
2. Set *ButtonStyle* to *cbsAuto*.
3. Click the Picklist button to bring up a string list editor.
4. In the String list editor, enter the list of values you want to appear in the drop-down list for this column. If the pick list contains data, the column becomes a pick list column.

Note: To force a column not to be a pick list column, delete all the text from the string list editor.

Putting an ellipsis button in a column

A column can display an ellipsis (…) button to the right of the normal cell editor. Mouse clicking or pressing Ctrl+Enter fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form which displays a graphic.

To add an ellipsis button to a column,

1. Select the column in the Columns list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

Setting column properties

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component, called the *default source*, such as the grid or an associated field component.

In the Columns editor, properties are divided into two pages: Column Properties and Title Properties. The following table summarizes the properties you can set on the Column Properties page.

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField.Alignment.*
ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data.*cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event.*cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. For text color, set the *Font* property. Default source: *TDBGrid.Color*.
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7 .
FieldName	Specifies the field name that is associated with this column. This can be blank.
ReadOnly	*true*: The data in the column cannot be edited by the user.*false*: (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default source: derived from *TField.DisplayWidth.*
Font	Specifies the font type, size, and color used to draw text in the column. Default source: *TDBGrid.Font*.
PickList	Contains a list of values to display in a drop-down list in the column.

Setting column title properties

The following table summarizes the properties you can set on the Title Properties page.

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default source: *TField.DisplayLabel*.
Color	Specifies the background color used to draw the column title cell. Default source: *TDBGrid.FixedColor*.
Font	Specifies the font type, size, and color used to draw text in the column title. Default source: *TDBGrid.TitleFont*.

Restoring default column properties

As you edit properties in the Columns editor, its text is displayed in boldface. This indicates that the property has been assigned and will not reflect changes to its default source.

The Restore Defaults button is enabled when the selected column has properties. Click the Restore Defaults button to discard all property changes in the selected column and revert to the column's defaults.

Controlling grid behavior and appearance

You can use the grid Options property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the Options property in the Object Inspector is displayed with a + (plus) sign to indicate that the Options property can be expanded to display a series of Boolean properties that you can set individually.

To view and set these properties, double-click the Options property. The list of options that you can set appears in the Object Inspector below the Options property. The + sign changes to a - (minus) sign, indicating that you can collapse the list of properties by double-clicking the Options property.

The following table lists the Options properties that can be set, and describes how they affect the grid at runtime:

Option	Purpose
dgEditing	true: (default). Enables editing, inserting, and deleting records in the grid.false: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	true: When a field is selected, it is in Edit state.false: (default). A field is not automatically in Edit state when selected. Press Enter to enter edit mode.
dgTitles	true: (default). Displays column titles across the top of the grid.false: Column title display is turned off.
dgIndicator	true: (default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam.false: The indicator column is turned off and columns cannot be resized or reordered.
dgColumnResize	true: (default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying *TField* component.false: Columns cannot be resized in the grid.
dgColLines	true: (default). Displays vertical dividing lines between columns.false: Does not display dividing lines between columns.
dgRowLines	true: (default). Displays horizontal dividing lines between records.false: Does not display dividing lines between records.
dgTabs	true: (default). Enables user to Tab and Shift+Tab between fields in records.false: Tabbing exits the grid control.
dgRowSelect	true: The selection bar spans the entire width of the grid.false: (default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	true: The selection bar in the grid is always visible, even if another control has focus.false: (default). The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	true: (default). Prompt for confirmation to delete records (Ctrl+Del).false: Delete records without confirmation.
dgCancelOnExit	true: (default). Cancels a pending insert if no modifications were made by the user when focus leaves the grid. This prevents inadvertent posting of partial or blank records.false: Allows a pending insert to be posted even if no modifications were made.
dgMultiSelect	true: Allows user to select multiple noncontiguous rows using Ctrl+Shift or Shift+arrow keys. The behavior is similar to a multiselect list box.false: (default). Does not allow user to select multiple noncontiguous rows.

Editing in the grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The CanModify property of the Dataset is *true*.
- The ReadOnly property of grid is *false*.

In most data-aware controls, the internal record buffer is updated when focus changes, which is not the same as posting the record back to the database file or server. In a grid, the behavior is different. *DBGrid* updates the internal record buffer as you move from field to field within a row. By default, edits and insertions within a field are posted to the database only when you move to a different row in the grid. Even if you use the mouse to change focus to another control on a form, the grid does not post changes until you move off the current row in the grid. When a record is posted, C++Builder checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, C++Builder raises an exception, and does not modify the record.

You can cancel all edits for a record by pressing Esc in any field before moving to another record.

To ensure that a value is entered in a field, set the field component's *Required* property to *true*.

Rearranging column order at design time

In grid controls with persistent columns, and default grids whose datasets contain persistent fields, you can reorder the grid columns at design time by clicking on the title cell of a column and dragging it to its new location in the grid.

Note: Reordering persistent fields in the Fields editor reorders columns in a default grid, but not a custom grid.

Important:

You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

Rearranging column order at runtime

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. The order of fields in the physical table is not affected.

A grid's *OnColumnMoved* event is fired after a column has been moved.

To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

Controlling grid drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The appearance of fields in the grid during display and editing is determined by the *DisplayFormat* or *EditFormat* properties of the field component, respectively, associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is *true*, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *false* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

Handling grid events

You can modify grid behavior by writing events handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector:

Event	Purpose
OnColEnter	Specify action to take when a user moves into a column on the grid.
OnColExit	Specify action to take when a user leaves a column on the grid.
OnColumnMoved	Called when the user moves a column to a new location.
OnDblClick	Specify action to take when a user double clicks in the grid.
OnDragDrop	Specify action to take when a user drags and drops in the grid.
OnDragOver	Specify action to take when a user drags over the grid.
OnDrawColumnCell	Called to draw individual cells.
OnEditButtonClick	Called when the user clicks on an ellipsis button in a column.
OnEndDrag	Specify action to take when a user stops dragging on the grid.
OnEnter	Specify action to take when the grid gets focus.
OnExit	Specify action to take when the grid loses focus.
OnKeyDown	Specify action to take when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Specify action to take when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Specify action to take when when a user releases a key when in the grid.
OnStartDrag	Specify the action to take when a user starts dragging on the grid.

There are many uses for these events. For example, you might write a handler for the OnDblClick event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the SelectedField property to determine the current row and column.

Displaying records like forms

TDBCtrlGrid displays multiple records from a data source. Unlike the TDBGrid component, which displays each record in a single row, you control the layout and appearance of each record in a *TDBCtrlGrid*. Each record is displayed in its own panel; you design one panel at design time and *TDBCtrlGrid* replicates that panel for each record displayed.

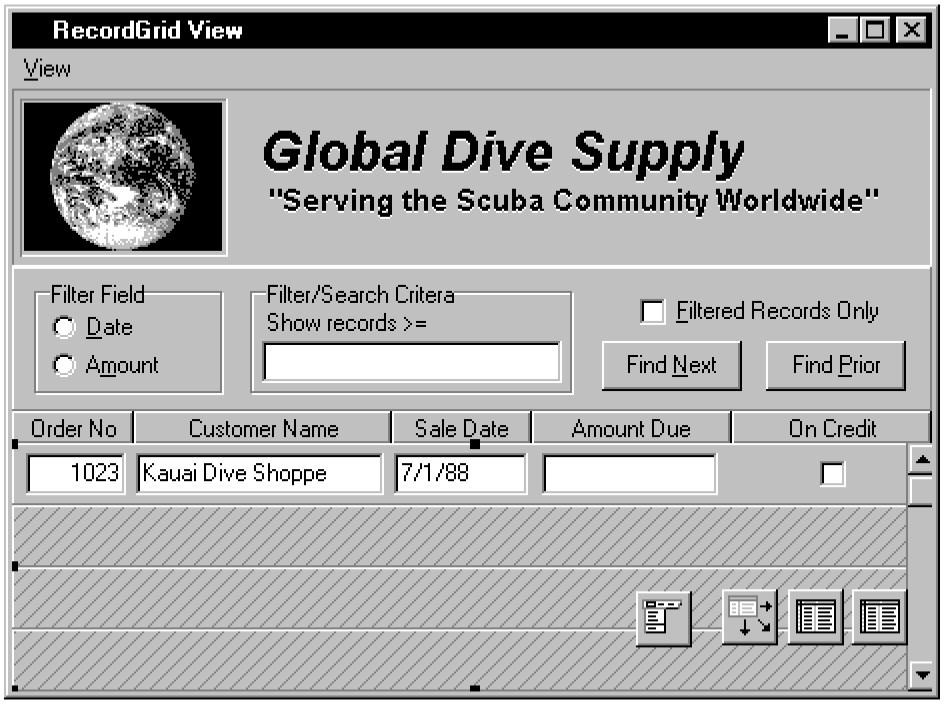
You cannot place a *DBGrid*, *DBNavigator*, *DBMemo*, *DBImage*, *DBListBox*, *DBRadioGroup*, *DBLookupListBox*, or another *DBCtrlGrid* within a *TDBCtrlGrid* control. *DBCtrlGrid* compatible controls must support replication and work best when the control displays only one field of one row of a dataset.

Like all data-aware controls, *TDBCtrlGrid*'s *DataSource* property tells it where to get the data it displays. To use a database control grid,

1. Place a database control grid on a form.
2. Set the grid's *DataSource* property to the name of a data source.
3. Place individual data-aware controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.
4. Set the *DataField* property for each data-aware control to the name of a field. The data source for these data-aware controls is already set to the data source of the database control grid.
5. Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data-aware controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.

TDBCtrlGrid at design time



The following table summarizes some of the unique properties for database control grids that you can set at design time:

Property	Purpose
AllowDelete	*true* (default): Permits record deletion.*false*: Prevents record deletion.
AllowInsert	*true* (default): Permits record insertion.*false*: Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.

Orientation	*goVertical* (default): Display records from top to bottom.*goHorizontal*: Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.
PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	*true* (default): Displays a focus rectangle around the current record's panel at runtime.*false*: Does not display a focus rectangle.

For more information about database control grid properties and methods, see the *VCL Reference*.

Handling batch move operations

This topic describes how to use the *TBatchMove* dataset component in your database applications. The *TBatchMove* component provides a high performance tool for manipulating sets of records. *TBatchMove* is fast because C++Builder simply passes the *Mode* and other parameters to the BDE and the entire operation is handled by the engine. This component enables you to copy a dataset to a table, append the records from a dataset to a table, delete records that match those in a dataset from a table, and update existing records with records from a dataset. It is most often used to

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

A batch move component inherits many of its fundamental properties and methods from TDataSet.

You can also duplicate a table's structure and data with a table component's *BatchMove method*.

You can copy a table on a one-shot basis outside of the application with the Data Migration wizard, or Data Pump

Press the >> button to read through each topic in sequence.

Using the batch move component

Specifying a batch move mode

Transliterating character data

Mapping column data types

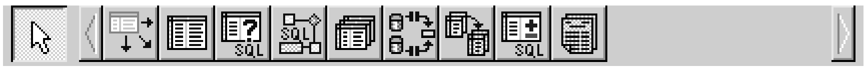
Executing a batch move

Handling batch move errors

Using the batch move component

To use a batch move component,

1. Place the table or query component for the dataset that you want to be the *Source* of the batch move operation and the table that you wish to be the *Destination* of the batch move operation on a form or in a data module.
2. Place a *TBatchMove* component from the Data Access tab of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.



3. Set the *Source* property of the component to the name of the table or query to copy, append, delete, or update from. If you placed a table or query component on your form or in a data module, you can select it from the drop-down list of available components.
4. Set the *Destination* property to the name of the table to create, append to, delete from, or update. If you are appending, updating, or deleting, *Destination* must be the name of an existing table. If you are copying and *Destination* is the name of an existing table, executing the batch move operation deletes the existing table. If you are creating a new table with a copy, the resulting table has the name specified in the *Name* property of the table component and is that server's database type (default data type for a local server is a Paradox data type). You can select tables from the drop-down list of available table components if you placed them on the form or in the data module.
5. Set the *Mode property* to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*.
6. Optionally set the *Transliterate property*. If *true* (the default), character data is transliterated from the dataset's character set to the destination's character set, if they are different.
7. Optionally set column mappings using the Mappings property. You need not set this property if you want batch move to match all columns based on their position in the source and destination tables.
8. Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch move operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists an original copy of all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used.
9. Execute the batch move to perform the specified operation.

Specifying a batch move mode

The Mode property specifies what the TBatchMove object will do. The following table describes the options for the *Mode* property:

Property	Purpose
batAppend	Append records from the source dataset to the destination table. C++Builder converts the data as necessary to adjust for differences in field type and size between the tables. If a field type conversion is not possible, C++Builder generates an exception and no data is appended. The destination table must already exist. This is the default mode.
batUpdate	Update records in the destination table with matching records from the source dataset. The destination table must exist and must have an index defined to match records. If the primary fields in the index match, the dataset replaces the existing record in the destination table. Records in the dataset that do not match existing records in the destination table are ignored. To allow records that do match to be added to the destination table, use the *batAppendUpdate* mode. If possible, C++Builder converts the data in the dataset to match the field types and sizes in the destination table.
batAppendUpdate	If the primary index fields in the source dataset match existing fields in the destination table, update it. Otherwise, it appends records to the destination table. The destination table must exist and must have an index defined to match records.
batCopy	Copy a dataset to a destination table. A *batCopy* operation copies only the tables's data and structure, and creates the destination table based on the structure of the source table. If the destination table already exists, the operation deletes it. If the dataset and destination table are on different platforms, say a server dataset and a local Paradox table, C++Builder creates the destination table with a structure as close to that of the source dataset as possible, and automatically performs any data conversion that is required. Other metadata that are part of the table, such as indexes, are not copied.
batDelete	Delete records in the destination table that match records in the source table. When you delete a dataset from a table, you remove each record from the destination table whose primary key matches the corresponding fields of a record in the dataset. The destination table must already exist and must have an index defined.

Transliterating character data

The Transliterate property specifies whether the data in the *Source* records should be converted from the locale of the *Source* to the locale of the *Destination* when the *Execute* method is called. *Transliterate* is *true* by default.

Set *Transliterate* to *true* when the *Source* dataset and the *Destination* table use different language drivers and the data may contain extended ASCII characters. Set *Transliterate* to *false* to avoid the overhead of the character set conversion when both the *Source* and the *Destination* use the same language driver or the data does not contain extended ASCII characters.

Mapping column data types

In a batch move operation, by default, columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on. In *batCopy* mode, a batch move component creates the destination table based on the column data types of the source table.

You may want to override the default column mappings if

- You are doing a copy operation and only want to copy a subset of columns to the destination table, you want to copy them in a different order, or you want to copy them to a different column name.
- You are doing a batch move update, append, or delete and the source dataset has different column names than the destination table.
- You are doing a batch move update, append, or delete and the source dataset and the destination table do not contain the same number and/or order of fields.

To override the default column mappings, use the Mappings property. You can enter the column data at design time using the String list editor. To invoke the String list editor for the *Mappings* property, double-click the *Mappings* property value column or click its ellipsis button.

You can create a list of column mappings (one per line) in one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a "best fit." It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column results in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of '5' to the corresponding integer value. Values that cannot be converted generate errors.

When moving data between different table types, a batch move component translates data types as appropriate. Fields in the destination table which have no entry in *Mappings* are set to null. The mappings from dBASE, Paradox, Oracle, Sybase, Informix, and InterBase data types are shown in the following tables.

Note: To batch move data to an SQL server database, you must have that database server and C++Builder Client/Server Suite with the appropriate SQL Link installed. For more information, see the SQL Links online Help.

From Paradox	To dBASE	To Oracle	To Sybase	To InterBase	To Informix
Alpha	Character	Character	VarChar	Varying	Character
Number	Float {20.4}	Number	Float	Double	Float
Money	Number {20.4}	Number	Money	Double	Money {16.2}
Date	Date	Date	DateTime	Date	Date
Short	Number {6.0}	Number	SmallInt	Short	SmallInt
Memo	Memo	Long	Text	Blob/1	Text
Binary	Memo	LongRaw	Image	Blob	Byte
Formatted memo	Memo	LongRaw	Image	Blob	Byte
OLE	OLE	LongRaw	Image	Blob	Byte
Graphic	Binary	LongRaw	Image	Blob	Byte

Long	Number {11.0}	Number	Int	Long	Integer
Time	Character {>8}	Character {>8}	Character {>8}	Character {>8}	Character {>8}
DateTime	Character {>8}	Date	DateTime	Date	DateTime
Bool	Bool	Character {1}	Bit	Character {1}	Character
AutoInc	Number{11.0}	Number	Int	Long	Integer
Bytes	Memo	LongRaw	Image	Varying	Byte
BCD	N/A	N/A	N/A	N/A	N/A

From dBASE	To Paradox	To Oracle	To Sybase	To InterBase	To Informix
Character	Alpha	Character	VarChar	Varying	Character
Number	Short	Number	SmallInt	Short	SmallInt
others	Number	Number	Float	Double	Float
Float	Number	Number	Float	Double	Float
Date	Date	Date	DateTime	Date	Date
Memo	Memo	Long	Text	Blob/1	Text
Bool	Bool	Character {1}	Bit	Character {1}	Character
Lock	Alpha {24}	Character {24}	Character {24}	Character {24}	Character
OLE	OLE	LongRaw	Image	Blob	Byte
Binary	Binary	LongRaw	Image	Blob	Byte
Bytes	Bytes	LongRaw	Image	Blob	Byte

From Oracle	To Paradox	To dBASE	To Sybase	To InterBase	To Informix
Character	Alpha	Character	VarChar	Varying	Character
Raw	Number	Float {20.4}	Float	Double	Float
Date	DateTime	Date	DateTime	Date	DateTime
Number	Number	Float {20.4}	Float	Double	Float
Long	Memo	Memo	Text	Blob/1	Text
LongRaw	Binary	Memo	Image	Varying	Byte

From Sybase	To Paradox	To dBASE	To Oracle	To InterBase	To Informix
Character	Alpha	Character	Character	Varying	Character
Var Character	Alpha	Character	Character	Varying	Character
Int	Number	Number {11.0}	Number	Long	Integer
Small Int	Short	Number {6.0}	Number	Short	SmallInt
Tiny Int	Short	Number {6.0}	Number	Short	SmallInt
Float	Number	Float {20.4}	Number	Double	Float
Money	Money	Number {20.4}	Number	Double	Money {16.2}
Text	Memo	Memo	Long	Blob/1	Text
Binary	Binary	Memo	Raw	Varying	VarChar
Var Binary	Binary	Memo	Raw	Varying	VarChar

Image	Binary	Memo	LongRaw	Blob	Byte
Bit	Alpha	Bool	Character	Varying	Character
DateTime	DateTime	Date	Date	Date	DateTime
TimeStamp	Binary	Memo	Raw	Varying	VarChar
Float4	Number	Number	Number	Double	Float
Money4	Money	Number {20.4}	Number	Double	Money {16.2}
DateTime4	DateTime	Date	Date	Date	DateTime

From InterBase	**To Paradox**	**To dBASE**	**To Oracle**	**To Sybase**	**To Informix**
Short	Short	Number {6.0}	Number	Small Int	SmallInt
Long	Number	Number {11.0}	Number	Int	Integer
Float	Number	Float {20.4}	Number	Float	Float
Double	Number	Float {20.4}	Number	Float	Float
Char	Alpha	Character	Character	VarChar	Character
Varying	Alpha	Character	Character	VarChar	Character
Date	DateTime	Date	Date	DateTime	DateTime
Blob	Binary	Memo	LongRaw	Image	Byte
Blob/1	Memo	Memo	Long	Text	Text

From Informix	**To Paradox**	**To dBASE**	**To Oracle**	**To Sybase**	**To InterBase**
Char	Alpha	Character	Character	VarChar	Varying
Smallint	Short	Number {6.0}	Number	Small Int	Short
Integer	Number	Number {11.0}	Number	Int	Long
Smallfloat	Number	Float {20.4}	Number	Float	Double
Float	Number	Float {20.4}	Number	Float	Double
Money	Money	Number {20.4}	Number	Float	Double
Decimal	Number	Float	Number	Float	Double
Date	Date	Date	Date	DateTime	Date
Datetime	DateTime	Date	Date	DateTime	Date
Interval	Alpha	Character	Character	VarChar	Varying
Serial	Number	Number {11.0}	Number	Int	Long
Byte	Binary	Memo	LongRaw	Image	Blob
Text	Memo	Memo	Long	Text	Blob/1
VarChar	Alpha	Character	Character	VarChar	Varying

Executing a batch move

The Execute method performs the batch operation specified by Mode from the Source dataset to the *Destination* table at runtime. The *Active* property of the *Destination* table must first be set to *false*. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
   BatchMoveAdd->Execute();
```

You can also execute a batch move operation at design time by right clicking on a batch move component and choosing Execute from the context menu.

The MovedCount property keeps track of the number of records that are moved when a batch move executes.

The *RecordCount* property is used to control the maximum number of records that will be moved. If zero, all records are moved, beginning with the first record in *Source*. If *RecordCount* is not zero, a maximum of *RecordCount* records will be moved, beginning with the current record. If *RecordCount* exceeds the number of records remaining in *Source*, no wraparound occurs; the operation is terminated.

Handling batch move errors

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. TBatchMove has a number of properties that specify how it handles errors.

The AbortOnProblem property specifies whether to abort an operation when a data type conversion error occurs, such as when data has to be trimmed to fit into the destination field. *ProblemCount* is the number of records which could not be added to *Destination* without loss of data due to field width constraints. If *AbortOnProblem* is *true*, this number is one, since the operation is aborted when the problem occurs.

The AbortOnKeyViol property indicates whether to abort the operation when a Paradox key violation occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- ChangedTableName, if specified, creates a local Paradox table containing an original copy of all records in the destination table that changed as a result of the update or delete batch operations. The *ChangedCount* property records the number of records changed, whether or not *ChangedTableName* is specified.
- KeyViolTableName, if specified, creates a local Paradox table containing all records from the source table that caused a referential integrity or key violation error when writing to the destination table. If *AbortOnKeyViol* is *true*, this table will contain at most one record since the operation will be aborted with that first record.
- ProblemTableName, if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *true*, there is at most one record in the table since the operation is aborted on that first problem record. If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table

Caching updates

This topic describes cached updates, and the situations in which they are useful. Cached updates enable you to write changes to a dataset to a temporary buffer on the client instead of writing the changes directly to the server. Cached updates are similar to transactions except that they will work with both local and server data.

Also described in this chapter is the *TUpdateSQL component* that can be used in conjunction with cached updates to update virtually any dataset, particularly datasets that are not normally updateable.

The cached updates feature in C++Builder sets up calls to the Borland Database Engine (BDE), which actually performs the work. You can learn more about how this is happening by viewing the source code that ships with C++Builder or viewing the BDE Configuration Help file.

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached changes to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Cached updates are primarily intended to reduce data access contention on remote database servers by

- Minimizing transaction times.
- Minimizing network traffic.

When cached updates are enabled for a dataset, a read-only transaction retrieves as much data as necessary for display purpose and ends. Each new fetch starts a new transaction. To fetch all records in a dataset, use the *FetchAll* method. While an application user edits data in the cache, the application does not keep a transaction open. When the application applies the changes to the database, a second transaction writes cached data to the database, and then terminates. Network traffic is reduced because rather than initializing and sending network packets over the wire each time a change to a single record is written to the database, cached updates write all changes once, when requested.

Press the >> button to read through each topic in sequence.

How cached updates differ from transactions

In the BDE, when a transaction is active, updates are immediately sent to the underlying tables. Thus, errors (such as integrity constraint violations, and so on) are instantly reported to the clients. Because updates are immediately sent to the underlying tables, the updates are visible to other transactions. And because each modified record is locked, other users cannot interfere.

This behavior differs from that of cached updates, where updates are not sent to the underlying table until the commit time. Hence no errors are reported until the commit time. No record locks are held until the user decides to commit the updates. The locks are held only during the commit process. If errors occur during the commit process, clients are given an option to abort the commit process. If clients abort a commit process, the original state of the table is restored.

The main advantage of cached update is that the locks are held only during the commit time, thereby increasing the access time of SQL servers for other system transactions. Transactions lock out other users after record is changed, and local transactions limit the user to changing only the maximum number of records that can be locked. Cached updates avoid these problems, but permit another user to change data underneath you.

Deciding to use cached updates

While cached updates can minimize transaction times and drastically reduce network traffic, they may not be appropriate to for all C++Builder database client applications that work with remote servers. Before using cached updates, consider that cached data is local to your application. In a busy client/server environment this has three implications that your application must be able to respond to:

- Other applications can access and change the actual data on the server while your users edit their local copies of the data.
- Other applications cannot see any data changes made by your application until it applies all its changes.
- Validity checks applied by the server will not be applied until the client applies its updates.

C++Builder provides cached update methods and transaction control methods you can use in your application code to handle these situations, but you must make sure that you cover all possible scenarios your application is likely to encounter in your working environment.

Cached updates: an overview of the process

This section is essential to understanding the rest of the chapter. It provides an overview of the various options that are available when cached updates are enabled.

1. Create a data module and add the datasets and data sources that you will be using in your application. To view data, place a data-aware control on your form
2. Enable cached updates. Cached updates are enabled and disabled through the *CachedUpdates* properties of a dataset (*TTable*, *TQuery*, and *TStoredProc*). To use cached updates, set *CachedUpdates* to *true*, either at design time (through the Object Inspector), or at runtime.
3. Access the old, or original, value of each field in a record, if necessary. C++Builder keeps track of the new, or current, value of each field in a record as well as the old, or original, value of each field in a record.
4. Cancel changes made since cached updates were enabled, if necessary. If you don't want to apply the updates, you can cancel changes made since cached updates were enabled for all pending updates or for individual records.
5. Apply the cached updates and write the changes to the database. You can also use a *TUpdateSQL* component to use prepared SQL statement to update a dataset.

If the dataset has an *OnUpdateRecord* event handler, it will be called once for each record that has updates in the cache.

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change the record in a database before your application applies its updates. If a record in the cache can't be updated, the dataset's OnUpdateError event will be triggered

Enabling and disabling cached updates

Cached updates are enabled and disabled through the *CachedUpdates* property of a dataset (*TTable*, *TQuery*, and *TStoredProc*). *CachedUpdates* is *false* by default, meaning that cached updates are not enabled for a dataset.

To use cached updates, set *CachedUpdates* to *true*, either at design time (through the Object Inspector), or at runtime. For example, the following code enables cached updates for a dataset

```
CustomersTable->CachedUpdates = true;
```

To disable cached updates for a dataset, set *CachedUpdates* to *false*.

Caution: If you disable cached updates before you apply any pending changes, those changes are discarded.

For example, the following code prompts for confirmation before disabling cached updates for a dataset:

```
if (MessageBox(NULL, "Discard pending updates?", "Confirmation", MB_OK) == MB_OK)
 CustomersTable->CachedUpdates = false;
```

Applying cached updates

When a dataset is in cached update mode, changes to data are not actually written to the database until your application explicitly calls methods that apply those changes. Normally an application applies updates in response to user input, such as through a button or menu item.

ApplyUpdates applies all pending cached updates. If you are applying updates to a dataset, call *CommitUpdates* to reflect that cached updates were written. Generally, you should call the database object's *ApplyUpdates* method instead of the dataset's. It calls *ApplyUpdates* and *CommitUpdates* within a transaction.

If the dataset has an *OnUpdateRecord* event handler, it will be called once for each record that has updates in the cache. If a record in the cache cant be updated, the dataset's *OnUpdateError* event will be triggered. If the dataset's *UpdateObject* property points to a *TUpdateSQL* component and doesn't have an *OnUpdateRecord* event handler, its *Apply* method will be called automatically for each record that has updates in the cache.

Applying updates is a two-phase process that must take place under the auspices of a database component's transaction control to enable your application to recover gracefully from errors.

When applying updates under transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1).

   If the database write is unsuccessful,

- Database changes are rolled back, ending the database transaction.
- Cached updates are not committed, leaving them intact in the internal cache buffer.

Note: If cached updates are applied to a server database that controls the transaction and all updates are applied within the context of a single transaction, then any error on a single record means that all updates are voided.

If the database write is successful,

- Database changes are committed, ending the database transaction.
- Cached updates are committed, clearing the internal cache buffer (phase 2).

The two-phased approach to applying cached updates allows for effective error recovery, especially when updating multiple datasets (for example, the datasets associated with a master/detail form).

Updating with a database component

To apply cached updates in the context of a database connection, call the database component's *ApplyUpdates* method. *ApplyUpdates* applies and commits all pending cached updates for the datasets specified in the *DataSets* open array parameter (by calling the datasets' *ApplyUpdates* and *CommitUpdates* methods). It uses a transaction (if one is not already started) to ensure that all updated records in all affected datasets (such as in a master/detail relationship) are updated in their entirety. The following code applies updates for the *CustomersQuery* dataset in response to a button click event:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TDBDataSet* ds[] = {Table1};            // need valid dataset in the array

Table1>Database>TransIsolation = tiDirtyRead;     // needed for Pdox and dBASE tables
try  {
  Table1>Database>ApplyUpdates (ds, 0);       // use (size 1)
}
catch(...) {
  MessageBox(0, "Exception caught", "INFO", MB_OK);
}

}
```

This method starts a transaction, and writes cached updates to the database. If successful, it also commits the transaction, and then commits the cached updates. If unsuccessful, this method rolls back the transaction, and does not change the status of the cached updates. In this latter case, your application should handle cached update errors through a dataset's OnUpdateError event.

The single argument to the *ApplyUpdates* method for a database is an array of dataset names. To apply updates for more than one dataset, separate dataset names with commas. For example, the following statement applies updates for two tables used in a master/detail form:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TDBDataSet* ds[] = {Table1, Table2};

Table1>Database>TransIsolation = tiDirtyRead;
try  {
  Table1>Database>ApplyUpdates (ds, 1);
}
catch(...) {
  MessageBox(0, "Exception caught", "INFO", MB_OK);
}

}
```

Updating with a dataset component

To apply cached updates in the context of a dataset, you can also explicitly code both phases for applying cached updates. Generally, you should call the database object's *ApplyUpdates* method instead of the dataset's. It calls *ApplyUpdates* and *CommitUpdates* within a transaction. The *CommitUpdates* method updates the cache to reflect that pending cached updates were successfully applied by a call to the *ApplyUpdates* method.

To apply cached updates in the context of a dataset, your application must call two dataset methods:

1. *ApplyUpdates*, to write cached changes to a database (phase 1)
2. *CommitUpdates*, to clear the internal cache when the database write is successful (phase 2)

If you call a dataset's cached update methods directly, you can still apply the updates under the auspices of a database transaction. For example, the following code illustrates how you must apply updates for the *CustomerQuery* dataset previously used to illustrate updates through a database method:

```
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
 Database1->StartTransaction();
 try
   {
    CustomerQuery->ApplyUpdates(); //try to write the updates to the database
    Database1->Commit(); //on success, commit the changes
   }
 catch(...)
   {
    Database1->Rollback(); //on failure, undo any changes
    throw ("error"); //raise the exception again to prevent a call to CommitUpdates
   }
 CustomerQuery->CommitUpdates(); //on success, clear the internal cache
}
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The throw statement inside the catch(...) block re-raises the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

The following example illustrates how you should apply cached updates code to two tables involved in a master/detail relationship:

```
Database1->StartTransaction();
try
 {
 Master->ApplyUpdates();
 Detail->ApplyUpdates();
 Database1->Commit();
 }
catch(...)
 {
 Database1->Rollback();
 throw ("error");
 }
Master->CommitUpdates();
Detail->CommitUpdates();
```

If an error occurs during the application of updates, this code also leaves both the cache and the underlying data in the database tables in the same state they were in before the calls to *ApplyUpdates*.

If an exception is raised during the call to *Master->ApplyUpdates*, it is handled like the single dataset case previously described. Suppose, however, that the call to *Master->ApplyUpdates* succeeds, and the subsequent call to *Detail->ApplyUpdates* fails. In this case, the changes are already applied to the master table. Because all data is updated inside a database transaction, however, even the changes to the master table are rolled back when *Database1->Rollback* is called in the catch(...) block. Furthermore, *Master->CommitUpdates* is not called because the exception which is re-raised causes that code to be skipped, so the cache is also left in the state it was before the attempt to update.

To appreciate the value of the two-phase update process, assume for a moment that *ApplyUpdates* is a single-phase process which updates the data and the cache. If this were the case, and if there were an error while applying the updates to the *Detail* table, then there would be no way to restore both the data and the cache to their original states. Even though the call to *Database->Rollback* would restore the database, there would be no way to restore the cache.

Canceling cached updates

There are three methods that cancel changes made while cached updates are enabled.

Canceling all updates

CancelUpdates discards all pending updates, clears the cache, and restores the dataset to the state it was in when the table was opened, cached updates were enabled, or updates were last successfully applied. *CancelUpdates* is always successful so no errors will occur. The dataset returns to the state it was at before cached updates were enabled. The following statement cancels updates for the *CustomersTable*:

```
CustomersTable->CancelUpdates();
```

Note: When you close a dataset or disable cached updates, *CancelUpdates* is called automatically.

Cancelling updates to the current record

RevertRecord discards any changes you posted to the current record when cached updates are enabled and restores the current record in the dataset to an unmodified state. Use this method to discard changes from a record before you call *ApplyUpdates*. Use the *UpdateRecordTypes* property to be able to revert the deletion of a record. If the record is not modified, this call has no effect. For example,

```
CustomersTable->RevertRecord();
```

Cancelling the most recently posted cached update

CancelCurrentUpdate discards the most recently posted cached update. It returns *true* if successful, and *false* if not. For example,

```
   CustomersTable->CancelCurrentUpdate();
```

Undeleting a record

To undelete a record requires some coding because once a record is deleted, it is no longer the current record. The process involves using the *UpdateRecordTypes* property to make the deleted records "visible," and then calling *RevertRecord*. Here is a code example which will undelete all deleted records in the customer table:

```
void __fastcall UndeleteAll(TDataSet DataSet)
{
 DataSet->UpdateRecordTypes=DataSet->UpdateRecordTypes<<rtDeleted;//recognize only deleted
//records
   try{
     DataSet->First();  //Go to the first previously deleted record
     while (!DataSet->Eof)
       DataSet->RevertRecord();  //Undelete until we reach the last record
   catch(...)
     {DataSet->UpdateRecordTypes=DataSet->UpdateRecordTypes << rtDeleted;}

//Restore updates types to recognize only modified, inserted, and unchanged records
   DataSet->UpdateRecordTypes=DataSet->UpdateRecordTypes<<rtModified,rtInserted,
rtUnmodified;
}
```

Controlling which types of records are included in the dataset

The *UpdateRecordTypes* property controls what type of records are included in the dataset (it works in much the same way as ranges and filters work) when cached updates are enabled. The *UpdateRecordTypes* property is primarily useful for accessing deleted records so they can be undeleted through a call to *RevertRecord*. This property would also be useful if you wanted to provide a way in your application for users to view only new (*rtInserted*) records, for instance. *UpdateRecordTypes* is a set, so it can contain any combination of the following values:

Value	Meaning
rtModified	Modified records
rtInserted	Inserted records
rtDeleted	Deleted records
rtUnmodified	Unmodified records.

The default value for *UpdateRecordTypes* is [*rtModified, rtInserted, rtUnmodified*].

Checking update status

You can keep track of the status of each record when cached updates are enabled. You can also query each record to see if it's been updated, and if so, you have access to the current value of the fields in the updated record as well as to the previous, or old, value of the fields

When cached updates are enabled for your application, you may want to be able to indicate to the user which records in a dataset have been modified, but not yet applied to the dataset. The *UpdateStatus* property provides this capability. It returns one of the following values for the current record:

Value	Meaning
usUnmodified	Record is unchanged.
usModified	Record is changed.
usInserted	Record is a new record.
usDeleted	Record is deleted.

When a dataset is first opened, all records will have an update status of *usUnmodified*. As records are inserted, deleted, and so on, the status values will reflect the change. Here is an example of *UpdateStatus* property used in an *OnCalcFields* event handler to display an asterisk in a calculated field if its associated record is modified:

```
void __fastcall TForm1::CalcFields(TDataSet DataSet)
{
 TStringField *Table1ModifiedRow = new TStringField(this);
 if (DataSet->UpdateStatus() != usUnmodified)
   Table1ModifiedRow->Value = "*";
 else
   Table1ModifiedRow->Value = NULL;
}
```

Note: If a record's *UpdateStatus* is *usModified*, you can examine the *OldValue* property for each field in the dataset to determine its previous value. *OldValue* is meaningless for records with *UpdateStatus* values other than *usModified*.

Handling cached update errors

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change the record in a database before your application applies its updates. Fortunately, the Borland Database Engine (BDE) specifically checks for this situation when attempting to apply updates, and reports it as an error.

A dataset component's *OnUpdateError* event enables you to catch and respond to this error (and any other update error as well). You should code this event handler if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Caution: Do not call any dataset methods that change the current record (such as *Next* and *Prior*) in an *OnUpdateError* event handler. Doing so will cause your event handler to get stuck in an endless loop.

Here is the skeleton code for an OnUpdateError event handler.

```
void __fastcall TForm1::Query1UpdateError(TDataSet *DataSet, EDatabaseError *E, TUpdateKind
UpdateKind, TUpdateAction &UpdateAction)
{
 // Perform updates here...
}
```

The *TUpdateError* event type defines the type of method that handles errors during the application of cached updates. The *OnUpdateError* event occurs when cached updates are enabled, *CachedUpdates* is *true*, applied (*ApplyUpdates*), and there are errors applying the updates. Such errors include integrity violations and cached records being modified or deleted by another user. *OnUpdateError* event handlers are supplied with the dataset that was involved, an *EDatabaseError* exception object to get more details about the error, the status code indicating the problem, and a reference variable parameter that lets you specify how the error should be dealt with. If you don't change *UpdateAction*, the processing of cached updates will fail with an exception.

The following sections describe each of the parameters passed to the update error handler, and how those parameters are used.

Referencing the dataset to which updates are applied

DataSet is a parameter of type *TDataSet* that references the dataset to which updates are applied. To process new and old record values during error handling you must supply this reference.

Extracting an error message

The *E* parameter is usually of type *EDBEngineError*. From this exception type you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel->Caption = E->Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it. For example, the following code checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
Word MyErrorCode;
EDBEngineError *E = new EDBEngineError(MyErrorCode);
PChar MyMessage;
TDBError *MyError = new TDBError(E, MyErrorCode, DBIERR_KEYVIOL, MyMessage);
// Add BDE to your #include statement for this example
if (E)
 {
 if (E->Errors[E->ErrorCount-1] == MyError)
   UpdateAction = uaSkip //Key violation, just skip this record
 else
   UpdateAction = uaAbort //Don't know what's wrong, abort the update
 }
```

Specifying how the current record was changed

UpdateKind is a parameter of type *TUpdateKind*. It describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting a new record caused an error.
ukDelete	Deleting an existing record caused an error.

Error handling

UpdateAction is a parameter of type *TUpdateAction*. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler. *UpdateAction* can be set to one of the following values:

Value	Meaning
uaAbort	Aborts the update operation without displaying an error message.
uaFail	Aborts the update operation, and displays an error message. This is the default value for *UpdateAction* when you enter an update error handler.
uaSkip	Skips updating the row, but leaves the update for the record in the cache
uaRetry	Repeats the update operation. Correct the error condition before setting *UpdateAction* to this value.
uaApplied	Not used in error handling routines.

If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.

When set to *uaSkip*, the update for the row that caused the error is skipped, and it remains in the cache after all other updates are completed.

uaFail and *uaAbort* are alike. Both cause the entire update operation to end. *uaFail* raises an exception, and displays an error message. *uaAbort* raises a silent exception (does not display an error message). None of the changes will have taken place and you will return to edit mode as if you had never attempted to commit the data.

Note: When an error occurs during application of updates, unless *ApplyUpdates* methods are called from within try...catch statement, an error message is displayed after the call to *ApplyUpdates* is made. If you also display an error message to the user from inside your error event handler, your application may display the same error message twice. To prevent error message duplication set *UpdateAction* to *uaAbort* to turn off error message display.

The *uaApplied* value should only be used inside an OnUpdateRecord event. Do not set this value in an update error handler.

Accessing the old and new values for a field

When cached updates are enabled for records, the original values for fields in each record before any pending cached updates changed it are stored in a read-only *TField* property called *OldValue*. Once cached updates are applied successfully, it's not possible to retrieve the old field value. Cached updates must be enabled to access the *OldValue* property.

Changed values are stored in the analogous *TField* property *NewValue*. The *NewValue* property represents the current value of the field including any changes made in pending cached updates. If the current field value is causing a problem in an *OnUpdateEvent* handler applying the cached update (such as in a key violation), you can change *NewValue* to correct the problem. Cached updates must be enabled to access the *NewValue* property.

OldValue and *NewValue* provide the only way to inspect and change update values in *OnUpdateError* and *OnUpdateRecord* event handlers.

In some cases, you may be able to use these properties to determine the cause of an error and correct it. For example, the following code handles corrections to a salary field which can only be increased by 25 percent at a time (enforced by a constraint on the server):

```
TIntegerField *EmpTabSalary = new TIntegerField(this);
int SalaryDif, OldSalary;
OldSalary = EmpTabSalary->OldValue;
SalaryDif = EmpTabSalary->NewValue - OldSalary;
 if (SalaryDif / OldSalary > 0.25)
 { // Increase was too large, drop it back to 25%
   EmpTabSalary->NewValue = OldSalary * 1.25;
   UpdateAction = uaRetry;
 }
 else
   UpdateAction = uaSkip;
```

NewValue is decreased to 25 percent in the case where the salary increased by a larger percentage, and then the update operation is retried. To improve the efficiency of this routine, the *OldValue* parameter is stored in a local variable.

Using prepared SQL statements to update a dataset

TUpdateSQL is an update component that uses prepared SQL statements to update a dataset. An update component actually encapsulates three *TQuery* components. Each query component performs a single update task. One query component provides an SQL UPDATE statement for modifying records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

You associate a *TUpdateSQL* component with a dataset by setting the dataset's *UpdateObject* property. The dataset automatically uses the *TUpdateSQL* component when cached updates are applied.

When you place an update component in a data module or on a form, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties you supply at design time:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.

The *ModifySQL* property is an SQL statement that is executed when the cached update is a modification of an existing record. The *InsertSQL* property is an SQL statement that is executed when the cached update is the insertion of a new record. *DeleteSQL* is an SQL statement that is executed when the cached update is the deletion of a record. All three properties support an extension to normal parameter binding for cached updates: You can prefix any field name with 'OLD_' to retrieve the value of the field as it was before cached updates were enabled. This extension lets you access the old field values, generally for use in creating the WHERE clause of the SQL statement.

The *ModifySQL* property returns the string list containing the SQL statement to be used when applying the cached modification of a record. Generally, such an SQL statement would be an UPDATE statement. You can use the UpdateSQL editor (by double-clicking the *TUpdateSQL* component) to generate an appropriate SQL statement. *UpdateMode* specifies which columns C++Builder uses to find the record. In SQL terms, *UpdateMode* specifies which columns are included in the WHERE clause of an UPDATE statement.

Note: You cannot use a *TUpdateSQL* component when updating BLOB fields. Use the *ApplyUpdates* method and let the BDE do the updating if you need to update a BLOB field when using cached updates.

The *InsertSQL* property returns the string list containing the SQL statement to be used when applying the cached insertion of a record. Generally, such an SQL statement would be an INSERT INTO statement. You can use the UpdateSQL editor (by double-clicking the *TUpdateSQL* component) to generate an appropriate SQL statement.

The *DeleteSQL* property returns the string list containing the SQL statement to be used when applying the cached deletion of a record. Generally, such an SQL statement would be an DELETE FROM statement. You can use the UpdateSQL editor (by double-clicking the *TUpdateSQL* component) to generate an appropriate SQL statement.

At runtime, when the update component is invoked by a dataset, the update component

1. Selects an SQL statement to execute based on the UpdateKind property for the dataset to which the update component belongs. *UpdateKind* specifies whether the current record is modified, inserted, or deleted..
2. Provides parameter values to the SQL statement.
3. Executes the SQL statement to perform the specified update.

The *Apply* method is useful for manually executing the SQL statements (such as from an *OnUpdateRecord* event handler). It combines a call to *SetParams* to perform the special parameter binding discussed in the previous paragraph and a call to *ExecSQL* to execute the SQL statement.

The *Query* array property returns a *TQuery* object for the SQL statement corresponding to the *TUpdateKind* index. Likewise, the SQL array property returns the string list representing the SQL statement corresponding to the index. Note that SQL returns the same string lists used in the

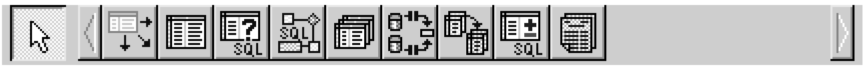
ModifySQL, *InsertSQL*, and *DeleteSQL* properties. For example, SQL[ukInsert] is the same as InsertSQL.

The *TUpdateSQL* component also provides a way to use C++Builder's cached updates support with read-only datasets. For example, you could use a *TUpdateSQL* component with a "canned" query to provide a way of updating the underlying datasets, essentially giving you the ability to post updates to a read-only dataset.

Creating update SQL statements

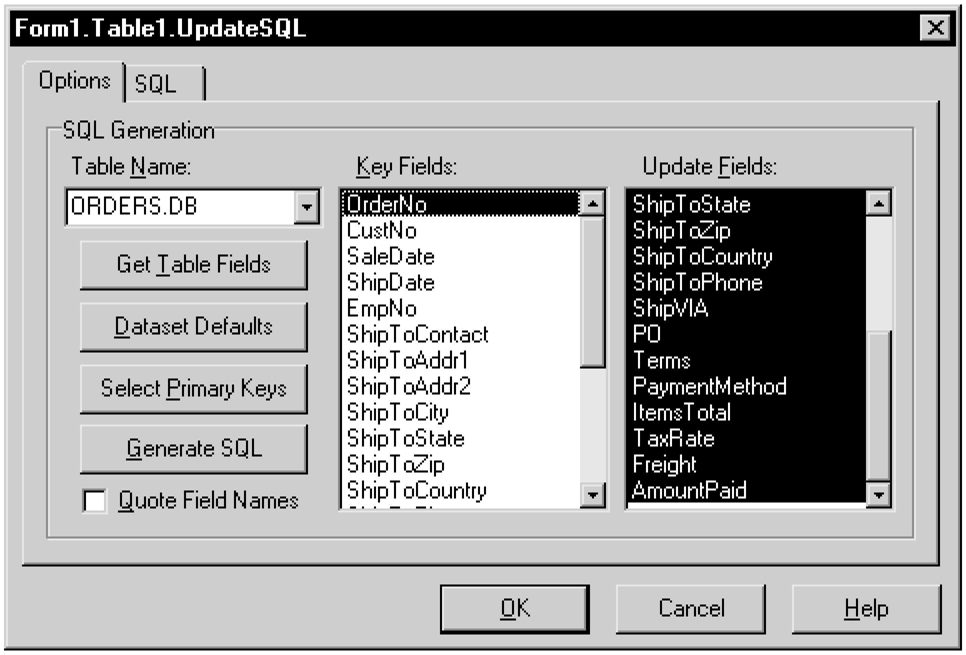
To create the SQL statements for an update component,

1. Add a *TUpdateSQL* component from the Data Access tab of the Component palette to the data module or form in your application if you haven't done so already.



2. Assign the *TUpdateSQL* component to a dataset in the dataset's *UpdateObject* property. This step ensures that the UpdateSQL editor you invoke in the next step can determine suitable default values to use for SQL generation options. The *TUpdateSQL* component can only be referenced from one dataset at a time.

3. Double-click the *UpdateSQL* component to invoke the UpdateSQL editor. The editor generates SQL statements for the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying dataset and on the values you supply to it.



The UpdateSQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired with Shift+Click and Ctrl+Click.

The Key Fields list box is used to specify the columns to use as keys during the update. Generally the columns you specify here should correspond to an existing index, especially for local Paradox and dBASE tables, but having an index is not a requirement.

Click Get Table Fields to display or refresh the list of fields displayed in the Key Fields and Update Fields list boxes.

Click Dataset Defaults to reset the Key Fields and Update Fields list boxes to the dataset defaults.

Click Select Primary Keys to select key fields based on the primary index for a table.

Select the Quote Field Names check box to generate field names with quotation marks. This may be necessary for compatibility with the server.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties and move to the SQL page (the statement for the *ModifySQL* property will be displayed be default).

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Important:

Keep in mind that generated SQL statements are intended to be starting points for creating update statements. You may need to modify these statements to make them execute correctly. Test each of the statements directly before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

Determining how records will be updated

The *UpdateMode* property determines how C++Builder will find records being updated in a SQL database. This property is important in a multi-user environment when users may retrieve the same records and make conflicting changes to them.

When a user posts an update, C++Builder uses the original values in the record to find the record in the database. This approach is similar to an optimistic locking scheme. *UpdateMode* specifies which columns C++Builder uses to find the record. In SQL terms, *UpdateMode* specifies which columns are included in the WHERE clause of an UPDATE statement. If C++Builder cannot find a record with the original values in the columns specified (if another user has changed the values in the database), C++Builder will not make the update and will generate an exception.

The *UpdateMode* property may have the following values:

- *upWhereAll* (the default): C++Builder uses every column to find the record being updated. This is the most restrictive mode.
- *upWhereKeyOnly*: C++Builder uses only the key columns to find the record being updated. This is the least restrictive mode and should be used only if other users will not be changing the records being updated.
- *upWhereChanged*: C++Builder uses key columns and columns that have changed to find the record being updated.

Understanding parameter substitution in generated SQL statements

Generated update SQL statements use a special form of parameter substitution that enables you to determine whether old or new field values are substituted on record update. When the UpdateSQL editor generates its statements, it determines which field values to use.

When the parameter name matches a column name in the table, the new value will be used. When the parameter name matches a column name prefixed by the string "OLD_", then the old value will be used.

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement to determine which record to update.

Executing update SQL statements

When the *ApplyUpdates* method for a database or dataset component is called, update statements associated with an update component execute automatically if the update component is associated with a dataset through the dataset's *UpdateObject* property.

Update statements are *not* executed automatically if you use more than one update component to perform updates on a dataset. In this case (or when you want to carry out additional processing in *ApplyUpdates*), you must write an *OnUpdateRecord* event handler to execute the update statements associated with each update component used.

Creating an update event handler

The *OnUpdateRecord* event occurs when cached updates are enabled (*CachedUpdates* is *true*) and applied (*ApplyUpdates*). It doesn't indicate an error (if there is an error, the *OnUpdateError* event occurs). One *OnUpdateRecord* event occurs for each record being updated. *OnUpdateRecord* lets you manually perform the updates for a read-only dataset. You can use a *TUpdateSQL* component (assigned to the dataset's UpdateObject property) to handle the updates; call *TUpdateSQL*'s *Apply* method, for example, to apply an update based on the SQL statements you provide.

OnUpdateRecord event handlers are supplied with the dataset being updated, a status code indicating the kind of update being performed, and a reference variable parameter that lets you specify how (or whether) the cached update was applied.

The *TUpdateRecord* event type defines the type of method that handles *OnUpdateRecord* events. The *OnUpdateRecord* event handler can be used where a single update component cannot be used to perform the required updates, or when your application needs more control over special parameter substitution.

```
DataSet->UpdateObject->Apply(UpdateKind);
```

To create an *OnUpdateRecord* event handler for a dataset

1. Select the dataset component.

2. Choose the Events page in the Object Inspector.

3. Double-click the *OnUpdateRecord* event value to invoke the code editor.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
void __fastcall TForm1::Query1UpdateRecord(TDataSet *DataSet, TUpdateKind UpdateKind,
TUpdateAction &UpdateAction)
{
 // Perform updates here...
}
```

The *DataSet* parameter specifies the dataset to update. Normally this property is set through a dataset's *UpdateObject* property. When you use more than one update component to update a dataset you must supply this value at runtime.

The *UpdateKind* parameter indicates the type of update to perform. When using an update component, you need to pass this parameter to its methods and properties that you access. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update to perform.

The *UpdateAction* parameter indicates if you applied an update or not. The default value is *uaFail*. Unless you encounter a problem during updating, your event handler should set this value to *uaApplied* before exiting. If you decide not to update a particular record, set the value to *uaSkip* to preserve unapplied changes in the cache.

If you do not change the value for UpdateAction, the entire update operation for the dataset is aborted.

In addition to these parameters, you will typically want to make use of the OldValue and NewValue properties for the field component associated with the current record.

Important:

The *OnUpdateRecord* event, like the *OnUpdateError* and *OnCalcFields* event handlers, should never call any methods that changes which record in a dataset is the current record.

OnUpdateRecord generally uses one or more update components to update a dataset. Using update components is the easiest way to update a dataset, but it is not a requirement. For example, the following code uses a table component to perform updates:

```
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet, TUpdateKind UpdateKind,
TUpdateAction &UpdateAction)
{
 TUpdateSQL *MyUpdateSQL = new TUpdateSQL(this);
 TStringField *EmpAuditSalary = new TStringField(this);
 TStringField *EmpAuditEmpNo = new TStringField(this);
 TVarRec *Field0 = new TVarRec(DataSet->Fields[0]->OldValue);
```

```
     TVarRec *Field1 = new TVarRec(DataSet->Fields[1]->NewValue);
     TLocateOptions MyLocateOptions;
     MyLocateOptions << loPartialKey;
    if (UpdateKind == ukInsert)
     {
      UpdateTable->AppendRecord((Field0,Fields1),2);
     }
     else
       if (UpdateTable->Locate("KeyField", DataSet->Fields[0]->OldValue,MyLocateOptions))
       {
         switch (UpdateKind)
          {
           case ukModify:
           UpdateTable->Edit();
           UpdateTable->Fields[1]->Value = DataSet->Fields[1]->Value;
           UpdateTable->Post();
           }
           case ukModify:
           UpdateTable->Delete();
           }
       UpdateAction = uaApplied;
    }
```

More typically, however, an *OnUpdateRecord* event handler uses two or more update components:

```
    void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet, TUpdateKind UpdateKind,
    TUpdateAction &UpdateAction)
    {
     TUpdateSQL *EmployeeUpdateSQL = new TUpdateSQL(this);
     EmployeeUpdateSQL->Apply(UpdateKind);
     JobUpdateSQL->Apply(UpdateKind);
     UpdateAction = uaApplied;
    }
```

In this example the *DataSet* parameter is not used. This is because the update components called in the handler belong to a data module, and are not referenced through the *UpdateObject* property.

Important:

If you specify both an update component in the *UpdateObject* property for a dataset, and define an *OnUpdateRecord* event handler for it, the update component pointed to in the *UpdateObject* property is not invoked unless you explicitly call its *Apply* method in your handler.

The following sections describe update component properties and methods that you can call from an *OnUpdateRecord* event handler.

Identifying the dataset to update

An update component's *DataSet* property identifies the dataset to update. Normally this property is set through a dataset's *UpdateObject* property. When you use more than one update component to update a dataset, however, you must supply this value at runtime.

Specifying how the current record was changed

UpdateKind is a parameter of type *TUpdateKind*. It describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

Accessing a query component

The *Query* property of an update component provides access to the *TQuery* components called by the update component to perform updates. In most cases you do not need to access this property, but if you do, you can use *UpdateKind* parameter values as an index into the array of query components.

Note: *Query* is not a property of the *UpdateObject*. When you reference the property from a dataset, you must typecast the *UpdateObject* property. For example,

```
((TUpdateSQL*)DataSet->UpdateObject)->Query(UpdateKind);
```

Using an update component's SQL property

At runtime, an update component's *SQL* property provides an indexed method of accessing its *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. Use the values of *UpdateKind* as an index. For example,

```
TUpdateSQL *MyUpdateSQL = new TUpdateSQL(this);
MyUpdateSQL->SQL(ukModify) = ukModify;
```

Normally, the properties indexed by the SQL property are set at design time using the UpdateSQL editor. You might, however, need to access these values at runtime if you are generating a unique UpdateSQL statement for each record and not using parameter binding. The following example generates a unique SQL property value for each row updated:

```
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet, TUpdateKind UpdateKind,
TUpdateAction &UpdateAction)
{
 TUpdateSQL *MyUpdateSQL = new TUpdateSQL(this);
 TStringField *EmpAuditSalary = new TStringField(this);
 TStringField *EmpAuditEmpNo = new TStringField(this);

switch(UpdateKind)
 {
 case ukModify:
   MyUpdateSQL->SQL[UpdateKind]->Text = printf("update emptab set Salary %d where       EmpNo
= %d",EmpAuditSalary->NewValue, EmpAuditEmpNo->OldValue);
   MyUpdateSQL->ExecSQL(UpdateKind);

  case ukInsert:
    ;

   case ukDelete:
     ;
   }
 UpdateAction = uaApplied;
}
```

Note: Rely on the parameter substitutions supplied by update components before building your own SQL statements on the fly.

Applying updates for the current record

The *Apply* method for an update component applies updates for the current record. There are two steps involved in this process:

1. Values for the record are bound to the parameters in the appropriate SQL statement.
2. The SQL statement is executed.

This method is either automatically invoked by the associated dataset, or explicitly called from the *OnUpdateRecord* event. The *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass the *UpdateKind* parameter of the event to *Apply*.

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note: The operations performed by *Apply* correspond to the *SetParams* and *ExecSQL* methods described in the following sections.

Using parameter substitution

The *SetParams* method replaces parameters (indicated by leading colons) in the SQL statement associated with the *UpdateKind* parameter with the value of the field with the same name as the parameter. For example, a parameter named :Sequence would be replaced by the value of the field named Sequence. If the parameter name has OLD_ in front, it is replaced the "old" value of the corresponding field, before any updates were made. For example, :OLD_Name would be replaced by the previous value of the Name field.

SetParams is called automatically when you use the *Apply* method. If you call *Apply* directly in an *OnUpdateRecord* event, do not call *SetParams* yourself.

You need only call *SetParams* directly if you need special control of parameter binding. The following example illustrates one such use:

```
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet, TUpdateKind UpdateKind,
TUpdateAction &UpdateAction)
{
 TStringField *EmpAuditSalary = new TStringField(this);
 TStringField *EmpAuditEmpNo = new TStringField(this);

 if (UpdateKind == ukModify)
   {
    ((TUpdateSQL*)DataSet->UpdateObject)->Query[UpdateKind]->
        ParamByName("DateChanged")->Value = Now();
   ((TUpdateSQL*)DataSet->UpdateObject)->ExecSQL(UpdateKind);
   }
 UpdateAction = uaApplied;
}
```

Note: This example assumes that the *ModifySQL* property for the update component is as follows:

```
UPDATE EMPAUDIT
SET EmpNo = :EmpNo, Salary = :Salary, Changed = :DateChanged
WHERE EmpNo = :OLD_EmpNo
```

In this example, the call to *SetParams* supplies values to the *EmpNo* and *Salary* parameters. The *DateChanged* parameter is not set because the name does not match the name of a field in the dataset, so the next line of code sets this value explicitly.

Executing a prepared update SQL statement

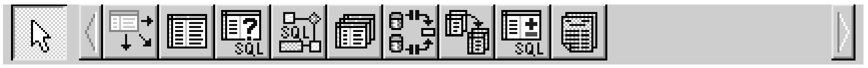
An update component's *ExecSQL* method executes the SQL statement associated with the *UpdateKind* parameter (for example, calling *ExecSQL* and passing *ukModify* would execute the SQL statement held in the *ModifySQL* property). Note that *ExecSQL* does not perform parameter binding; you can do so by calling *SetParams* before calling *ExecSQL*, or by calling *Apply*, which calls *SetParams* then *ExecSQL*. If you call *Apply* directly in an *OnUpdateRecord* event, do not call *ExecSQL* yourself.

If an exception is raised during the execution of the update SQL statement, the *OnUpdateError* event is triggered.

Updating a read-only result set

Although the BDE attempts to provide an updateable, or "live" query result when the *RequestLive* property for a dataset component is *true*, there are some situations where it cannot do so. In these situations, you can manually update a dataset as follows:

1. Add a *TUpdateSQL* component from the Data Access tab of the Component palette to the data module in your application.



2. Set the dataset component's *UpdateObject* property to the name of the TUpdateSQL component in the data module.
3. Enter the SQL update statement for the result set to the update component's *ModifySQL*, *InsertSQL*, or *DeleteSQL* properties, or use the UpdateSQL editor.
4. Close the dataset by calling the dataset's *Close* method or by setting its *Active* property to *false*.
5. Set the dataset component's *CachedUpdates* property to *true*.
6. Reopen the dataset by calling its *Open* method or by setting its *Active* property to *true*.

Note: In many circumstances, you may also want to write an *OnUpdateRecord* event handler for the dataset.

Setting the UpdateObject property for a dataset

A dataset component's *UpdateObject property* is an optional reference to a component of type *TUpdateObject*. When *ApplyUpdates* is called on a read-only dataset, and *UpdateObject* is set, the update object component is invoked for each record that requires updating.

Although there is only one *UpdateObject* property for a dataset component, there is no restriction on the number of update object components that can be used to perform the updates. In cases where more than one update object component is needed, use the OnUpdateRecord event handler for the dataset component to invoke the additional update object components.

Displaying and editing data in data-aware controls

This topic describes how to use data-aware visual components, called visual controls, on a form and how to display and edit data associated with the tables and queries in your database application. A data-aware control derives display data from a database source outside the application, and can also optionally post (or return) data changes to a data source.

In particular, this topic describes the following features common to data-aware controls:

- Associating a data-aware control with a data set
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display

The following topics describe basic features common to all data-aware controls, then describe how and when to use individual components.

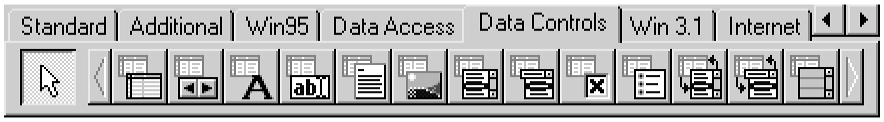
Most data-aware controls are described in the following topics. Omitted are *TDBNavigator*, *TDBGrid*, and TDBCtrlGrid.

Press the >> button to read topics in sequence.

Common data-aware control features

Data control components are data-aware components that you connect to a dataset through a *DataSource* component. You place data-aware controls from the Data Controls tab of the Component palette onto the forms in your database application. Data-aware controls generally enable you to display and edit fields of data associated with the current record in a dataset.

The following figure displays and the following table summarizes the data-aware controls in order from left to right as they appear on the Data Controls tab of the Component palette.



Data-aware control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigating through dataset records, update records, post records, delete records, cancel edits to records, insert records, edit records, and refresh data display.
TDBText	Display data from a field as a label.
TDBEdit	Display and edit data from a field in an edit box.
TDBMemo	Display and edit data from a memo, multi-line text, or BLOB text field in a scrollable, multi-line edit box.
TDBImage	Display and edit a graphics image or binary BLOB data in a graphics box.
TDBListBox	Display a list of choices from which to update a field in the current data record.
TDBComboBox	Display a drop-down list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Display and set a Boolean field condition in a check box.
TDBRadioGroup	Display and set a set of mutually exclusive options for a field.
TDBLookupListBox	Display a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Display a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware combo box.
TDBCtrlGrid	Display a configurable, repeating set of data-aware controls within a grid.

Data Control tab of the Component palette

Data-aware controls are data-aware at design time. When you set a control's *DataSource* property to an active data source while building an application, you can immediately see live data in the controls. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. To scroll through records using the Fields editor, the dataset must be open, and you must have used the Add Fields option to create persistent fields for the dataset. If you've done these things, then you'll notice that the navigator buttons at the top of the Fields editor are enabled. Use them to scroll from record to record. If you have any dataset controls attached to the datasource for the dataset, you can see the values in the controls change as you scroll through the records.

At runtime, data-aware controls also display data and permit editing of data if that is appropriate to the control, to your application, and to the database to which your application connects.

Displaying and editing data in a data-aware control

To display data in a data-aware control,

1. Place a dataset and a data source in a data module or on a form
2. Place a data-aware control from the Data Controls tab of the Component palette onto a form.
3. Set the *DataSource* property of the control to the name of a data source component from which to get data. A data source component acts as a conduit between the control and a dataset containing data.
4. If appropriate, set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property.
5. For list box and radio group controls, enter the strings to be displayed in the control into the *Items* property of the control.

If the *Enabled property* of the data-aware control's data source is *true* (the default), and the *Active* property of the dataset attached to the data source is also *true*, data is now displayed in the data-aware control.

Note: Two data-aware controls, TDBGrid and TDBNavigator, access all available field components within a dataset, and therefore do not have DataField properties. For these controls, omit step 4.

Enabling mouse, keyboard, and timer events

Enabling editing in controls on user entry

Editing data in a control

Disabling and enabling data display

Refreshing data

Enabling mouse, keyboard, and timer events

The *Enabled* property of a data-aware control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *true*.

To prevent mouse, keyboard, or timer events from accessing a data-aware control, set its *Enabled* property to *false*. When *Enabled* is *false*, a data source does not receive information from the data-aware control. The data-aware control continues to display data, but the text displayed in the control does not get highlighted.

Enabling editing in controls on user entry

A dataset must be in dsEdit state to permit editing to its data. The *AutoEdit* property of the data source to which a control is attached determines if the underlying dataset enters *dsEdit* mode when data in a control is modified in response to keyboard or mouse events. When *AutoEdit* is *true* (the default), *dsEdit* mode is set as soon as editing commences. If *AutoEdit* is *false*, you must provide a TDBNavigator control with an *Edit* button (or some other method) to permit users to set *dsEdit* state at runtime.

Editing data in a control

The *ReadOnly* property of a data-aware control determines if a user can edit the data displayed by the control. If *false* (the default), users can edit data. To prevent users from editing data in a control, set *ReadOnly* to *true*.

Properties of the data source and dataset underlying a control also determine if the user can successfully edit data with a control and post changes to the dataset.

The *Enabled* property of a data source determines if controls attached to a data source are able to display fields values from the dataset, and therefore also determines if a user can edit and post values. If *Enabled* is *true* (the default), controls can display field values.

The *ReadOnly* property of the dataset determines if user edits can be posted to the dataset. If *false* (the default), changes are posted to the dataset. If *true*, the dataset is read-only.

Note: Table components have an additional, read-only runtime property *CanModify* that determines if a dataset can be modified. *CanModify* is set to *true* if a database permits write access. If *CanModify* is *false*, a dataset is read-only. Query components that perform inserts and updates are, by definition, able to write to an underlying database, provided that your application and user have sufficient write privileges to the database itself.

The following table summarizes the factors that determine if a user can edit data in a control and post changes to the database:

Data-aware control ReadOnly property	**Data source Enabled property**	**Dataset ReadOnly property**	**Dataset CanModify property (tables only)**	**Database write access**	**Can write to database?**
false	true	false	true	Read/Write	Yes
false	true	false	false	Read-only	No
false	false	--	--	--	No
true	--	--	--	--	No

In all data-aware controls except TDBGrid, when you modify a field, the modification is copied to the underlying field component in a dataset when you move from the control. If you press Esc before you Tab from a field, C++Builder abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In TDBGrid, modifications are copied only when you move to a different record; you can press Esc in any field of a record before moving to another record to cancel all changes to the record.

When a record is posted, C++Builder checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, C++Builder throws an exception, and no modifications are made to the record.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...catch** statement so that you can re-enable controls even if an exception occurs during processing. The **catch** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable->DisableControls();
try
{
 CustTable->First(); // Go to first record, which sets Eof false
 while(!CustTable->Eof) //Cycle until Eof is true
 {
   // Process each record here
   ...
   CustTable->Next(); // Eof false on success; Eof true when Next fails on last record
 }
 CustTable->Refresh();
 CustTable->EnableControls();
}
catch(...)
{
 CustTable->Refresh();
 CustTable->EnableControls();
}
```

Refreshing data

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application.

Important:

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*. You may with to call *Refresh* prior to deleting or updating data.

Displaying fields as labels

TDBText is a read-only control similar to the *TLabel* component on the Standard tab of the Component palette. TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic--the text changes as the user navigates the database table. Therefore, you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

A *TDBText* control cannot be modified by the user. If you want to enable users to modify the contents of the field, use a *TDBEdit control* instead.

A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A TDBText component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

When you place a TDBText component on a form, set its AutoSize property to *true* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If AutoSize is set to *false*, and the control is too small, data display is truncated. To wrap text displayed in the label, set *WordWrap* to *true*. Setting *WordWrap* to *true* causes text entered after a space to wrap to the next line, unless there is still room in the current line. To specify whether you want the label text to align left, right, or centered, use the *Alignment* property. To prevent the label from obscuring other components, set its *Transparent* property to *true*.

Displaying and editing fields in an edit box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques. Use the *TDBEdit* component to read or write a single line of data from a specific column in the current record of a dataset. To read and write multiple lines of text, use the *TDBMemo component*. To display text that the user cannot modify, use the *TDBText component* (or set the *ReadOnly* property of the *TDBEdit* component to *true*).

For example, suppose *CustomersSource* is a TDataSource component that is active and linked to an open TTable called *CustomersTable*. You can then place a TDBEdit component on a form and set its properties as follows:

- DataSource: *CustomersSource*
- DataField: *CustNo*

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime. If the *DataField* value of the edit box is an integer or floating point value, only characters that are valid in such a field can be entered in the edit box. Characters that are not legal are not accepted.

To specify the font of the text, use the *Font* property. To specify the line length, set the *MaxLength* property. To automatically select the text when the edit box receives focus, set *AutoSelect* to *true* (default). To prevent the height of the edit box from changing dynamically to accommodate font changes, set the *AutoSize* property to *false* (*true* is the default). Only the height, not the length (*Width*), of the edit box is affected by the *AutoSize* property. To specify a case for the text in the edit box (all upper, all lower, or mixed case), use the *CharCase* property.

Displaying and editing text fields in a memo control

TDBMemo is a data-aware component--similar to the Standard TMemo component--that can display and modify a multi-line field in a dataset or formatted text in binary large object (BLOB) format. TDBMemo displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields. Use the TDBEdit component to display or edit a single line of text. Use the TDBImage component to display graphic images in BLOB format.

By default, TDBMemo permits a user to edit memo text. To prohibit editing, set the *ReadOnly* property of *TDBMemo* to *true*. To permit users to enter tabs in a memo, set the *WantTabs* property to *true*. To limit the number of characters users can enter into the database memo, use the MaxLength property. The default value for *MaxLength* is *0,* meaning that there is no limit on the number of characters the control can contain. Any number other than *0* limits the number of characters the control accepts.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the ScrollBars property. To prevent word wrap, set the WordWrap property to *false*. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

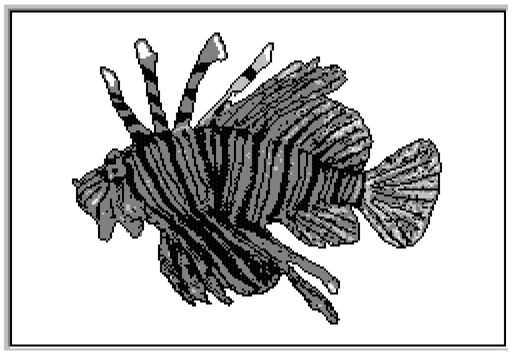
At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be automatically displayed. If you set *AutoDisplay* to *false*, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing graphics fields in an image control

TDBImage is a data-aware component that displays bitmapped graphics or BLOB data in image format. It captures BLOB graphics images from a dataset, and stores them internally in the Windows .DIB format. *TDBImage* cannot display formatted text in BLOB format. Use the *TDBMemo component* for this purpose.

TDBImage component



By default, TDBImage permits a user to edit a graphics image by cutting and pasting to and from the Clipboard. You can also supply your own editing methods. For example, you can use the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should be automatically displayed. If you set *AutoDisplay* to false, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Set the *Stretch* property to *true* to allow the graphic image to automatically stretch or shrink to fit into the available space. When *Stretch* is set to *false* (the default), as much of the graphic as fits will be displayed in the control.

Displaying and editing data in list and combo boxes

Four data-aware controls provide data-aware versions of standard list box and combo box controls. These useful controls provide the user with a set of default data values to choose from at runtime.

Note: Data-aware list and combo box can be linked only to data sources for table components. They do not work with query components.

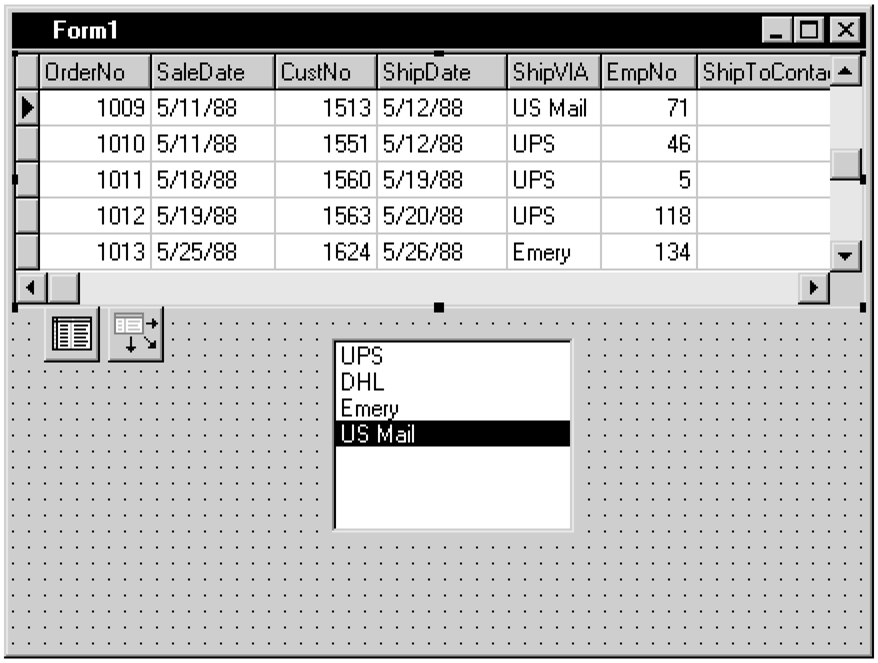
The following table describes these controls:

Data-aware control	Description
TDBListBox	Displays a list of items from which a user can update a field in the current record. The list of display items is a property of the control.
TDBComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is a property of the control.
TDBLookupListBox	Displays a list of items from which a user can update a column in the current record. The list of display items is looked up in another dataset.
TDBLookupComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is looked up in another dataset.

Displaying and editing data in a list box

TDBListBox displays a scrollable list of items from which a user can update a data field in the current record of the dataset. A data-aware list box displays the current value for a field in the current record and highlights its corresponding entry in the list. In the following figure, the current record is *OrderNo* 1009, and the corresponding *ShipVia* field is US Mail. The list box is highlighting the corresponding entry and will allow the user to change the value in this field by selecting a different value from the list box. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

The TDBListBox component shown with a TDBGrid component



OrderNo	SaleDate	CustNo	ShipDate	ShipVIA	EmpNo	ShipToConta
1009	5/11/88	1513	5/12/88	US Mail	71	
1010	5/11/88	1551	5/12/88	UPS	46	
1011	5/18/88	1560	5/19/88	UPS	5	
1012	5/19/88	1563	5/20/88	UPS	118	
1013	5/25/88	1624	5/26/88	Emery	134	

UPS
DHL
Emery
US Mail

The *Items* property of the *TDBListBox* component, which specifies the items contained in the list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. Use the String list editor at design time to create the list of items to display.
The *Height* property determines the vertical size of a control in pixels. The IntegralHeight property controls the way the list box is displayed. If IntegralHeight is *false* (the default), the bottom of the list box is determined by the ItemHeight property, and the bottom item might not be completely displayed. If IntegralHeight is *true* the visible bottom item in the list box is fully displayed. To specify the height of each item, set the *ItemHeight* property. You must also set the *Style* to *csOwnerDrawFixed*.

Displaying and editing data in a combo box

TDBComboBox is a data-aware version of the *TComboBox* component. The TDBComboBox control combines the functionality of a data-aware edit control and a drop-down list. At runtime, users can update a field in the current record of a dataset by typing a value or choosing a value from the drop-down list.

The Items property of the component, which specifies the items contained in the drop-down list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. Use the String list editor to populate the *Items* list.

When a control is linked to a field through its *DataField* property, it displays the value for the field in the current row, regardless of whether it appears in the Items list. The *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the Items list is displayed at runtime:

- Style determines the display style of the component:
- csDropDown (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
- csSimple: Does not display the drop-down list.
- csDropDownList: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
- csOwnerDrawFixed and csOwnerDrawVariable: Allows the items list to display values other than strings (for example, bitmaps). For more information, see the VCL Reference manual.
- DropDownCount: the maximum number of items displayed in the list. If the number of Items is greater than DropDownCount, the user can scroll the list. If the number of Items is less than DropDownCount, the list will be just large enough to display all the Items.
- ItemHeight: The height of each item when *Style* is csOwnerDrawFixed.
- Sorted: If *true*, then the Items list is displayed in alphabetical order.

Looking up data for displaying and editing in list and combo boxes

TDBLookupListBox and *TDBLookupComboBox* are a data-aware list box and a data-aware combo box. The *TDBLookupListBox* component displays a scrollable list of available choices, the *TDBLookupCombo* displays a drop-down list. Both controls derive their list of display items dynamically from a second dataset, known as the lookup dataset, at runtime, from one of two sources:

- Lookup field defined for a dataset.
- Secondary data source, data field, and key.

In either case, a user is presented with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names or a *TDBLookupComboBox* that displays a drop-down list of available company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

Setting lookup list and combo box properties

Specifying a list based on a lookup field

To specify list box or combo box list items using a lookup field, the dataset to which you link the control must already define a lookup field.

To specify a lookup field for the list box or drop-down list items,

1. Set the *DataSource* property of the box to the data source for the dataset containing the lookup field to use (for example, *OrdersSource*).
2. Choose the lookup field to use from the drop-down list for the *DataField* property.

When you activate a table associated with a lookup list box control or combo box control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

Specifying a list based on a secondary data source

If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item.

To specify a secondary data source for list box items or drop-down list items, set these properties:

1. Set the *DataSource property* of the list box or drop-down box to the data source for the dataset to lookup values for (for example, *OrdersSource*).
2. Choose a field into which to insert looked-up values from the drop-down list for the *DataField property* (for example, *CustNo*). The field you choose cannot be a lookup field.
3. Set the ListSource property of the list box or combo box to the data source for the dataset that contain the field whose values you want to look up (for example, *CustomersSource*).
4. Choose a field to use as a lookup key from the drop-down list for the KeyField property (for example, *CustNo*). The drop-down list displays fields for the dataset associated with data source you specified in step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
5. Choose a field whose values to return from the drop-down list for the ListField property. The drop-down list displays fields for the dataset associated with the data source you specified in step 3. To display multiple fields, separate the field names with semi-colons (for example, *CustNo*; *Company*).

When you activate a table associated with a lookup list box or combo box control, the control recognizes that its drop-down list items are derived from a secondary source, and displays the appropriate values from that source.

Setting lookup list and combo box properties

The following table lists important properties specific to lookup list and combo boxes:

Property	Purpose
DataField	Specifies the field in the master dataset which provides the key value to be looked up in the lookup dataset. This field is modified when a user selects a list box or drop-down list item. If *DataField* is set to a lookup field, the *KeyField*, *ListField*, and *ListSource* properties are not used (for example, *CustNo* from *OrdersSource*).
DataSource	Specifies a data source for the control. This should be the data source for the master dataset. If the selection in the control is changed, this dataset is placed in *dsEdit* mode (for example, *OrdersSource*).
KeyField	Specifies the field in the lookup dataset corresponding to *DataField*. The control searches for the *DataField* value in the *KeyField* of the lookup dataset. The *KeyField* property column must contain the same values as the *DataField* property column, although the column names can differ. The lookup dataset should have an index on this field to facilitate lookups (for example, *CustNo* from *CustomersSource*).
ListField	Specifies the field of the lookup dataset to display in the control (for example, *CustNo*; *Company* from *CustomersSource*).
ListSource	Specifies a data source for the dataset you want the control to use to look up the information you want displayed in the control. The sort order of items displayed in the list box or drop-down list is determined by the index specified by the *IndexName* property of the lookup dataset. That index need not be the same one used by the *KeyField* property (for example, *CustomersSource*).
RowCount	Specifies the number of lines of text to display in the list box. The height of the list box is adjusted to fit this row count exactly.
DropDownRows	Specifies the number of lines of text to display in the drop-down list.

Note: At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing ROB selects the first item in the list box beginning with the letters ROB. Typing an additional E selects the first item starting with ROBE, such as Robert Johnson. The search is case-insensitive. Backspace and Esc cancel the current search string (but leave the selection intact), as does a two-second pause between keystrokes.

Handling Boolean field values with check boxes

TDBCheckBox is a data-aware version of the *TCheckBox* component. It can be used to display or edit the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The *TDBCheckbox* controls its own checked or unchecked state by comparing the contents of the field to the contents of the *ValueChecked* and *ValueUnchecked* properties (both values cannot match at the same time). If the value of the *ValueChecked* property matches the value of the field, C++Builder checks the check box. If the value of the *ValueUnchecked* property matches the value of the field, C++Builder unchecks the check box.

Set the ValueChecked property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to *True*, but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of ValueChecked. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a ValueChecked string like

```
DBCheckBox1->ValueChecked = "True;Yes;On";
```

If the field for the current record contains values of "True," "Yes," or "On," then the check box is checked. Comparison of the field to ValueChecked strings is case-insensitive. If a user checks a box for which there are multiple ValueChecked strings, the first string is the value that is posted to the database.

Set the ValueUnchecked property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to *False*, but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of ValueUnchecked. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is gray-checked (not True and not False) whenever the field for the current record does not contain one of the values listed in the ValueChecked or ValueUnchecked properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the ValueChecked and ValueUnchecked properties have no effect on logical fields.

To group check boxes in the form, place them all inside a single panel, group box, or scroll box component. You must first place the container component on the form, then place the check box components inside it.

Use the *Caption* property to display a label for the check box on your form. To place the check box to the right or left of the text in its caption, set its *Alignment* property.

To make a check box unavailable to the user, set its *Enabled* property to *false*. To enable the user to gray the check box, set its *AllowGrayed* property to *true*.

Restricting field values with radio controls

TDBRadioGroup is a data-aware version of the *TRadioGroup* component. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group consists of one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the number of radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button's label.

If the current value of a field associated with a radio group matches one of the strings in the Items property, that radio button is automatically selected. For example, if three strings, Red, Yellow, and Blue, are listed for Items, and the field for the current record contains the value Blue, then the third button in the group is selected.

Note: If the field does not match any strings in Items, a radio button may still be selected if the field matches a string in the Values property. If the field for the current record does not match any strings in Items or Values, no radio button is selected.

The Values property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains Red, Yellow, and Blue, and *Values* contains Magenta, Yellow, and Cyan. If a user selects the button labeled Red, Magenta is posted to the database.

If strings for Values are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

Using data modules

This topic describes data modules, a C++Builder feature that enables you to centralize data access for a database application.

Press the >> button to read through topics in sequence.

Understanding data modules

A *data module* is a specialized C++Builder class for centralized handling of any nonvisual component in an application. Typically these are data-access components (*TSession*, *TDatabase*, *TTable*, *TQuery*, *TStoredProc*, and *TBatchMove*), but they can also be other nonvisual components (*TTimer*, *TOpenDialog*, *TSaveDialog*, *TImageList*, and *TDdeClientConv*, for example). A data module enables you to:

- Place all your data-access components in a single visual container at design time instead of duplicating them on each application form.
- Design tables and queries once for use with many forms instead of recreating them separately for each form.
- Create business rules--using component events, and additional methods you add to the unit code for a data module--that can be shared across an entire application.
- Separate business logic and data access from user interface code for easier maintenance.
- Standardize common dialogs, timers, DDE client, DDE client items, DDE server, DDE server items, and image lists across an application.
- Store well-designed data-access modules in the Object Repository to share with other projects and developers.

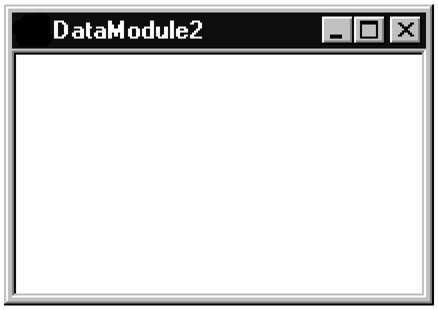
To use a component from a data module in a form, choose File|Include Unit Hdr to add the data module header file to the form unit. Then use the component's methods or properties in the form. For data-access components, set the *DataSource* property of each data-aware control in the form to point to the datasource in the data module.

At runtime a data module is not visible. Your application code can change or read the properties of the components in the data module, and it can call the methods belonging to those components.

Creating a new data module

To create a new, empty data module for a project, choose File|New Data Module. C++Builder opens a data module container on the Integrated Development Environment (IDE) desktop and adds the data module to the project file.

Blank data module



At design time a data module looks like a standard C++Builder form with a white background and no alignment grid. As with forms, you can place nonvisual components on a module from the Components palette, and you can resize a data module to accommodate the components you add to it. You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a module:

Menu item	Purpose
Align To Grid	Aligns data-access components to the data module's invisible grid.
Align	Aligns data-access components according to criteria you supply in the Alignment dialog box.
Revert to Inherited	Discards changes made to a data module inherited from another data module in the Object Repository, and reverts to the originally inherited data module.
Creation Order	Enables you to change the order in which data-access components are created at start-up.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties. You can view the properties as text only when the dataset is not active (*Active* = *false*).

Behind the data module container you see in the IDE, there is a corresponding unit file containing source code for the data module.

Naming a data module

The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a data module to it before doing any other application building, the name of the data module defaults to "DataModule2". The corresponding unit file for DataModule2 defaults to "Unit2", with the header file becoming UNIT2.H and the .CPP file becoming UNIT2.CPP.

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your data modules.

To rename a data module:

1. Select the data module.
2. Edit the *Name* property for the data module in the Object Inspector and press Enter.

The new name for the data module appears in the title bar when the *Name* property in the Object Inspector is updated.

Changing the name of a data module at design time changes its name in both the header and the .CPP file. It also changes any use of the data module name in function declarations. You must manually change any references to the data module in code you write.

To rename a unit for a data module:

1. Select either the .H or .CPP file.
2. Choose File|SaveAs.
3. In the Save As dialog box, enter a file name that clearly identifies the unit with the renamed data module.

Placing and naming components

You place nonvisual components, such as *TTable* and *TQuery*, in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visible controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, C++Builder assigns it a generic name that identifies what kind of component it is, such as *DataSource1* and *Table1*. This makes it easy to select specific components whose properties and methods you want to work with. To make it even easier, you should give your components more descriptive names (for example, *CustSource* and *CustTable*).

To change the name of a component in a data module:

1. Select the component.
2. Edit the component's *Name* property in the Object Inspector and press Enter.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the Object Inspector is updated.

When you name a component, the name you give it should reflect the type of component and what it is used for. For example, for database components, the name should reflect the type of component, and the database it accesses. For example, suppose your application uses the CUSTOMER table. To access CUSTOMER you need a minimum of two data-access components: a datasource component and a table component. When you place these components in your data module, C++Builder assigns them the names *DataSource1* and *Table1*. To reflect that these components use CUSTOMER, and to relate the components to one another, you could change these names to *CustSource* and *CustTable*.

Using component properties and methods in a data module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TTable* and *TQuery*, to control the data available to the datasource components that use those datasets. Setting the *ReadOnly* property to *true* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset to restrict the fields within a table or query that are available to a datasource and therefore to the data-aware controls on forms.

The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a *TDataSource* component has three possible events, *OnDataChange*, *OnStateChange*, and *OnUpdateData*. A *TTable* component has over twenty potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

For a complete list of the properties and events available for components, see the individual component entries in the *VCL Reference*.

Creating business rules in a data module

Besides writing event handlers for the components in a data module, you can code methods directly in the .H and .CPP files for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a function to perform month-, quarter-, or year-end bookkeeping. You might call the function from an event handler for a component in the data module.

The prototypes for the functions you write for a data module should appear in the data module's class declaration:

```
class TDataModule2 : public TDataModule
{
__published:
   TTable *Customers;
   TTable *Orders;
   ...
private:
   // private user declarations
public:
   // public user declarations
   virtual __fastcall TDataModule2(TComponent * Owner);
   void LineItemsCalcFields(DataSet *TDataSet); // A function you add
};
extern TDataModule2 *DataModule2;
```

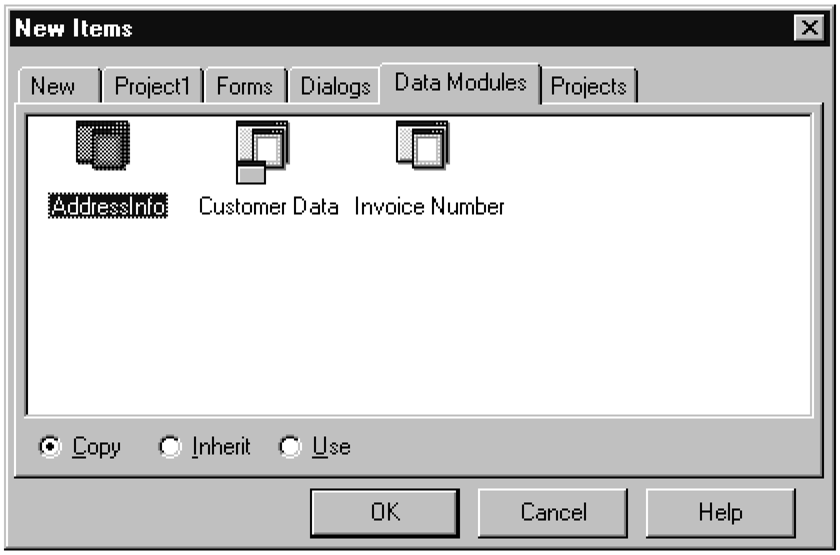
Write the code for the functions in the .CPP file for the data module.

Reusing data modules in the Object Repository

Instead of creating a data module from scratch, you may be able to reuse an existing data module from the Object Repository. Data modules in the Object Repository are modules that you or other developers created that are generic enough to be of use in different projects.

You can reuse an existing data module from the Object Repository at any time. To see the data modules available in the Object Repository:

1. Choose File|New.
2. Select the Data Module page in the New Items dialog box.



The Data Modules page displays the data modules you can use. Radio buttons at the bottom of the page enable you to specify the method of reuse.You can borrow an object in one of three ways: Copy, Inherit, and Use. Sometimes one or more of these options may be unavailable. Unavailable options are dimmed.

Copying a data module

Copying a data module from the Object Repository puts an exact duplicate of the data module and its code in your project. Your copy of a data module is like a snapshot of the data module in the repository at the time of the copy operation. The data module copied into your project exists independently of its ancestor data module in the repository.

Copy a data module from the repository when

- The data module already provides a fundamental level of access to data your application needs,
- The demands of your application require substantial changes and additions to the components in the data module, and
- You do *not* want changes made to the ancestor data module in the repository to affect the data module in your application.

Note: If you want future changes made to ancestor data modules in the repository to ripple into your project, you should inherit the data module instead of copying it.

Inheriting a data module

Inheriting a data module from the Object Repository creates a duplicate of the data module in your project, and creates a link to the ancestor data module in the repository. Your copy of the data module inherits any subsequent changes made to the components, properties, and methods of the repository's data module. Changes made to the data module in the repository are applied to the inherited data module in your project the next time you recompile. These changes apply in addition to any changes or additions you make to the data module in your project.

If you add components and event handlers to your copy of an inherited data module, you only generate new code in your application for the added components and event handlers.

Inherit a data module from the Object Repository when

- The data module provides a fundamental level of access to data your application needs, and
- You want to propagate changes made to the ancestor data module in the repository to the data module in your application (for example, because you want to maintain a consistent corporate view of data in all your applications).

Note: Do not inherit data modules that contain *TSession* or *TDatabase* components. You should copy these data modules instead of inheriting them. A second option is to put *TSession* and *TDatabase* components in a separate data module that you can use, and to put all other components in another data module that you can inherit.

Inheriting event handlers

If you override an event handler for a component inherited from a data module (for example, to provide functionality specific to your application), you can still invoke the inherited handler from your new handler by qualifying the name of the event handler with the name of the data module.

Using a data module

You can both copy a data module from the Object Repository into your application and make changes to it that replace the original data module in the repository by selecting the Use option in the New Items dialog box. Using a data module is like reverse inheritance. Instead of inheriting changes others make to a data module, they inherit your changes when they use the data module in the repository.

"Use" a data module in the Object Repository when

- The data module already provides a fundamental level of access to data your application needs,
- The demands of your application require substantial changes and additions to the components in the data module, and
- The changes you need to make in your application are also fundamentally applicable to other applications.

Important:

Be careful with the Use radio button option. Make sure the changes you make to a data module are robust and thoroughly tested before letting others copy it into their applications from the repository.

Accessing a data module from a form

To associate visual controls on a form with a a data module, you must first make the data module available to the form. To make the data module available to the form, you can

- Choose File|Include Unit Hdr and enter the name of the data module or pick it from the list box in the Use Unit dialog box, or
- Drag a selected field from the Fields editor of a *TTable* or *TQuery* component in the data module to a form. In this case C++Builder prompts you to confirm that you want to add the data module to the form, then creates controls based on data dictionary information for each field you dragged into the form.

Before you drag fields to a form, set the datasource for the fields you intend to drag. C++Builder uses an existing datasource in the data module if it is already connected to the dataset.

If a datasource is not defined, C++Builder adds a new datasource component to the form and sets the *DataSource* properties of the controls it creates to point to this datasource. The datasource itself is associated with a table or query component in the data module. Should this happen, you can keep this arrangement, or, to keep your data-access model cleaner, you can change it. Delete the datasource component on the form, and set the *DataSource* properties of the control on the form to point directly to the appropriate datasource in the data module.

Adding a data module to the Object Repository

You can save your data modules and add them to those already available in the Object Repository. This is helpful when you want to develop standard data-access models throughout an organization.

Before adding a data module to the Object Repository, you should develop and test it as fully as possible to minimize additional work for future users of the data module.

To store a data module in the Object Repository:

1. Save the data module.
2. Right-click the data module to bring up the context menu.
3. Choose Add to Repository.
4. Enter a title, description, and author for the data module in the Add to Repository dialog box.
5. Select Data Modules from the Page combo box, and choose OK.

Linking visual controls to datasets

This topic describes the *TDataSource* component that acts as a conduit between a dataset (*TTable*, *TQuery*, and *TStoredProc* components) and the data-aware, visual control components (from the Data Controls tab) in a form.

Every dataset must have a corresponding data source component to make use of any data-aware controls. If your application is not using data-aware controls, you probably will not need a data source component. A data source component links visual database controls on forms to a dataset. Visual database controls receive data from and send data to data source components. A data source component channels data from a dataset to visual controls, and from visual controls back to the dataset.

"Understanding database components" provides more information about data sources and their relationship to other database components.

Press the >> button to read through topics in sequence.

Establishing a database-to-data-aware-control link

Responding to data changes

Responding to updates

Responding to dataset state changes

Other places to look:

Grid controls

Data-aware controls

Working with tables

Working with queries

Establishing a database-to-data-aware-control link

A *TDataSource* component enables the display, navigation, and editing of the data underlying the dataset. All datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form. Similarly, each data-aware control needs to be associated with a data source component in order to have data to display and manipulate. You also use data source components to link datasets in master-detail relationships.

You place a data source component in a data module or form just as you place other nonvisual database components. You should place at least one data source component for each dataset component in a data module or form.

To establish a database-to-data-aware-control link:

1. Place a dataset component from the Data Access tab of the Component palette in a data module or on a form. See "Surfacing datasets" for information on associating a dataset component with a database. To view data in a data-aware control, make sure the dataset's *Active* property is *true*.
2. Place a *DataSource* component from the Data Access tab of the Component palette in a data module or on a form. Set the DataSet property of the data source to the name of the dataset established previously.

3. Place a data-aware control from the Data Controls tab of the Component palette on a form. Set its *DataSource* property to the name of the data source established previously.

TDataSource has only a few published properties and events. The following topics discuss these key properties and events and how to set them at design time and runtime, when applicable.

Associating the data source with a dataset

Naming the data source

Connecting to a dataset

Permitting automatic editing

Setting an integer value for programming needs

Associating the data source with a dataset

The *DataSet* property specifies the dataset component (*TTable*, *TQuery*, and *TStoredProc*) in a form or data module that is providing data to the data source. Usually you select a dataset at design time from the drop down list in the Object Inspector, but you can also set it at runtime. The advantage of this interface approach to connecting data components is that the dataset, data source, and data-aware controls can be connected and disconnected from each other through the *TDataSource* component. In addition, these components can belong to different forms.

The following code swaps the dataset for the *CustSource* data source component between *Customers* and *Orders* to show how you can switch the dataset for a data source component as needed at runtime.

```
if (CustSource->DataSet == "Customers")
 CustSource->DataSet = "Orders";
else
 CustSource->DataSet = "Customers";
```

You can also set the DataSet property to a dataset on another form to synchronize the data-aware controls on the two forms. For example,

```
void __fastcall TForm2::FormCreate (TObject *Sender)
{
 DataSource1->Dataset = Form1->Table1;
}
```

Naming the data source

Name enables you to specify a meaningful name for a data source component that distinguishes it from all other data sources in your application. The name you supply for a data source component is displayed below the component's icon in a data module and in the Object Selector at the top of the Object Inspector.

Generally, you should provide a name for a data source component that indicates the dataset with which it is associated. For example, suppose you have a dataset called *Customers*, and that you link a data source component to it by setting the data source component's *DataSet* property to "Customers." To make the connection between the dataset and data source obvious in a data module, you should set the *Name* property for the data source component to something like "CustomersSource".

Connecting to a dataset

The *Enabled* property specifies if the display in data-aware controls connected to *TDataSource* is updated when the current record in the dataset changes. During an *UpdateState* call, when the *Enabled* property is *true*, the *State* property of the data source would be set match the state of the dataset. When the *Enabled* property is *false*, the state of the data source would be set to *dsInactive*. During a *SetState* call, if the state of the data source changes, *OnStateChange* is called, all of the data links are updated via a call to *OnDataChange* or *OnUpdateData*. If *State* changes from *dsInactive*, *OnDataChange* is also called. For example, when *Enabled* is *true* and the *Next* method of a dataset component is called many times, each call updates all controls. Setting *Enabled* to *false* allows the *Next* calls to be made without performing updates to the controls. Once you reach the desired record, set *Enabled* to *true* to update the controls to that record.

Note: Setting *Enabled* to *false* clears the display in data-aware controls until you set it to *true* again. If you want to leave the controls with their current contents while moving through the table or query, call the *DisableControls* and *EnableControls methods*.

Permitting automatic editing

The AutoEdit property of TDataSource specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If AutoEdit is *true* (the default), C++Builder automatically puts the dataset in Edit state when a user types in a linked data-aware control. Set *AutoEdit* to *false* to protect the data from being unintentionally modified. When *AutoEdit* is *false*, a dataset enters Edit state only when the application explicitly calls its Edit method.

Setting an integer value for programming needs

The *Tag* property is available to store an integer value as part of a component. While the *Tag* property has no meaning to C++Builder, your application can use the property to store a value for its special needs.

Responding to data changes

The *OnDataChange* event occurs when the *State* property changes from *dsInactive*, or when a data-aware control notifies the *TDataSource* that something has changed.

Notification occurs when the following items change because of field modification or moving to a new record: field component, record, dataset component, content, and layout. The *Field* parameter to the method may be null if more than one of the fields changed simultaneously (as in a move to a different record). Otherwise, *Field* is the field which changed.

The *TDataChangeEvent* type points to a method that handles the changing of data in a data source component (*TDataSource*). The *Field* parameter is the field in which the data is changing. It is used by the *OnDataChange* event of the data source.

This event is useful if an application is keeping components synchronized manually.

Responding to updates

OnUpdateData is activated by the *Post* or *UpdateRecord* method of a dataset component when the current record is about to be updated in the database. For instance, an OnUpdateData event occurs after Post is called, but before the data is actually posted to the database. It causes all data-aware controls connected to the data source to be notified of the pending update, allowing them to change their associated fields to the current values in the controls.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

Responding to dataset state changes

OnStateChange is called whenever the *State* property for a data source's dataset changes. *OnStateChange* is useful for performing actions as a TDataSource's state changes. A dataset's State property records its current state. The *State* property is a public, but not published, read-only property that has no write method. However, the *State* property of a dataset may be changed indirectly. For example, you can change the *State* property to *dsEdit* programmatically by calling the dataset's *Edit* method as in the following code example.

```
Table1->Edit();
```

By assigning a method to this property, you can react programmatically to state changes. This event is useful for enabling or disabling buttons (for example, enabling an edit button only when a table is in edit mode) or displaying processing messages.

Note: *OnStateChange* can occur even for null datasets, so it is important to protect any reference to the *DataSet* property with a null test:

```
if (DataSource1->Dataset != NULL)
...
```

For example, during the course of a normal database session, a dataset's state changes frequently. To track these changes, you could write an OnStateChange event handler that displays the current dataset state in a label on a form. The following code illustrates one way you might code such a routine. At runtime, this code displays the current setting of the dataset's *State* property and updates it every time it changes:

```
void __fastcall TForm1::StateChange(TObject *Sender)
{
 char S[10];

 switch (CustTable->State)
 {
   case dsInactive: strcpy(S,"Inactive");
   case dsBrowse: strcpy(S, "Browse");
   case dsEdit: strcpy(S, "Edit");
   case dsInsert: strcpy(S, "Insert");
   case dsSetKey: strcpy(S, "SetKey");
 }
}
```

Similarly, you can use OnStateChange to enable or disable buttons or menu items based on the current state:

```
void __fastcall TForm1::StateChange( TObject *Sender)
{
 if (CustTable->State == dsBrowse)
   CustTableEditBtn->Enabled;
 if (CustTable->State == (dsInsert|dsEdit|dsSetKey))
   CustTableCancelBtn->Enabled;
 ...
}
```

Accessing data in databases

This topic describes the C++Builder components that encapsulate the data in a database. These components are *TDataSet* and its descendents, *TTable*, *TQuery*, and *TStoredProc*.

Press the >> button to read through topics in sequence.

Providing common dataset functionality

Taken collectively, *TTable*, *TQuery*, and *TStoredProc* are called "datasets." The following figure highlights the relation of dataset components to all the data access components in C++Builder:

C++Builder data access component hierarchy

Interacting with the database

You never use *TDataSet* directly in your applications. Instead, you read and write data in a database table by using a *TTable*, *TQuery*, or *TStoredProc* component that is descended from *TDataSet* through *TDBDataSet*. (*TDBDataSet* adds session and database properties and methods to *TDataSet*.) *TDataSet*, however, provides functionality common to all its descendents. To learn more about this common functionality, see "Surfacing datasets".

Every dataset must have a corresponding *TDataSource* component in order to display data in data-aware controls. A data source component links visual database controls on forms to a dataset. Visual database controls receive data from and send data to data source components. A data source component channels data from a dataset to visual controls, and from visual controls back to the dataset. To learn more about *TDataSource*, see "Linking visual controls to datasets."

Surfacing datasets

A dataset component encapsulates data in a database, such as a table or the result set returned by a query. *TDataSet* defines much basic functionality common to dataset components.

You never use a *TDataset* component directly. Instead you use one of its descendants: *TTable*, *TQuery*, or *TStoredProc*. Each of these components has features that make it useful in different situations. The three sections that follow briefly introduce these descendants.

Accessing all of your data through tables

TTable is the most fundamental and flexible dataset component class in C++Builder. A table component gives you access to every row and column in an underlying database table, whether it is from Paradox, dBASE, an ODBC-compliant database such as Microsoft Access, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

If you are familiar with a desktop database such as Paradox or dBASE, you will be very comfortable working with table components because they most closely emulate the desktop paradigm. Table components offer you an easy transition to C++Builder's client/server database applications model.

If you are more familiar with SQL databases and are already comfortable developing multiuser client/server applications, you may find that the unique features of the *TTable* component (such as batch move operations and easy setup of master/detail forms) offers you additional flexibility and power when building a user interface to databases on your remote servers.

Accessing a subset of data with queries

TQuery is a BDE wrapper around an SQL statement that returns a set of related rows and columns from one or more database tables. Generally, a query component provides access to a subset of rows and columns in its underlying tables, whether those tables are from Paradox, dBASE, an OBDC-compliant database such as Microsoft Access, or an SQL database on a remote server, such as InterBase, Sybase, Oracle, or SQL Server.

If you are familiar with SQL databases, and are already comfortable developing multiuser client/server applications, you will be comfortable working with query components because they most closely emulate the SQL paradigm you already know. You can easily create both static and dynamic queries. *TQuery* also provides a powerful feature not commonly found in server-based databases: the ability to create heterogeneous queries against more than one server or table type.

Accessing data through a remote server's stored procedure

The TStoredProc component encapsulates a stored procedure in a database on a remote server. A *stored procedure* is a set of semi-procedural statements, stored as part of a remote server's database metadata, that performs a frequently repeated database-related task on the server and passes results to a client application, such as an C++Builder database application. Most stored procedures also accept input parameters. The *TStoredProc* component enables C++Builder database applications to take advantage of stored procedures on remote servers.

If you are familiar with relational databases on remote servers, you already know about stored procedures, and will be able to use stored procedure components in your C++Builder applications.

If you are new to relational databases on remote servers, you should learn about the stored procedures your relational database management system provides on your remote server. Powerful and reusable, stored procedures can help you split database processing between your client application on the desktop and the remote server. See your server's database documentation for more information about its support for stored procedures.

Placing datasets in data modules

Before you can use datasets in your application, you must place them from the Data Access tab of the Component palette into your application.

▪Data Access tab of the Component palette
You can place dataset components in a data module or on a form. Placing datasets in a data module enables you to standardize all database access in your application from a central location. It also makes it easier to maintain database access. For more information about data modules, see "Using data modules."

Note: Placing datasets (and data sources) directly on forms is recommended only for very simple database applications. Even for moderately simple database applications that only work with a few tables or queries on a few forms, using a data module eases development and maintenance.

Opening and closing datasets

To read or write data in a table or through a query, an application must first open a dataset. You can open a dataset in two ways:

- Set the *Active* property of the dataset to *true*, either at design time in the Object Inspector, or in code at runtime:

```
CustTable->Active = true;
```

- Call the *Open* method for the dataset at runtime:

```
CustQuery->Open();
```

You can also close a dataset in two ways:

- Set the *Active* property of the dataset to *false*, either at design time in the Object Inspector, or in code at runtime:

```
CustQuery->Active = false;
```

- Call the *Close* method for the dataset at runtime:

```
CustTable->Close();
```

You need to close a dataset when you want to change any of its properties that affect the query, such as the *DataSource* property. At runtime, you may also want to close a dataset for other reasons specific to your application.

Determining and setting dataset states

The *state*--or *mode*--of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. A dataset is always in one state or another. At runtime, you can examine a dataset's read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Value	State	Meaning
dsInactive	Inactive	Dataset closed. Its data is unavailable.
dsBrowse	Browse	Dataset open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	Dataset open. The current row can be modified.
dsInsert	Insert	Dataset open. A new row can be inserted.
dsSetKey	SetKey	*TTable* only. Dataset open. Enables searching for rows based on indexed fields, or indicates that a *SetRange* operation is under way. A restricted set of data can be viewed, and no data can be changed.
dsCalcFields	CalcFields	Dataset open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsUpdateNew	UpdateNew	Dataset open. Indicates that the new record buffer should be modified.
dsUpdateOld	UpdateOld	Dataset open. Indicates that the old record buffer should be modified.
dsFilter	Filter	Dataset open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed.

When an application opens a dataset, C++Builder automatically puts the dataset into *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- Code in your application, or
- C++Builder's built-in behavior.

To put a dataset into *dsBrowse*, *dsEdit*, *dsInsert*, or *dsSetKey* states, call the method corresponding to the name of the state. For example, the following code puts *CustTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the database:

```
CustTable->Insert(); // Your application explicitly sets dataset state to Insert
AddressPromptDialog->ShowModal();
if (AddressPromptDialog->ModalResult == IDOK)
 CustTable->Post(); // C++Builder sets dataset state to Browse on successful completion
else
 CustTable->Cancel(); // C++Builder sets dataset state to Browse on cancel
```

This example also illustrates how C++Builder automatically sets the state of a dataset to *dsBrowse* when

- The *Post* method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into the *dsInactive* state, set its *Active* property to *false*, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable->Active = false;
CustTable->Close();
```

The remaining states (*dsCalcFields*, *dsUpdateNew*, *dsUpdateOld*, and *dsFilter*) cannot be set by your application. Instead, C++Builder sets them as necessary. For example, *dsCalcFields* is set when a dataset's *OnCalcFields* event is called. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Note: Whenever a dataset's state changes, the *OnStateChange event* is called for any data source components associated with the dataset.

The following sections provide overviews of each state, how and when states are set, how states relate to one another, and where to go for related information, if applicable.

Understanding the dataset's Inactive state

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time a dataset is closed until you set its *Active* property to *true*. At runtime, a dataset is closed until it is opened either by calling the *Open* method, or by setting the *Active* property to *true*.

When you open an inactive dataset, C++Builder automatically puts it into *dsBrowse* state. The following diagram illustrates the relationship between these states and the methods that set them.

Relationship of Inactive and Browse states

To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* mode when an application calls *Close*, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
void __fastcall CustForm::CustTableBeforeClose(TDataSet *DataSet)
{
 bool posted;
 if ((CustTable->State == dsEdit) || (CustTable->State == dsInsert))
   if (::MessageBox(0, "Post changes before closing?", "", MB_YESNO) == IDYES)
     {
       CustTable->Post();
       posted = true;
     }
   else
     {
       CustTable->Cancel();
       posted = false;
     }
}
```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

1. Select the table in the data module (or form).
2. Click the Events page in the Object Inspector.
3. Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Understanding the dataset's Browse state

When an application opens a dataset, C++Builder automatically puts the dataset into *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset.

From *dsBrowse* you can set all other dataset states. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (note that other factors and dataset properties such as *CanModify*, may prevent this change). Calling *SetKey* to search for records puts a dataset in *dsSetKey* mode. For more information about inserting and appending records in a dataset, see "Modifying data".

You can use two methods associated with all datasets to return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

The following figure illustrates the relationship of *dsBrowse* both to the other dataset modes you can set in your applications, and the methods that set those modes.

Relationship of Browse to other dataset states

Understanding the dataset's Edit state

A dataset must be in *dsEdit* mode before an application can modify records. In your code, you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the database underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *false* (the default),
- The *AutoEdit* property of the data source for the control is *true*, and
- *CanModify* is *true* for the dataset.

Important:

For *TTable* components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

Data-aware controls, for which editing is enabled automatically, call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

Understanding the dataset's Insert state

A dataset must be in *dsInsert* mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *false* (the default),
- The *AutoEdit* property of the data source for the control is *true*, and
- *CanModify* is *true* for the dataset.

Important:

For *TTable* components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note: Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls, for which inserting is enabled automatically, call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

Understanding the dataset's SetKey state

dsSetKey mode applies only to *TTable* components. To search for records in a *TTable* dataset, the dataset must be in *dsSetKey* mode. You put a dataset into *dsSetKey* mode with the *SetKey* method at runtime. The *GotoKey* method, which carries out the actual search, returns the dataset to *dsBrowse* state upon completion of the search.

Note: Other search methods, including *FindKey* and *FindNearest,* automatically put a dataset into *dsSetKey* state during a search, and return the dataset to *dsBrowse* state upon completion of the search.

Understanding the dataset's CalcFields state

C++Builder puts a dataset into *dsCalcFields* mode whenever an application calls the dataset's *OnCalcFields* event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler modifies itself. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see "Creating and using fields."

Understanding the dataset's Update states

The *dsUpdateNew* and *dsUpdateOld* modes are used to determine which record buffer to modify. C++Builder has two delayed update buffers, an old and a new, which are maintained by *TDataSet* objects. This flag tells other dataset functions which buffer to use.

Understanding the dataset's Filter state

C++Builder puts a dataset into *dsFilter* mode whenever an application calls the dataset's *OnFilterRecord* event handler or the dataset is being filtered with the *Filtered* property. This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated.

When the *OnFilterRecord* handler finishes or the dataset has been filtered with the *Filtered* property, the dataset is returned to *dsBrowse* state.

Navigating within datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose values can be manipulated by edit, insert, and delete methods, and the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Method	Description
First	Moves the cursor to the first row in a dataset.
Last	Moves the cursor to the last row in a dataset.
Next	Moves the cursor to the next row in a dataset.
Prior	Moves the cursor to the previous row in a dataset.
MoveBy	Moves the cursor a specified number of rows forward or back in a dataset.

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime.

In addition to these methods, two Boolean properties of datasets provide useful information when iterating through the records in a dataset:

Property	Description
Bof (Beginning-of-file)	*true*: the cursor is at the first row in the dataset.*false*: the cursor is not known to be at the first row in the dataset.
Eof (End-of-file)	*true*: the cursor is at the last row in the dataset.*false*: the cursor is not known to be at the last row in the dataset.

Moving to the first or last record (First, Last)

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to *true*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable->First();
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to *true*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable->Last();
```

Note: While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you should enable your users to navigate from record to record using the TDBNavigator component. The navigator component contains buttons that when active and visible enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons call the *First* and *Last* methods of the dataset.

Moving to the next or previous record (Next, Prior)

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to *false* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable->Next();
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to *false* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable->Prior();
```

Moving relative to the current record (MoveBy)

MoveBy enables you to specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable->MoveBy(-2);
```

Note: If you use *MoveBy* in your application and you work in a multiuser database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back because several users may be simultaneously accessing the database and changing its data.

Trapping beginning- and end-of-file conditions (Eof, Bof)

Two read-only, run-time properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful for controlling dataset navigation, particularly when you want to iterate through all records in a dataset.

Eof

When *Eof* is *true*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *true* when an application:

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Eof is set to *false* in all other cases; you should assume *Eof* is *false* unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*), *Eof* is *false*. To iterate through the dataset a record at a time, create a loop that terminates when *Eof* is *true*. Inside the loop, call *Next* for each record in the dataset. *Eof* remains *false* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
try
{
 CustTable->DisableControls();
 while(!CustTable->Eof) //Cycle until Eof is true
 {
   // Process each record here
   // ...
   CustTable->Next(); // Eof false on success; Eof true when Next fails on last record
 }
}
catch(...)
 {
   CustTable->EnableControls();
 }
```

Tip:

This example also demonstrates how to disable and enable data-aware visual controls tied to a dataset. Disabling visual controls during dataset iteration speeds processing because C++Builder does not have to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the *catch* clause of a *try...catch* statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When *Bof* is *true*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *true* when an application:

- Opens a dataset.
- Calls a dataset's *First* method.
- Calls a dataset's *Prior* method, and the method fails (because the cursor is currently at the first row in the dataset).
- Calls *SetRange* on an empty range or dataset.

Bof is set to *false* in all other cases; you should assume *Bof* is *false* unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
 try
 {
   CustTable->DisableControls(); // Speed up processing, prevent screen flicker
   while(!CustTable->Bof) // Cycle until Bof is true
   {
     // Process record here
     ...
     CustTable->Prior();  // Bof false on success; Bof true when Prior fails on first
                          // record
   }
 }
catch(...)
 {
   CustTable->EnableControls(); // Display new current row in controls
 }
```

Marking and returning to records (bookmarks)

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired.

C++Builder provides three bookmark methods that let you flag a record in a dataset so that you can return to it later.

Method	Description
GetBookmark	Allocates a bookmark for your current position in the dataset.
GotoBookmark	Returns to a bookmark previously created by *GetBookmark.*
FreeBookmark	Frees a bookmark previously allocated by *GetBookmark*.

To create a bookmark:

1. Declare a variable of type *TBookmark* in your application.
2. Call *GetBookmark* to allocate storage for the variable.
3. Set the value of *GetBookmark* to a particular location in a dataset. The *TBookmark* variable is a pointer.

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark.

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

The following code illustrates the use of bookmarking:

```
void DoSomething (const TTable *Tbl)
{
 TBookmark Bookmark;

 Bookmark = Tbl->GetBookmark(); //Allocate memory and assign a value
 Tbl->DisableControls(); // Turn off display of records in data controls
 try
 {
   Tbl->First(); // Go to first record in table
   while (!Tbl->Eof) // Iterate through each record in table
   {
     // Do your processing here
     ...
     Tbl->Next();
   }
 }
 catch(...)
 {
   Tbl->GotoBookmark(Bookmark);
   Tbl->EnableControls(); // Turn on display of records in data controls, if necessary
   Tbl->FreeBookmark(Bookmark); // Deallocate memory for the bookmark
 }
}
```

Before entering the record iteration process controls are disabled. Should an error occur during iteration through records, the *catch* clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

Filtering datasets

Filtering lets you specify criteria to temporarily restrict the data being viewed. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find records that contain orders in excess of $2,000.00. C++Builder supports filtering of a table or query to handle both of these requirements.

Filters are similar to, though less powerful than, queries, with the benefit that filters work on the dataset itself, meaning that the result is always "live" (unlike queries which sometimes produce result sets that can't be modified).

You can filter a dataset in three ways:

- Setting the *Filter* property of the dataset.
- Restricting record visibility at the time of record retrieval using an *OnFilterRecord* event handler.
- Finding a record in a dataset that matches search values using the *Locate* method for the dataset.

If you use Locate, C++Builder automatically generates a filter for you at runtime if it needs to

This section discusses properties and methods common to the first two methods of filtering datasets as well as information specific to each method.

Note: Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

Turning filters on and off

The simplest and most common way to use filtering is to turn on filtering for the entire dataset. To turn on filtering a dataset, set the *Filtered* property to *true*. With filtering on, the dataset generates an event for each record in the dataset. In handling that event, you can determine whether to accept or "filter out" each record. You could also supply a filter condition as the value of the *Filter* property instead of writing an event handler.

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions which can make the current record disappear if it no longer passes the filter. When this occurs, the next record that passes the filter condition becomes the current record.

You can also filter records programmatically by creating a standard control (such as an edit box). The following code example illustrates this.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 Table1->Filtered = true;
 Table1->Filter = Edit1->Text;
 Table1->Refresh;
}
```

Some other filter examples are:

```
CustTable->Filter = "State = California";
Table1->Filter = "PatientAge >= 18";
```

Turning filters on and off at runtime

At runtime you can turn filtering off by setting the *Filtered* property to *false*, and you can turn it on by setting the *Filtered* property to *true*. If you turn off filtering, all records in a dataset are available to your application, but for the application to see all records, you must also call the dataset's *Refresh* method. For example

```
{
 CustTable->Filtered = false;
 CustTable->Refresh();
}
```

If you turn on filtering at runtime, you must also call *Refresh* again to make the filter take effect. The current record may no longer pass the filter condition, and may disappear. If that happens, the next record that passes the filter condition becomes the current record.

Fine-tuning the filter

The *FilterOptions* property allows you to fine-tune the filtering provided by the *Filter* property. Setting *foCaseInsensitive* to *true* allows the filter to be processed without regard to case in the dataset's data. Setting *NoPartialCompare* to *true* requires string matches to be exact over the length of the data in the dataset, partial matches are not allowed. Both *FilterOptions* are *false* by default.

FilterOptions is a set property and would be turned on at runtime with the following code:

```
CustTable->FilterOptions=CustTable->FilterOptions << foCaseInsensitive << NoPartialCompare;
```

or

```
CustTable->FilterOptions=CustTable->FilterOptions << foCaseInsensitive;
```

Locating records in a filtered dataset

You can use *Find* methods in conjunction with filters to go to the first, last, next, and previous records based on filter criteria when the dataset is not currently being filtered. When filtering is not enabled, the methods temporarily set the *Filtered* property to *true* then use the code in either the *OnFilterRecord* event handler or the condition in the *Filter* property to find the record. To locate records that match the filter, call the *FindFirst*, *FindLast*, *FindNext*, or *FindPrior* methods. A typical use for these methods is to add a button to your form with the code:

```
CustTable->FindNext();
```

All four methods are Boolean functions that return *true* if they locate the matching record. For example, if you call *FindFirst*, it iterates through the dataset, generating *OnFilterRecord* events until a record passes the *FindFirst* filter. If it finds such a record, it makes that the current record and returns *true*. Otherwise, the current record stays the same and *FindFirst* returns *false*.

Filtering datasets with the OnFilterRecord event

The OnFilterRecord event occurs when filtering is active, either from setting *Filtered* to *true*, or by using the *FindFirst*, *FindLast*, *FindNext*, or *FindPrior* methods. *OnFilterRecord* event handlers have two parameters: the dataset being filtered and a boolean *Accept* parameter to specify whether the record should be included. By default, *Accept* is *true*; only when you set *Accept* to *false* will a record be excluded.

When *Filtered* is *true*, the dataset generates an *OnFilterRecord* event for each record it retrieves. The event handler for the *OnFilterRecord* tests each record and only those that meet the Filter's conditions are visible in the application. As performance is directly related to how often the event fires and how long it takes to process each event, the code in the *OnFilterRecord* event handler should be kept short.

The *TFilterRecordEvent* type points to a method that responds to an *OnFilterRecord* event, for including or excluding records from being visible in a dataset.

This method of filtering is most valuable when you want to compare a value in the table to something outside of the table. For example, you could compare a value in the table with a value in an edit box or other visual component to filter "on the fly."

To restrict visible records in a dataset to a subset of those that match a certain set of criteria:

- Write an *OnFilterRecord* event handler for the dataset.
- Set the *Filtered* property for the dataset to *true*.

Writing a filter event handler

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your filter handler must set an *Accept* parameter to *true* to include a record, or *false* to exclude it. For example, the following filter displays only those records with the *State* field set to CA:

```
void __fastcall TForm1::CustTableFilterRecord(TDataSet *DataSet, Boolean &Accept)
{
 Accept = (String("CA") == DataSet->FieldByName("State")->AsString);
}
```

Switching filter event handlers at runtime

You can change how records are filtered at runtime by assigning a different handler
to the *OnFilterRecord* event for a dataset. To apply the new filter to the dataset after changing it, call the dataset's *Refresh* method:

```
{
 CustTable->OnFilterRecord = Query2FilterRecord;
 Refresh();
}
```

Specifying a filter

You can filter dataset components to restrict which records you want to see in the dataset by setting these properties:

- Set the value of the *Filter* property for the dataset.
- Set the *Filtered* property for the dataset to *true*.
- Fine-tune the filter by modifying the *FilterOptions* property.

This method of filtering datasets is most valuable when the filter will be set at run-time. You cannot compare the value in the *Filter* property to a value outside of the table, for example, it cannot be compared to a value in an edit box. To change the value of a filter "on the fly," see "Filtering datasets with the OnFilterRecord event" .

The *Filter* property is a string that lets you set conditions on one or more fields of your dataset. You can compare fields to literal values and constants using the comparison operators in the following table and the AND, NOT, and OR operators to combine comparisons. You can also compare a field to a field in an edit control. You must enclose field names that contain spaces within square brackets.

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal

The following are some examples that can be used to set filtering conditions:

- PatientAge >= 18
- State = 'FL'
- (PatientAge >= 18) AND (Balance > 0)
- (Temperature < 212) AND (NOT Windy)

In the previous example, Windy is a logical field.

- (SalePrice < $10,000) OR (Terms > 30)
- [Sale Date] > 'January 1, 1996'.

Searching datasets

You can search a dataset for specific records using the generic search methods *Locate* and *Lookup* or the *Find* methods *FindFirst*, *FindLast*, *FindNext*, or *FindPrior*. These methods enable you to search on any type of columns in any dataset.

- *Locate* moves the cursor to the first row matching a specified set of criteria.
- *Lookup* returns the values from the first row that matches specified search criteria, but does not move the cursor to that row.
- Find methods are used with filters and move the cursor to the first row matching the search criteria when filtering is not enabled

Moving the cursor to the located record (Locate)

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":

```
{
 bool LocateSuccess;
 TLocateOptions SearchOptions;

 SearchOptions << loPartialKey;
 LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
    SearchOptions);
}
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *true* if it finds a matching record, *false* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search multiple columns and specify multiple values to search for. Search values are variants; this enables you to specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the variant array on the fly using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
{
 Variant tmp(OPENARRAY(int, (0,1)), vtInteger);
 tmp << (int)("Sight Diver");
 tmp << (int)("P");
 VarArrayOf(&tmp,1);
}
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index. Otherwise, *Locate* creates a BDE filter for the search.

Returning values from a located record (Lookup)

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of the column to search, the field value to match, and the field or fields to return. For example, the following code looks for the first row in the *CustTable* where the value in the *Company* column is Professional Divers, Ltd., and returns the company name, a contact person, and a phone number for the company:

```
{
 Variant LookupResults;
 LookupResults = CustTable->Lookup("Company", "Professional Divers, Ltd.", "Company;
    Contact; Phone");
}
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as variants. If more than one return value is requested, *Lookup* returns a variant array. If there are no matching records, *Lookup* returns a null variant. For more information about variant arrays, see the C++Builder *Programmer's Guide*.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual string items with semi-colons.

Because search values are variants, if you pass multiple values, you must pass a variant array type as an argument (for example, the return values from the *Lookup* method. The following code illustrates a lookup search on multiple columns:

```
{
 Variant tmp(OPENARRAY(int, (0, 1)), vtInteger);
 tmp << (int)("Sight Diver", 0);
 tmp << ("Christiansted", 1);
 Variant LookupResults;
 LookupResults = CustTable->Lookup("Company;City", tmp, "Company;Addr1;Addr2;State;Zip");
}
```

Lookup uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index. Otherwise, *Lookup* creates a BDE filter for the search.

Modifying data

You can use dataset methods to enable an application to insert, update, and delete data:

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code, you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the table(s) underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if:

- The control's *ReadOnly* property is *false* (the default),
- The *AutoEdit* property of the data source for the control is *true*, and
- *CanModify* is *true* for the dataset.

Important:

For *TTable* components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the record that appears in any data-aware controls on a form. Data-aware controls, for which editing is enabled automatically, call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you provide a navigator component on your forms, users can cancel edits by clicking the navigator Cancel button. Canceling edits returns a dataset to
dsBrowse state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, Edit and Post are often used together. For example,

```
{
 CustTable->Edit();
 CustTable->FieldbyName("CustNo")->AsInteger = 1234;
 CustTable->Post();
}
```

The first line of the code fragment places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record to the database.

Note: If the *CachedUpdates* property for a dataset is *true*, posted modifications are written to a temporary buffer. To write cached edits to the database, call the *ApplyUpdates* method for the dataset.

Editing entire records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit Post.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. Application must perform an explicit *Post*.

These methods take a comma-delimited array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or null. If the number of values in an argument is less than the number of columns in a dataset, the remaining values are assumed to be null.

For datasets that are not indexed, AppendRecord adds a record to the end of the dataset and InsertRecord inserts a record after the current cursor position. For indexed tables, both methods places the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a Post.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass null values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns null values to them. Null values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a *TTable* component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
{
 TVarRec* Item1 = new TVarRec("Japan");
 TVarRec* Item2 = new TVarRec("Tokyo");
 TVarRec* Item3 = new TVarRec("Asia");

CountryTable->InsertRecord((Item1,Item2, Item3), 3);
 delete Item1;
 delete Item2;
 delete Item3;
}
```

This statement does not specify values for Area and Population, so null values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan."

To update the record, an application could use the following code:

```
{
 TLocateOptions tlo;
 tlo << loCaseInsensitive;
 if (CountryTable->Locate("Name", "Japan", tlo))
 {
   CountryTable->Edit();
   TVarRec *tvr1 = new TVarRec(344567);
   TvarRec *tvr2 = new TVarRec(164700000);
NULL, tvr1, tvr2), 5);
   CountryTable->Post();
   delete tvr1;
   delete tvr2;
 }
}
```

This code assigns values to the Area and Population fields and then posts them to the database. The three null values act as place holders for the first three columns to preserve their current contents.

Adding new records

A dataset must be in *dsInsert* mode before an application can add new records to an existing table. In your code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *false* (the default), and
- *CanModify* is *true* for the dataset.

Important:

For *TTable* components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note: Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls, for which inserting is enabled automatically, call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise, you must call *Post* in your code.

Post writes the new record to the database, or, if cached updates are enabled, *Post* writes the record to a buffer. To write cached inserts and appends to the database, call the *ApplyUpdates* method for the dataset.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when cached updating is enabled), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one. Field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when cached updating is enabled), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is added to the end of the dataset.
- For SQL databases the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting records

A dataset must be active before an application can delete records. Delete deletes the current record from a dataset and puts the dataset in *dsBrowse* mode. The record that followed the deleted record becomes the current record. If cached updates are enabled for a dataset, a deleted record is only removed from the temporary cache buffer until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting data changes

The Post method is central to an C++Builder application's interaction with a database table. Post writes changes to the current record to the database, but it behaves differently depending on a dataset's state.

- In *dsEdit* state, Post writes a modified record to the database (or buffer if cached updates is enabled).
- In *dsInsert* state, Post writes a new record to the database (or buffer if cached updates is enabled).
- In *dsSetKey* state, Post returns the dataset to *dsBrowse* state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, C++Builder calls Post implicitly. Calls to the First, Next, Prior, and Last methods perform a Post if the table is in *dsEdit* or *dsInsert* modes. The Append and Insert methods also implicitly post any pending data.

Note: Post is not called implicitly by the Close method. Use the BeforeClose event to post any pending edits explicitly.

Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called Post. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record to its original values by calling the Cancel method for the dataset. A call to Cancel always returns a dataset to *dsBrowse* state.

On forms, you can allow users to cancel edit, insert, or append operations by including a Cancel button on a navigator component associated with the dataset, or you can code your own Cancel button on the form.

Accessing text files

The ASCIIDRV text driver allows BDE clients to access text files. The text driver allows BDE clients to access text data directly without first importing into a table format. By using this driver, the application developer can build a more efficient import/export utility. Filters can be set on the cursors that are opened on the text files to import/export only those records that satisfy the filter's criteria.

When you open a text table, you can provide the field descriptor information by calling the BDE function *DbiSetFieldMap* to set a field map or you can bind external schema to text tables.

A text file can be created by using the BDE function *DbiCreateTable*. The developer supplies only the table name and driver type values in the *CRTblDesc* descriptor; the rest of the field values are ignored. *DbiCreateTable* creates a file with the given name; no field descriptions are necessary.

The BDE function *DbiOpenTable* can be used to open a text file for import/export. The file can be opened as a delimited text file or as a fixed length text file.

For more information on accessing text files, see the BDE online help file BDE32.HLP and search for ASCIIDRV.

Using dataset events

Datasets have a number of events that enable an application to perform validation, compute totals, and perform other tasks. The events are listed in the following table.

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

For more information about events for TDataSet components, see the *VCL Reference*.

Aborting a method

To abort a method, such as an Open or Insert, call the *Abort* procedure in any of the *Before* methods (BeforeOpen, BeforeInsert, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

```
void TForm1::TableBeforeDelete (TDataset *Dataset)
{
 if MessageBox(0, "Delete This Record?", "CONFIRM", MB_YESNO) != IDYES)
   Abort();
}
```

Setting values of calculated fields with OnCalcFields

The OnCalcFields event is used to set the values of calculated fields. The AutoCalcFields property determines when OnCalcFields is called. If AutoCalcFields is *true*, then OnCalcFields is called when

- A dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of AutoCalcFields.

Caution: OnCalcFields is called frequently, so the code you write for it should be kept short. Also, if AutoCalcFields is *true*, OnCalcFields should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if OnCalcFields performs a Post, and AutoCalcFields is *true*, then OnCalcFields is called again, leading to another Post, and so on.

If AutoCalcFields is *false*, OnCalcFields is only called when Post (or any method that implicitly calls Post, such as Append or Insert) is called.

When OnCalcFields executes, a dataset is in dsCalcFields mode, so you cannot set the values of any fields other than calculated fields. After OnCalcFields is completed, the dataset returns to *dsBrowse* state.

Using the C++Builder database development model

This topic introduces the C++Builder tools and features you use to build PC LAN-based and remote client/server database applications. It also explains a tiered approach to application development using C++Builder's tools and features and your databases of choice.

C++Builder database tools and features

The following table lists many C++Builder database tools and features, briefly describes them, and indicates which editions of C++Builder contain them:

Tool/Feature	Description	Client/Server	Professional	Standard
SQL Explorer Database Explorer	Integrated tool for browsing databases, managing BDE aliases, and creating data dictionaries. Client/Server is SQL-enabled.	3	3	3
SQL Monitor	Integrated tool for tracing and examining SQL query performance.	3		
Visual Query Builder	Integrated tool for visual building of SQL queries.	3		
Data Pump Expert	Standalone tool for moving metadata and data between databases.	3		
Data Dictionary	Stores extended field attributes apart from databases and application code; shares attributes across fields, datasets, and applications.	3	3	
Object Repository	Stores data modules and forms for sharing among applications.	3	3	3
Data Modules	Nonvisual component containers for centralized data-access across forms and applications.	3	3	3
Data-access components	Nonvisual components for encapsulating database connections.	3	3	3
Data-aware controls	Visual components for providing a user-interface to data.	3	3	3
Database Desktop	(DBD) Tool for browsing, creating, and changing desktop databases.	3	3	3
Borland Database Engine	(BDE) Borland's core database engine and connectivity software.	3	3	3
SQL Links	SQL drivers for Sybase, SQL Server, Oracle, and InterBase.	3	3 3	3
ODBC Socket	BDE support socket for third-party ODBC drivers.	3		
BDE API	BDE API and online help files.			
Quick Reports	C++Builder components for creating pre-defined reports in an application.	3	3	3
InterBase NT	Borland InterBase Workgroup Server for NT, two-user license.	3		
Local InterBase Server	32-bit Local InterBase Server.	3	3	

The database features and tools in C++Builder form a tightly-integrated suite designed to ease the design, implementation, deployment, and maintenance of database applications in a variety of computing environments.

Some C++Builder database features and tools, such as the SQL Explorer, and SQL Monitor, are specific to the C++Builder Client/Server edition, but provide corollary, non-SQL enabled tools, such as the Database Explorer for C++Builder Professional. Other tools, such as database components, the Borland Database Engine (BDE), and the stand-alone Database Desktop (DBD) are provided in all three editions of C++Builder to enable database applications development.

In general, the C++Builder Client/Server edition provides the tightest integration of tools and the power necessary for building, testing, and deploying client applications that work with SQL databases on remote servers, and both local and networked Paradox® and dBASE® databases. With third-party ODBC drivers you can also access local and remote ODBC-compliant databases.

C++Builder Professional enables you to build and test client applications for local and networked Paradox and dBASE databases. With third-party ODBC drivers you can also access local and remote ODBC-compliant databases.

C++Builder Standard enables building and testing of client applications for local and networked Paradox and dBASE databases.

The following figure illustrates how C++Builder's tools work together in the Client/Server edition.

C++Builder Client/Server development environment

To learn how to make the most effective use of C++Builder's database features and tools, see "Understanding the C++Builder development model".

Even if you have C++Builder Professional or C++Builder Standard, the investment you make in building database applications is protected when you upgrade to C++Builder Client/Server. C++Builder simplifies database application development so that you can migrate to a larger, enterprise-wide, client application/database server implementation as the need arises.

What is a C++Builder database application?

All C++Builder database applications are database clients. A *client* requests information from and sends information to a database server. A server processes requests from many clients simultaneously, coordinating access to and updating of data.

C++Builder clients actually communicate with the BDE, which in turn communicates with the database server of your choice. The following figure illustrates this relationship.

The relationship of a C++Builder client to a database server

All C++Builder database applications are also user-interfaces to a database. A well-designed C++Builder application makes it easy for a user to view and edit data, even if the application works with a complex data model. As an applications developer, you must understand the complexities of your data models even as you work to mask them from the eventual end-users of your application.

Understanding database components

C++Builder provides nonvisual *data-access components* that encapsulate your client's communication with a database. Data-access components only deal with database connectivity, which enables you to focus your attention on your application's data needs without worrying about user interface. Typically these components are placed in a data module container within an application.

C++Builder also provides *data-aware visual components*, called *visual controls*, that encapsulate user interaction with your application's data sources. You design a user interface with these controls by placing these controls on the forms in your application. Each control is linked to one or more fields or records, and controls how a user sees a field or record in your application.

You link visual controls to data-access components through a data source component. A data-source component acts as a conduit between an application's low-level data-access interactions and the high-level view of data its users are provided.

In the C++Builder IDE, the Component palette provides two database component pages:

- Data Access contains nonvisual data-access components.
- *Data Controls* contains data-aware visual controls.

The following table summarizes the components that appear on the Data Access page and where in this guide you can get more information about them:

Component	Purpose
TDataSource	Acts as a conduit between other data access components and data-aware visual controls. For more information about *TDataSource*, see "Linking visual controls to datasets."
TTable	Represents a dataset that retrieves all columns and records from a database table. For more information about *TTable*, see "Working with tables."
TQuery	Represents a dataset that retrieves a subset of columns and records from one or more local or remote database tables based on an SQL query. For more information about *TQuery*, see "Working with queries."
TStoredProc	Represents a dataset that retrieves one or more records from a database table based on a stored procedure defined for a database server. For more information about *TStoredProc*, see "Working with stored procedures."
TDatabase	Encapsulates a client/server connection to a single database in one session. For more information about *TDatabase*, see "Connecting to databases."
TSession	Represents a single session in a multi-threaded database application. Each session can have multiple database connections as long as each thread associated with a particular database has its own session. For more information about *TSession*, see "Connecting to databases."
TBatchMove	Encapsulates a dataset used to move data from one table to another. For more information about *TBatchMove*, see "Handling batch move operations."
TUpdateSQL	Represents SQL INSERT, UPDATE, and DELETE statements that can be used to update the read-only result sets of some queries. For more information about *TUpdateSQL*, see "Caching updates."

The following table summarizes the data-aware visual controls that appear on the Data Controls page:

Component	Purpose
TDBGrid	Display and edit dataset records in tabular format.
TDBNavigator	Cursor through dataset records; enable Edit and Insert states; post new or modified records; cancel edit mode; refresh data display.
TDBText	Display a field as a label.

TDBEdit	Display and edit a field in an edit box.
TDBMemo	Display and edit multi-line or blob text in a scrollable, multi-line edit box.
TDBImage	Display and edit a graphics image or binary blob data.
TDBListBox	Display a list of choices for entry in a field.
TDBComboBox	Display an edit box and drop-down list of choices for edit and entry in a field.
TDBCheckBox	Display and set a Boolean field condition in a check box.
TDBRadioGroup	Display and set exclusive choices for a field in a radio button group.
TDBLookupListBox	Display a list of choices derived from a field in another dataset for entry into a field.
TDBLookupComboBox	Display an edit box and drop-down list of choices derived from a field in another dataset for edit and entry in a field.
TDBCtrlGrid	(Client/Server and Professional only). Display and edit records in a tabular grid, where each cell in the grid contains a repeating set of data-aware components and one record.

The following figure illustrates how data-access components and visual controls relate to data, to one another, and to the user interface in a C++Builder database application:

Relationship of database components in an application

As this figure illustrates, a database client application usually consists of one or more data modules and forms.

A data module contains the application's data access components. These include *TTable*, *TQuery*, and sometimes *TStoredProc* dataset components that encapsulate the fields and records in one or more database tables and communicate with the BDE. They also include several *TDataSource* components that act as a conduit between datasets and data-aware controls on the applications' forms.

Optionally, data modules may contain one or more explicitly declared *TDatabase* components that encapsulate one or more datasets' connections to a database. (If an application does not explicitly declare *TDatabase* components, C++Builder creates and manages temporary ones as needed at run time.) All applications have a default *TSession* component that C++Builder creates and manages, too. A *TSession* component encapsulates a database thread. Multi-threaded applications must explicitly declare additional *TSession* components.

Forms contain an application's data-aware visual controls. Visual controls are the user interface to databases in C++Builder. Most commonly used visual controls include *TDBGrid*, *TDBNavigator*, and *TDBEdit*.

The C++Builder IDE, its supporting tools, the object-oriented data-access components that encapsulate database connections, and the data-aware components that comprise the database user interface make it easy to develop complex and usable database applications. The following sections describe a C++Builder development model designed to help you use C++Builder as effectively as possible.

Press the >> button to read through topics in sequence.

Building database forms

Most C++Builder database applications require that you use at least three database components. This is the minimum number of components you need:

- One dataset component
  Dataset components, such as *TTable* and *TQuery*, communicate with the Borland Database Engine. Data received from a component is sent to the database through the BDE.
- One data-aware control
  Data-aware controls provide the user interface to the data in the dataset. Users manipulate data-aware controls to browse, edit, or enter data.
- One *TDataSource* component
  A *TDataSource* component acts as a conduit between a dataset component and a data-aware control. It links the data-aware control with the dataset component, and therefore, the data in the database. Without a *TDataSource* component, data-aware controls cannot access the data in a database.

Each dataset can be linked through one or more *TDataSource* components to multiple data-aware controls. For example, a *TDataSource* component could link a dataset referenced with a *TTable* component to a data-aware check box, a data-aware list box, and a data-aware image control on a form. Each of these controls would display a single field of the dataset. If the *TDataSource* linked a data grid to the dataset, the data grid could display all or several of the fields in the dataset.

To build a database form, you place the data-aware controls you want on the form and arrange them as you wish. While you can also place data-access components on the form, you should add them to a data module instead

Making the connections: linking database components

This section describes how to connect database components. This allows you to create a single-table database application without writing a line of code. Follow the steps below to link a *TTable* component, a *TDataSource* component, and a data-aware component together:

1. Begin a new application with a blank form.
2. Choose File|New Data Module to add a data module to the project.
3. Place a *TTable* component and a *TDataSource* component in the data module.
4. Set the *DatabaseName* property of *TTable* to the name of the database you want to access.

▪ While you are learning the technique of linking database components, try selecting the BCDEMOS alias that appears in the list of database names available to *TTable*. BCDEMOS identifies a set of database tables that were installed as part of C++Builder.

5. Set the *TableName* property of *TTable* to the name of the table in the database you want to access.

▪ You will see a list of tables from which you can choose.

6. Set the *Dataset* property of the *TDataSource* component to the name of the *TTable* component by selecting it from the list of available dataset components.

▪ If you have only one data source component on the form, you will have only one choice.

7. On the form, place any data-aware control except the database navigators.
8. Make the data module available to the form unit by choosing File|Include Unit Hdr and selecting the unit name of the data module that holds the *TTable* and *TDataSource* components.

▪ Unless you've already given the data module a name, the unit name is likely to be *Unit2*.

9. Set the *DataSource* property of the data-aware control to the name of the *TDataSource* component by selecting it from the list of available data source components.

▪ If you have only one datasource component in the data module, you will have only one choice.

10. If the data-aware control has a *DataField* property, select the field you want the control to access by selecting it from the list of available fields.

▪ All data-aware controls except the data grids and the database navigator have a *DataField* property.

11. To display the data in the data-aware control, return to the *TTable* component in the data module and set its *Active* property to *true*.

▪ Setting *Active* to *true* opens the table and the data displays in the data-aware control.

You might want to place one more data-aware control on your form. The database navigator provides an easy way to move through the data in the dataset.

To use a database navigator control,

1. Add the database navigator (*TDBNavigator*) to the form.
2. Set its *DataSource* property to the name of the *TDataSource* component that links to the dataset you want to access.

To run your application, choose Run|Run. Your database application compiles, links, and runs, and you can use the database navigator to move through the records in the dataset.

Understanding the C++Builder development model

The development of a client application for local or remote database access involves the following broadly-defined steps:

- Defining your application's tasks and database sources.
- Exploring existing database tables and features, and creating additional tables as needed to support your application's data requirements.
- Creating your application's data access model using data modules. When you define the datasets your application uses, you can use extended field attributes in the Data Dictionary. You can also copy, inherit, or use existing data modules from the Object Repository, and add your own data modules to the repository for later reuse.
- Creating your application's database user interface using forms. You can copy, inherit, or use existing forms from the Object Repository, and add your own forms to the repository for later reuse.

These steps are iterative in nature. For example, as you build your application, you may discover new data needs that require you to modify or add to your existing databases. Changes to the database mean changes to your application's data access and user interface.

Browsing and designing with the SQL and Database Explorers

In C++Builder Client/Server edition you browse databases and populate tables with data using the SQL Explorer from the IDE. In C++Builder Professional you use the Database Explorer. Both versions of the Explorer enable you to:

- Examine existing database tables and structures. The SQL Explorer enables you to examine remote SQL databases, and to query them.
- Populate tables with data.
- Create extended field attribute sets in a Data Dictionary for later retrieval and reuse. Extended field attributes describe how values in a field are formatted and displayed.
- Associate extended field attributes with fields in your application.
- Create and manage BDE aliases, used by your application to connect to databases.

To learn more about the SQL Explorer, and the Database Explorer see their respective online Help files.

Designing with the Database Desktop

In every version of C++Builder you can use the Database Desktop (DBD) to browse and modify existing tables or create and populate new ones, create indexes, define referential integrity, and create database-level validation and business rules for them. You can browse and create BDE aliases as well.

The DBD is a stand-alone utility that runs outside the C++Builder IDE. For more information about the DBD, see its online Help.

Using data modules

Data modules radically simplify data-access development in your applications. Data modules offer you a centralized design-time container for all your data access components that enables you to modularize your code and separate the database access logic and business rules in your applications from the user interface logic in the application's forms.

Once you define your datasets and their fields in a data module, all forms that use the module have consistent access to datasets and fields without requiring you to recreate them on every form each time you need them. In fact, data modules can and should be stored in the Object Repository for shared use among developers and applications.

Using the Data Dictionary

The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the Data Dictionary. Using the Data Dictionary also ensures a consistent data appearance within and across the applications you create.

In a client/server environment the Data Dictionary can reside on a remote server for additional sharing of information.

To learn more about creating a Data Dictionary and extended field attributes with the SQL and Database Explorers, see their respective online Help.

Designing a database interface with data-aware controls

By placing data-access components in data modules, you can develop forms in your database applications that are concerned only with a consistent user interface. With the extended functionality of TDBGrid controls, flexible *TDBLookupListBox* and *TDBLookupComboBox* controls, and a TDBCtrlGrid control that provides a multi-field, multi-record view of data, your application's user interface can be more compelling and effective.

By storing links to well-designed forms and data modules in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms and modules also makes it possible for you to develop corporate standards for database access and application interfaces.

Using the Object Repository

The Object Repository stores links to data modules, forms, and projects for reuse and reference. When you create a new application, you can:

- *Copy* an existing data module, form, or project, ensuring that your copy is completely independent of the repository.
- *Inherit* an existing data module, form, or project, ensuring that changes to the linked module, form, or project in the repository are replicated to your application when you recompile.
- *Use* an existing data module, form, or project, ensuring that changes you make to the module, form, or project are reflected upon the object in the repository and are available for use in other applications.

The Object Repository supports team development practices. It uses a referencing mechanism to data modules, forms, and projects where they exist on a network server or shared machine. Every developer in your organization can save objects to a shared location, and then set C++Builder's Object Repository reference to point to that location.

The Object Repository is also customizable through the Options|Repository dialog box in the IDE.

For general information about using the Object Repository for all your C++Builder programming, see the C++Builder *User's Guide*.

Connecting to databases

This topic describes the components that encapsulate database connectivity in a C++Builder application and it explains how to use them. In C++Builder, database connections are encapsulated by *TSession* and *TDatabase* components. The following figure highlights the relation of these components to all the data-access components in C++Builder.

C++Builder Data Access components hierarchy

Each time an application runs, C++Builder creates a default *TSession* component for it. A *session component* provides global control over all database connections in an application. If you do not create multi-threaded database applications, you need only concern yourself with the default session in your application.

A *multi-threaded database application* is a single application that attempts to run two or more simultaneous operations such as SQL queries, against the same database. If you create a multi-threaded database application, then each additional thread after the first requires its own session component. Multi-threaded applications need to manage sessions through a *TSessionList* component. A default session list component, called *Sessions*, is created for you whenever you start a database application. You never need to create your own session list component.

A *database component* encapsulates the connection to a single database within an application. If you do not need to control database connections (such as specifying a transaction isolation level), do not create database components. Temporary database components are created automatically at runtime when an application attempts to open a table for which there is not already a database component. But if you want to control the persistence of database connections, logins to a database server, property values of database aliases, or transactions, then you must create a database component for each desired connection.

Press the >> button to read through topics in sequence.

Using TSession components

Using TDatabase components

Using TSession components

The following table lists the properties of *TSession*, what they are used for, their default values if any, and whether they are available at design time:

Property	Purpose	Default	Design time access
Active	*true*, starts the BDE session. *false*, disconnects datasets and stops the session.	*false*	Yes
Databases	Specifies an array of all active databases in the session.		No
DatabaseCount	Provides an integer value specifying the number of currently active databases in the session.		No
KeepConnections	*true*, maintain database connection(s) even if there are no open datasets. *false*, close database connection(s) when there are no open datasets.	*true*	Yes
Name	Names a session component (for example, *Session1*).		Yes
NetFileDir	Specifies the directory path of the Paradox network control file, which enables sharing of Paradox tables on network drives.		Yes
PrivateDir	Specifies the path in which to store temporary files (for example, files used to process local SQL statements).		Yes
SessionName	Specifies the name of the session that must be used by database components to link themselves to a particular session.		Yes
Tag	Stores an integer value as part of a component. While the Tag property has no meaning to C++Builder, your application can use the property to store a value for its special needs		Yes

The following sections describe some of these properties and their uses in more detail.

Using the Active property

Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it.

If *Active* is *false* (the default), all databases and datasets associated with the session are closed. If *true*, databases and datasets are open.

For session components you place in a data module or form, setting *Active* to *false* when there are open databases or datasets closes them. At runtime, closing databases and datasets may invoke events associated with them.

Note: You cannot set *Active* to *false* for the default session at design time. While you can close the default session at runtime, it is not recommended.

For session components you create, use the Object Inspector to set *Active* to *false* at design time to disable all database access for a session with a single property change. You might want to do this if, during application design, you do not want to receive exceptions because a remote database is temporarily unavailable.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following code closes all open databases and datasets for the a session with a single line of code:

```
Session1->Close();
```

This code sets the Session1's *Active* property to *false*. When a session's *Active* property is *false*, any subsequent attempt by the application to open a database or dataset resets *Active* to *true* and calls the session's *OnStartup* event if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1->Open();
```

Using the KeepConnections property

KeepConnections provides the default value for the *KeepConnection* property of temporary database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *true* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *false*, a database connection is dropped as soon as all its datasets are closed.

Note: Connection persistence for database components you explicitly place in a data module or form is controlled by the database component's *KeepConnection* property. If they differ in setting, *KeepConnection* for a database component overrides the *KeepConnections* property of the session.

KeepConnections should always be set to *true* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or re-establishes a connection, it incurs the overhead of attaching and detaching the database.

Even when *KeepConnections* is *true*, you can close all inactive database connections at any time with the *DropConnections* method. For example, the following code drops inactive connections for the default session:

```
Session->DropConnections();
```

Using the NetFileDir property

NetFileDir specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server).

C++Builder derives a value for *NetFileDir* from the Borland Database Engine (BDE) Configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE Configuration setting, so be sure to validate the new value.

At design time you can specify a value for *NetFileDir* in the Object Inspector. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session->NetFileDir = ExtractFilePath(ParamStr(0));
```

Note: *NetFileDir* can only be changed when an application is not using any Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

Using the PrivateDir property

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements.

If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you may want to set *PrivateDir* to a local drive to prevent temporary files from being created on the server.

You can set or change *PrivateDir* in the Object Inspector at design time, or in code at runtime. The following code changes the setting of the default session's *PrivateDir* property to a user's C: drive:

```
Session->PrivateDir = "C:\\";
```

Using the SessionName property

SessionName is used to associate databases and datasets with a session. For the default session, *SessionName* is "Default". For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component, it is automatically associated with the default session. You can also assign the *SessionName* for a database or dataset component a name that corresponds to the *SessionName* of a session component you create. For example, the following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to "InterBaseSession", activates the session, and associates an existing database component *Database1* with that session:

```
TSession IBSession;
{
 IBSession = Sessions->OpenSession("InterBaseSession");
 Database->SessionName = "InterBaseSession";
}
```

Note: Do not confuse a session component's *SessionName* property with the *Name* property that is common to all components.

Using the Databases property

Databases is an array of all currently active database components associated with a session. Used with the *DatabaseCount* property, *Databases* can be used to iterate through all active database components to perform a selective or universal action. For example, the following code closes all active databases associated with the default session except for the one named "InterBase1":

```
int MaxDbCount;
{
while (Session1->DatabaseCount > 1)
 {
 MaxDbCount = Session1->DatabaseCount - 1;
 while (Session1->Databases[MaxDbCount]->Name = "InterBase1")
   {
     MaxDbCount--;
     Session1->Databases[MaxDbCount]->Close();
   }
 }
}
```

Using the DatabaseCount property

DatabaseCount is an integer property that indicates the number of currently active databases associated with a session. As connections are opened or closed during a session's life-span, this number can change. For example, if a session's *KeepConnections* property is *false* and all database components are created by C++Builder as needed at runtime, then each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

DatabaseCount is typically used with the *Databases* property to perform actions common to active database components. For example, the following code sets the *KeepConnection* property of each active database in the default session to *true*:

```
int MaxDbCount;
{
if (Session1->DatabaseCount > 0)
 {
 MaxDbCount = 1;
   while (MaxDbCount <= DatabaseCount)
     {
     Session1->DatabasesCount[MaxDbCount]->KeepConnection = true;
     MaxDbCount++;
     }
 }
}
```

Using session events

Two events are associated with a session component: *OnStartup* and *OnPassword*. *OnStartup* is triggered when a session is activated. A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *true* from *false* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets).

OnPassword is only called when an application attempts to open a Paradox table for the first time and the BDE reports insufficient access rights. If you do not code *OnPassword*, the session creates a default dialog box that prompts the user for a password. For more information about writing your own *OnPassword* event handler, see the VCL Reference.

Using session methods

A session component has many methods. The following table lists some of the more useful methods and their purposes:

Method	Purpose
Close	Closes all active databases and datasets, and sets its *Active* property to *false*;
CloseDatabase	Closes a specified database component.
Open	Sets the *Active* property of a session to *true*.
OpenDatabase	Opens a database specified by name. If successful, this function returns a pointer to the database component.
AddPassword	Adds a password to a list of known passwords for the session. If you write an OnPassword event handler to get new passwords at runtime, the handler should call this function to add new passwords to the list.
RemovePassword	Deletes a password from the list of passwords.
RemoveAllPasswords	Clears the password list.
DropConnections	Closes all currently inactive databases and datasets.

In addition to these methods, the following table lists session methods that enable an application to get database-related information:

Method	Purpose
GetAliasNames	Retrieves the list of BDE aliases for a database.
GetAliasParams	Retrieves the list of parameters for a specified BDE alias of a database.
GetAliasDriverName	Retrieves the BDE driver for a specified alias of a database.
GetDatabaseNames	Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use.
GetDriverNames	Retrieves the names of all currently installed BDE drivers.
GetDriverParams	Retrieves the list of parameters for a specified BDE driver.
GetTableNames	Retrieves the names of all tables matching a specified pattern for a specified database.
GetStoredProcNames	Retrieves the names of all stored procedures for a specified database.

For a complete list of all the methods and their arguments available to session components, see the VCL Reference.

Using the default session

C++Builder creates a default session component named "Session" for a database application each time it runs (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are temporary (created by the session at runtime when a dataset is opened that is not associated with a database component you create), or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

When you create a database component, it is automatically associated with the default session. You need only associate a database component that you create with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, you are creating a multi-threaded database application, and must create one additional session to handle each additional thread.

Creating additional sessions

If you create a single application that uses multiple threads that perform database operations, you must create one additional session for each thread. The Data Access page on the component palette contains a session component that you can place in a data module or on a form at design time.

Important:

When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the global function *Sessions->OpenSession* at runtime.

Sessions->OpenSession requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions->OpenSession(String("RunTimeSession" + IntToStr(Sessions->Count + 1)));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

Managing multiple sessions

Sessions is a component of type *TSessionList* that is automatically instantiated for database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the *TSessionList* component:

Property or method	**Purpose**
Count	Returns the number of sessions, both active and inactive, in the sessions list.
FindSession	Searches the session list for a session with a specified name, and returns a pointer to the session component, or NULL if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*.
GetSessionNames	Returns a string list containing the names of all currently instantiated session components. This procedure always returns at least one string, 'Default' for the default session (note that the default session's name is actually a blank string).
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

Using TDatabase components

A *TDatabase* component encapsulates the connection to a single database within a session in an application. If you do not create database components at design time or runtime, a temporary database component is created automatically at runtime each time an application attempts to open a dataset not associated with an available database component.

For most client/server applications you should create your own database components instead of relying on temporary ones. You gain greater control over your databases, including the ability to

- Create persistent database connections.
- Customize database server logins.
- Control transactions and specify transaction isolation levels.
- Create BDE aliases local to your application.

Using temporary database components

To use temporary database components, you need do nothing. C++Builder creates temporary database components as necessary for any datasets in a data module or form for which you do not create them yourself. Temporary database components provide broad support for many typical desktop database applications without requiring you to handle the details of the database connection.

Some key properties of temporary database components are set by the session. For example, the controlling session's *KeepConnections* property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPassword* event for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box. Other properties of temporary database components provide standard login and transaction handling.

The default properties created for temporary database components provide reasonable, general behaviors meant to cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own database components to tune each database connection to your application's needs.

Creating database components at design time

The Data Access page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write *OnLogin* events for it. *OnLogin* offers you a chance to customize the handling of security on a database server when a database component first attaches the server.

Creating database components at runtime

You can also create database components at runtime. An application might do this when the number of database components needed at runtime is unknown, and your application needs explicit control over the database connection. In fact, C++Builder itself creates temporary database components as needed at runtime. When you create a database component at runtime, you need to give it a unique name and to associate it with a session. You actually create the component by calling the *TDatabase->Create* constructor. Given both a database name and a session name, the following function creates a database component at runtime, associates it with a specified session (creating a new session if necessary), and sets a few key database component properties:

```
TDatabase RunTimeDbCreate(String DatabaseName, String SessionName)
{
 TDatabase *TempDatabase;
 TempDatabase = NULL;
 try
   { //if the session exists, make it active; if not, create a new session
   Sessions->OpenSession(SessionName);
   if ((Sessions->FindSession(SessionName)->FindDatabase(DatabaseName)) == NULL)
     {
        // Create a new database component
         TempDatabase = new TDatabase(this);
         TempDatabase->DatabaseName = DatabaseName;
         TempDatabase->SessionName = SessionName;
         TempDatabase->KeepConnection = true;
     }
     return OpenDatabase(DatabaseName);
   }
 catch(...)
 {
   delete TempDatabase;
   throw;
 }
}
```

The following code fragment illustrates how you might call this function to create a database component for the default session at runtime:

```
{
TDatabase MyDatabase[10];
int MyDbCount;
MyDbCount = 1;
 ...
 //Later, create a database component at runtime
MyDatabase[MyDbCount] = RunTimeDbCreate(String("MyDb" + IntToStr(MyDbCount)));
MyDbCount++;
}
```

Using TDatabase properties

Whether you create database components at design time or runtime, you can use the properties of *TDatabase* to change the behavior of the components. The following table summarizes the most important database component properties, what they are used for, their default values if any, and whether they are available at design time:

Property	Purpose	Default	Design time access
AliasName	Identifies the BDE alias associated with this component. If *DriverName* is set, this property is cleared.		Yes
Connected	*true*, component is connected to a database at runtime when the application starts. *false*, component is not connected at start-up.	*false*	Yes
DatabaseName	Specifies the name of the database that must be used by dataset and session components to associate themselves with this component.		Yes
DataSetCount	Provides an integer value specifying the number of currently active datasets for the component.		No
DataSets	Specifies an array of all active datasets associated with the component.		No
Directory	Specifies the directory where the database resides.		No
DriverName	Identifies the BDE driver for the component. If *AliasName* is set, this property is cleared.		Yes
IsSQLBased	Indicates whether a component is associated with an SQL-based database. Read-only.		No
KeepConnection	*true*, maintain database connection even if there are no open datasets. *false*, close database connection when there are no open datasets.	*true*	Yes
LoginPrompt	*true*, prompt for a login name and password on first attaching to a database. *false*, password must be supplied in *Params* property.	*true*	Yes
Params	Lists the login parameters for a given connection.		Yes
Session	Identifies the session component with which this component is attached. Read only.		No
SessionName	Specifies name of the session with which this component is attached.	Default	Yes
TransIsolation	Specifies the transaction isolation level for the component.	*tiReadCommitted*	Yes

The following sections discuss some of these properties in more detail. For details about all *TDatabase* properties, see the VCL Reference.

Using the AliasName, DatabaseName, and DriverName properties

AliasName and *DriverName* are mutually-exclusive BDE-specific properties. *AliasName* specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

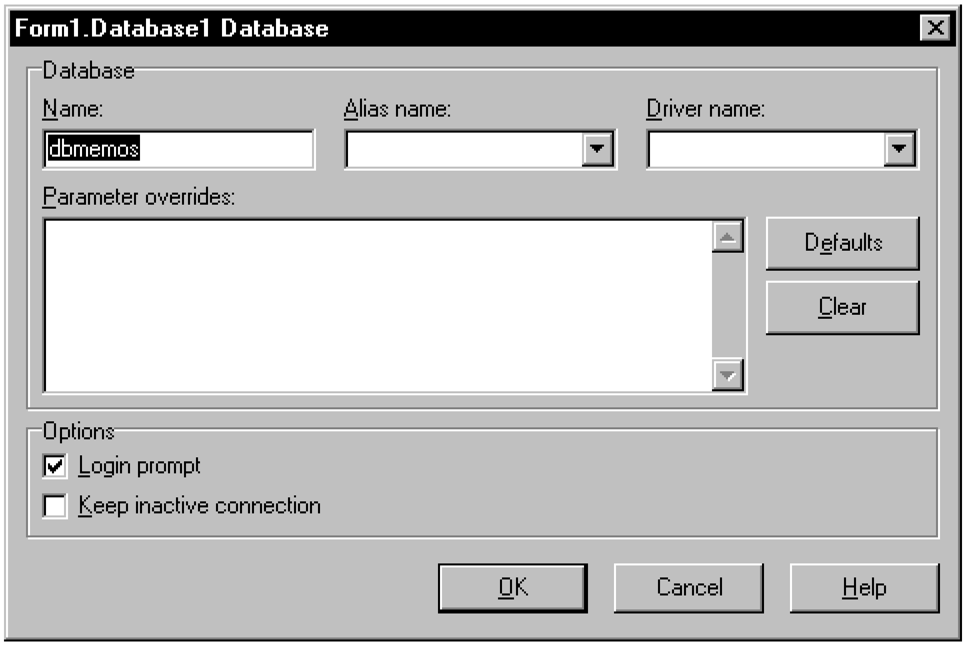
Note: You create and edit BDE aliases using the Database Explorer or the BDE Configuration Utility.

DatabaseName enables you to provide an alternative name for a database component. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to enable you to link them to a database component.

DriverName is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local
BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

Database Properties editor



You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

Note: The Database Properties editor also enables you to view and set BDE connection parameters, and set the states of the LoginPrompt and KeepConnection properties.

To set *DatabaseName*, *AliasName*, or *DriverName* at runtime, include the appropriate assignment statement in your code. For example, the following code uses the text from an edit box to create a local alias for the database component *Database1*:

```
Database1->DatabaseName = Edit1->Text;
```

You can also set or change these values at runtime in the *Params* string list.

Using the KeepConnection and Connected properties

KeepConnection determines if your application maintains a connection to a database even when all datasets associated with that database are closed. If *true*, a connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, make sure *KeepConnection* is *true* to reduce network traffic and speed up your application. If *false* (the default) a connection is dropped when there are no active datasets using the database. If a dataset is later opened which uses the database, the connection must be reestablished and initialized.

Connected enables you to close all active datasets for a database component and drop all database connections even if *KeepConnection* is *true*. When *Connected* is *true* (the default), this property has no effect on database connections. If you set *Connected* to *false*, all active datasets are closed, and all database connections are dropped. For example, the following code closes all active datasets for a database component and drops its connections:

```
Database1->Connected = false;
```

Using the DataSets and DataSetCount properties

DataSets is an indexed array of all active datasets (*TTable*, *TQuery*, and *TStoredProc*) for a database component. An active dataset is one that is currently open. *DataSetCount* is a read-only integer value specifying the number of currently active datasets.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets to set *CachedUpdate* for datasets of type *TTable* to *true*:

```
for (int i = 0; i < DataSetCount; i++)
{
 if (DataSets[i] == TTable)
   DataSets[i]->CachedUpdates = true;
}
```

Using the LoginPrompt and Params properties

LoginPrompt specifies whether your users are prompted to log in to a database server the first time your application attempts to connect to a database requiring a login. If *true* (the default), your application displays a standard Login dialog box. The Login dialog box prompts for a user name and password. A mask symbol (an asterisk by default) displays for each character entered in the *Password* edit box.

If you set *LoginPrompt* to *false*, the standard Login dialog box is not displayed at runtime when an application attempts to connect to a database server that requires a login. Instead, your application must provide the user name and password, either through the *Params* property or the *OnLogin* event for the database component. At design time, the standard Login dialog box appears when you first connect to a remote database server if *LoginPrompt* is *false* and the *Params* property does not contain user name and password entries.

Important:

Storing hard-coded user name and password entries in the *Params* property or in code for an *OnLogin* event can compromise server security.

The *Params* property is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links Help.

At design time you can create or edit connection parameters in three ways:

1. Use the Database Explorer or BDE Configuration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
2. Double-click the *Params* property in the Object Inspector to invoke the String List editor. To learn more about the String List editor, see the C++Builder *User's Guide*.
3. Double-click a database component in a data module or form to invoke the Database Properties editor.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings for a selected alias or driver name, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

Using the SessionName and Session properties

SessionName identifies the alias for the session component with which to associate a database component. When you first create a database component at design time, *SessionName* is set to "Default".

Multi-threaded applications may have more than one session. At design time, you can pick a valid

SessionName from the drop-down list in the Object Inspector. Session names in that list come from the *SessionName* properties of each session component.

Session is a runtime, read-only property that references the session component corresponding to the value of the *SessionName* property. For example, if *SessionName* is blank or "Default", then the *Session* property references the same *TSession* instance referenced by the global *Session* variable.

Using the TransIsolation property

TransIsolation specifies the transaction isolation level for a database component's transactions. *Transaction isolation level* determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transaction's changes to a table. Before changing or setting *TransIsolation*, you should be familiar with transactions and transactions management in C++Builder. For a complete discussion of transaction management in C++Builder, see "Managing transactions."

The default setting for *TransIsolation* is *tiReadCommitted*. The following table summarizes possible values for *TransIsolation* and describes what they mean:

Isolation level	Meaning
tiDirtyRead	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
tiReadCommitted	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
tiRepeatableRead	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

To learn more about choosing the correct isolation level for each database component in your application, see "Managing transactions."

Using database events

A single event, *OnLogin* is associated with a database component. If you code it, OnLogin is triggered when a database component establishes a new connection to a database server. This event is usually used to trap and handle remote server login requests instead of displaying the default Login dialog box when an application runs.

Using database methods

A database component has several methods. The following table lists them and their purposes:

Method	Purpose
ApplyUpdates	Applies cached updates to a database.
Close	Closes the database connection.
CloseDataSets	Closes any active datasets associated with the component.
Commit	Commits data changes and ends the current transaction.
Open	Opens a connection to the database.
Rollback	Discards data changes and ends the current transaction.
StartTransaction	Starts a new transaction.
ValidateName	Verifies that a specified database name is valid.

Connecting to a remote server

When you connect to a remote database server in a C++Builder application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the Params property of a database component. For information about SQL Links parameters, see the online *SQL Links User's Guide*.

Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP. C++Builder client applications can use TCP/IP, SPX/IPX, or NetBEUI network protocols if the proper communications software is installed on both the server and the client machines.

Establishing an initial connection between client and server can be problematic, especially when using TCP/IP. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- If you are using TCP/IP:
- Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
- Is either

the server's IP address registered in the client's HOSTS file? or
the Domain Name Services (DNS) properly configured?

- Can you ping the server?
- Are the DLLs for your connection and database drivers in the search path?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

Using ODBC

A C++Builder application can use ODBC data sources such as DB2, Btrieve, or Microsoft Access. An ODBC driver connection requires:

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- The BDE Configuration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Configuration utility. The BDE Configuration setting AUTO ODBC (on the systems page) enables the BDE to configure itself automatically for ODBC. When AUTO ODBC is TRUE, data source and driver information are automatically imported from the ODBC.INI file. When AUTO ODBC is FALSE, you must manually create an ODBC configuration using the BDE Configuration utility. For more information, see the BDE Configuration utility's online Help file.

Handling server security

Most remote database servers include security features to prohibit unauthorized access. Generally, the server requires a user name and password login before permidtting database access.

At design time, if a server requires a login, C++Builder displays a standard Login dialog box that prompts you for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

- Set the *LoginPrompt* property of a database component to *true* (the default). Your application displays the standard Login dialog box when the server requests a user name and password.
- Set the *LoginPrompt* to *false*, and include hard-coded USER NAME and PASSWORD parameters in the *Params* property for the database component. For example:

```
USER NAME = SYSDBA
PASSWORD = masterkey
```

Important:

Note that because the *Params* property is easy to view, this method compromises server security, so it is not recommended.

- Write an *OnLogin* event for the database component, and use it to set login parameters at runtime. *OnLogin* gets a copy of the database component's *Params* property, which you can modify. The name of the copy in *OnLogin* is *LoginParams*. Use the *Values* property to set or change login parameters as follows:

```
LoginParams->Values["USER NAME"] = UserName;
LoginParams->Values["PASSWORD"] = PasswordSearch(UserName);
```

On exit, *OnLogin* passes its *LoginParam* values back to *Params*, which is used to establish a connection.

Interactions between TSession and TDatabase

In general, *TSession* properties, such as *KeepConnections*, provide global, default behaviors that apply to all temporary database components created by C++Builder as needed at runtime.

Methods apply somewhat differently. *TSession* methods affect all database components, regardless of database component status. For example, the session method *DropConnections* closes all datasets belonging to a session's database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *true*.

TDatabase methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1->CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

Using TSession and TDatabase in data modules

You can safely place *TSession* and *TDatabase* components in data modules. If you put a data module that contains *TSession* or *TDatabase* components into the Object Repository, however, other users can copy your data module, but they cannot inherit from it. These components share global namespace, and inheriting from them results in namespace conflicts.

...

Introduction

This topic describes using Borland C++Builder™ to create database client applications. In C++Builder, database applications use the Borland Database Engine (BDE) to retrieve data from and send data to local and remote database servers. This guide

- Provides an overview of C++Builder's data-access components, data-aware visual controls, and the C++Builder database development model.
- Explains how to use data modules and the Object Repository to modularize and reuse database connections and forms within and among applications.
- Describes how to connect to databases and manage database transactions.
- Details how to work with the field, table, query, and stored procedure components that encapsulate your application's view of databases.
- Describes the data-aware visual controls that provide a user interface to data in an application.
- Explains how and when to use cached updates in an application.
- Discusses how to upsize an application from using desktop databases to using SQL databases on remote servers.

The VCL Reference more fully describes the data-access components and data-aware visual controls. For an overview of the complete C++Builder documentation set, see the introduction to the C++Builder User's Guide.

Press the >> button to read through topics in sequence.

Other places to look:

What's in help?

Software registration and technical support

What's in help?

This guide contains the following information :

Chapter	Description
"Using the C++Builder database development model"	Describes C++Builder's database features and how to use them most effectively when building a database application.
"Using data modules"	Explains data modules and how to use them.
"Connecting to databases"	Explains how to use the data-access components that encapsulate connections to databases and database sessions.
"Managing transactions"	Discusses managing transactions with database component methods or with passthrough SQL.
"Accessing data in databases"	Describes the properties and methods common to all dataset components such as *TTable*, *TQuery*, and *TStoredProc*.
"Linking visual controls to datasets"	Explains how to use the *TDataSource* component that acts as a conduit between dataset components and data-aware visual controls on forms.
"Creating and using fields"	Explains how to use the properties and methods of field components that encapsulate the appearance of individual fields in records displayed in data-aware visual controls on forms.
"Working with tables"	Describes how and when to use table components in a database application.
"Working with queries"	Describes how and when to use query components in a database application.
"Working with stored procedures"	Describes how and when to use stored procedures in a database application.
"Displaying and editing data in data-aware controls"	Describes the properties and methods common to all data-aware visual controls, and how to use most controls on forms.
"Navigating datasets"	Describes how to use the *TDBNavigator* visual control to enable users to move from record to record in a dataset, change dataset states, post and cancel data changes, and refresh data display.
"Displaying and editing data in grids"	Describes how to the *TDBGrid* visual control to display and edit data in tabular format; describes how to use the *TDBCtrlGrid* visual control to display other data-aware components as repeating units in a grid format where each set of controls corresponds to a single record in an underlying dataset.
"Caching updates"	Explains how and when to use cached updates in an application, and how to use the *TUpdateSQL* component to update read-only result sets returned by some queries.
"Handling batch move operations"	Explains how to use the *TBatchMove* component to copy data from one table to another in an application.
"Upsizing and deploying an application"	Discusses the issues involved in modifying an application written to local desktop database access to use SQL databases on remote servers.

Software registration and technical support

The Borland Assist program offers a range of technical support plans to fit the diverse needs of individuals, consultants, large corporations, and developers. To receive help with this product, return the registration card and select the Borland Assist plan that best suits your needs. North American customers can register by phone 24 hours a day at 1-800-845-0147. For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with this product.

Navigating datasets

This topic describes how to use the data-aware *TDBNavigator* control to provide dataset record navigation and manipulation on forms in your database applications. It also describes how to customize the navigator at design time and runtime, and how to use a single navigator to control navigation and manipulation of multiple datasets.

Data-aware controls

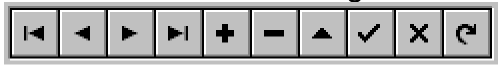
Press the >> button to read through each topic in sequence.

Navigating and manipulating records

The TDBNavigator component is a simple control that is used to move through the data in a database table or query and perform operations on the data, such as inserting a blank record or posting a record. It is used with the data-aware controls, such as the data grid, which give you access to the data, either for editing the data, or for simply displaying it.

The following figure shows the navigator component as it appears by default. The component consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Buttons on the TDBNavigator control



The following table describes the buttons on the navigator:

Button	Purpose
First	Calls the dataset's First method to set the current record to the first record in the dataset, disable the First and Prior buttons, and enable the Next and Last buttons if disabled.
Prior	Calls the dataset's Prior method to set the current record to the previous record and enable the Last and Next buttons if disabled.
Next	Calls the dataset's Next method to set the current record to the next record and enable the First and Prior buttons if disabled.
Last	Calls the dataset's Last method to set the current record to the last record, disable the Last and Next buttons, and enable the First and Prior buttons if disabled.
Insert	Calls the dataset's Insert method to insert a new record before the current record, and set the dataset in Insert and Edit state.
Delete	Deletes the current record and makes the next record the current record. If the ConfirmDelete property is *true,* it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, restores the record display to its condition prior to editing, and turns off Insert and Edit states if active.
Refresh	Clears data-aware control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

In some data-aware controls, such as DBGrid, you can also click in the field value to put a dataset into Edit state and begin making changes or entering data. In some data-aware controls, such as *DBGrid*, you can also post a record by hitting the Enter key or by clicking in another field value.

Creating a navigator

You can place a navigator component on a form by selecting a *DBNavigator* component from the Data Controls tab of the Component palette.

Data Controls tab of the Component palette

You link the database navigator with a dataset when you specify a data source component that identifies the dataset as the value of navigator's *DataSource* property.

Using a single navigator for multiple datasets

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two DBEdit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the *DBEdit* control connected to *CustomersSource* (CustomerCompany), the navigator should also use *CustomersSource*, and when the user enters the *DBEdit* control connected to *OrdersSource* (*OrderNum*), the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the *DBEdit* controls, and then share that event with the other *DBEdit* control. For example,

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
 if (Sender == CustomerCompany)
   DBNavigatorAll->DataSource = CustomerCompany->DataSource;
 else
   DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

To learn more about sharing event handlers, see the C++Builder *User's Guide*.

Choosing navigator buttons to display

When you first place a TDBNavigator component on a form, all of its buttons are visible by default. You can set the VisibleButtons property to turn off buttons you do not want to use on a form. For example, on a form that is intended for browsing rather than editing, you might want to disable the Edit, Insert, Delete, Post, and Cancel buttons.

Choosing navigator buttons to display at design time

Hiding and showing navigator buttons at runtime

Displaying fly-over help

Choosing navigator buttons to display at design time

The VisibleButtons property in the Object Inspector is displayed with a plus (+) sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, double-click the VisibleButtons property. The list of buttons that can be turned on or off appears in the Object Inspector below the VisibleButtons property. The + sign changes to a minus (-) sign, indicating that you can collapse the list of properties by double-clicking the VisibleButtons property.

Button visibility is indicated by the Boolean state of the button value. If a value is set to *true*, the button appears in the TDBNavigator. If *false*, the button is removed from the navigator at design and runtime.

Note: As button values are set to *false*, they are removed from the TDBNavigator on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

For more information about buttons and the methods they call, see the *VCL Reference*.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert, Delete, Edit, Post, Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, the code illustrates how to switch a navigator's data source at runtime. Suppose that in addition to switching data sources, you want to prevent edits to the *OrdersTable* by hiding the *Insert, Delete, Edit, Post, Cancel,* and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might add the necessary code to the previously coded *OnEnter* event handler:

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
 if (Sender == CustomerCompany)
 {
   DBNavigatorAll->DataSource = CustomerCompany->DataSource;
   Set<TNavigateBtn, 0, 9> btnShow;
   btnShow << nbFirst << nbPrior << nbNext << nbLast << nbInsert << nbDelete << nbEdit
<< nbPost << nbCancel << nbRefresh;
   DBNavigatorAll->VisibleButtons = btnShow;
 }
 else
 {
   DBNavigatorAll->DataSource = OrderNum->DataSource;
   Set<TNavigateBtn, 0, 3> btnShow;
   btnShow << nbFirst << nbPrior << nbNext << nbLast;
   DBNavigatorAll->VisibleButtons = btnShow;
 }
}
```

Displaying fly-over help

To display fly-over help for each navigator button at runtime, set the navigator *ShowHints* property to *true*. When *ShowHints* is *true*, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. *ShowHints* is *false* by default.

The *Hints* property controls the fly-over help text for each button. By default, *Hints* is an empty string list. When *Hints* is empty, C++Builder displays default help text for each button. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

Creating and using fields

This topic describes the *TField* object class that represents individual database columns, or fields, in datasets, and it describes how to use its descendant field components in your database applications. *TField* objects control such things as whether or not data from a column is displayed and its format. By default, one field object is created for every column of its associated table component. The following figure highlights the relationship of the *TField* component to the other data access components in C++Builder:

C++Builder data access component hierarchy

Fields may be of different data types. Each data type maintains a different type of data internally and transfers that data to and from an associated database table. You never directly use a *TField* component in your applications. Instead you use its descendant field components, each of which represents a different data type in a column of a table or query underlying a dataset component. The following table lists each type of field component:

Component name	Used for:
TAutoIncField	Represented as a binary value with a range from -2,147,483,648 to 2,147,483,647. Used for auto-incrementing fields that hold large, signed, whole numbers.
TBCDField	Real numbers with a fixed number of digits after the decimal point. Accurate to 18 digits. Range depends on the number of digits after the decimal point. [Paradox only]
TBooleanField	true or false values
TBlobField	Arbitrary data field without a size limit
TBytesField	Arbitrary data field without a size limit
TCurrencyField	Currency values. The range and accuracy is the same as *TFloatField*
TDateField	Date value
TDateTimeField	Date and time value
TFloatField	Real numbers with absolute magnitudes from 5.0*10-324 to 1.7*10+308 accurate to 15-16 digits
TGraphicField	Arbitrary length graphic, such as a bitmap
TIntegerField	Whole numbers in the range -2,147,483,648 to 2,147,483,647
TMemoField	Arbitrary length text
TNumericField	Numeric data types
TSmallintField	Whole numbers in the range -32,768 to 32,767
TStringField	Fixed length text data up to 255 characters
TTimeField	Time value
TVarBytesField	Arbitrary data field up to 65,535 characters, with the actual length stored in the first two bytes
TWordField	Whole numbers in the range 0 to 65,535

Even though there are many field components, they share most of their properties, methods, and events. Press the **>>** button to read through a discussion of the shared properties, methods, and events of field components in sequence. For information specific to individual field components, see the VCL Reference.

Other places to look:

Understanding field components

Understanding field components

Like the other C++Builder data access components, field components are nonvisual. Field components are also not directly visible at design time. Instead they are associated with a dataset component, and provide data-aware components such as TDBEdit and TDBGrid access to database fields through that dataset.

When a dataset is opened, one field component is generated for each column of data in the underlying table or query. C++Builder uses the Borland Database Engine (BDE) to determine the correct type of field component to assign to each column. The type of field component determines its properties and how data associated with that field is displayed in data-aware controls on a form. For example, a *TFloatField* component has four properties that directly affect the appearance of data:

Property	Description
DisplayWidth	Controls the number of digits displayed in a control.
DisplayFormat	Specifies the appearance of values in a control (for example, the number of decimal places to display).
Alignment	Centers or left or right aligns the values in a control for display.
EditFormat	Controls the appearance of values in a control during editing.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Some properties, such as *Precision*, can also affect how data entered in a control by a user is written back to the table underlying a dataset.

The appearance of data can also be affected in other ways. You can use the Data Dictionary to create extended field attribute sets that describe the content and appearance of data. You can also control the appearance of columns in data-aware grids.

You cannot change a field component directly. To change the data type of a field, you must define a new field and replace one data type with another.

Field components also provide events, such as *OnChange*, which can react to editing of data in a field, and *OnValidate*, which can be used to implement field-based validation rules.

Choosing between dynamic and persistent field generation

Field components are dynamically generated at design time when the *Active* property for a dataset is set to *true*. C++Builder creates field components for the dataset based on the underlying physical structure of columns in one or more database tables each time a dataset it activated. One dynamic field component is created for each column in the underlying table.

Dynamic field generation means that C++Builder provides a standard set of default properties for each field component it generates, and that these defaults might change if there are fundamental changes to the underlying table. Every time the physical structure of the database tables changes (for example, because columns are dropped, added, or altered), the fields available to your C++Builder database application also change.

However, while dynamic generation of field components might be appropriate for some applications and might not be an issue at design time when you can change what your application looks like, it may become an issue at runtime, when your application's behavior and data appearance are fixed. For example, if you create a customized database grid based on an expected number of fields in a table, and a new field is added to the table--or one is removed--your application may not work as expected. In this case, you need to generate a persistent set of field components for your application.

Creating persistent field components is different from dynamically generating field components because if the physical structure of the table changes, the fields in the program do not change. The field descriptions are "hard-coded," making a more stable program. You can use persistent field components to add, change, remove, and create field components, but you cannot modify the underlying table structure.

Benefits of dynamic field generation

Dynamic generation of field components is the best way to create field components in the following situations:

- You are designing an application where the underlying structure of the table may change.
- You want to allow the user to change the underlying structure of the table.
- You are displaying existing data and are not making any assumptions about that data.
- You are creating an application for exploring a database.

You can choose which fields to display and the order of display in a grid by double-clicking the grid and using the DBGrid Columns editor

Benefits of persistent field generation

Creating persistent field components offers the following advantages. You can

- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Modify field component display and edit properties.
- Define lookup fields that compute their values based on fields in other datasets.
- Define calculated fields that compute their values based on other fields in the dataset.
- Define new fields--usually to replace existing fields--based on columns in the table or query underlying a dataset.
- Add field components to the list of persistent components.
- Restrict the fields in your dataset to a subset of the columns available in the underlying database.

At design time, you can use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. All fields in a dataset are either persistent or dynamic.

Note: In customized data grids you can create persistent column objects that duplicate much of this functionality.

Creating persistent fields

You can create persistent field components with the Fields editor to provide efficient, readable, and type-safe programmatic access to underlying data. Persistent field generation guarantees that each time your application runs, it always uses and displays the same columns in the same order, even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, C++Builder generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields, start the Fields editor by double-clicking the dataset (TTable, TQuery) component.

The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. So, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers', or as much of the name as fits.

Below the title bar is a set of navigation buttons that enable you to scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty.

The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

Tip:

You can drag and drop fields from the Fields editor onto a form.

Press the >> button to read through topics in sequence.

Other places to look:

Adding persistent fields

Arranging the order of persistent field components

Defining new persistent fields

Deleting persistent field components

Adding persistent fields

To add a persistent field component for a dataset, right-click the Fields editor list box and choose Add fields.

The Available fields list box displays all fields in the dataset which do not have persistent field components. Select the fields for which to create persistent field components, and click OK.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, C++Builder no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified and only the fields you specified will be displayed in the data-aware control.

Each time you open the dataset, C++Builder verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, C++Builder raises an exception warning you that the field is not valid, and does not open the dataset.

Arranging the order of persistent field components

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of a one or more fields

1. Select one or more fields (use Ctrl-Click to select more than one field).
2. Drag it to a new location.

Alternatively, you can select the field, and use Ctrl-Up and Ctrl-Dn to move the field to a new location in the list.

If you select a non-contiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block the order of fields to one another does not change.

Defining new persistent fields

Besides making dynamic field components into persistent field components for a dataset, you can also create new persistent fields as additions to or replacements of the other persistent fields in a dataset. You can create the following types of persistent fields:

Field type	Description
Data fields	Replaces existing fields (for example to change the data type of a field), based on columns in the table or query on the underlying dataset.
Calculated fields	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup fields	Retrieves values from a specified dataset at runtime based on criteria you specify.

These types of persistent fields are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and data underlying the dataset is not changed in any way.

To create a new persistent field component, right-click the Fields editor list box and choose New field.

The Field properties group box lets you enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's *DataField* property, excluding spaces. C++Builder uses this name to build a component name in the Component edit box. The name in the Component edit box corresponds to the field component's *Name* property and is provided for informational purposes (*Name* contains the identifier by which you refer to the field component in your source code). C++Builder discards anything you enter directly in the Component edit box.

Specify the field component's data type in the Type combo box. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. The Size edit box enables you to specify the maximum number of characters that can be displayed or entered in a string-based field or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.

Specify the type of new field component to create in the Field type radio group. The default type is Data. If you choose Lookup, the Dataset and Key Fields edit boxes in the Lookup definition group box are enabled.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallintField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important:

Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a data field,

1. Start the Fields editor by double-clicking the dataset (*TTable*, *TQuery*) component.
2. Select the field in the list box and delete the field name. The field will not be deleted from the table, only from the current list of persistent fields.
3. Enter the name of an existing persistent field in the Name edit box. Do not enter a new field name.
4. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing, but it should be a logical change.
5. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
6. Select Data in the Field type radio group if it is not already selected.
7. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in step 1, and the component declaration in the data module or form's class declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's OnCalcFields event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field

1. Start the Fields editor by double-clicking the dataset (*TTable*, *TQuery*) component.
2. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
3. Choose a data type for the field from the Type combo box.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
5. Select Calculated in the Field type radio group.
6. Choose OK. The newly defined calculated field is automatically added to end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's class declaration in the source code.
7. Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see below.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Note: The *Calculated* and *Lookup* properties of a field component are mutually exclusive. Setting one to *true* automatically sets the other to *false*.

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the OnCalcFields event for its dataset.

To program a value for a calculated field:

1. Select the dataset component (*TTable*, *TQuery*) from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the OnCalcFields property to bring up or create a CalcFields function for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for a *Customers* table on a *CustomerData* data module. *CityStateZip* should display a customer's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* function for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustForm::CustomersCalcFields* function appears in the unit's source code window. Add the following code to the function to calculate the field:

```
CustomersCityStateZip->Value = CustomersCity->Value +", " + CustomersState->Value + " " +
CustomersZip->Value;
```

Defining a lookup field

Lookup fields are much like calculated fields, but their "calculation" is to "lookup" information in another table. A lookup field displays the values it returns at runtime based on lookup criteria. In its simplest form, a lookup field is passed the name of a field to search, a field value to search for, and the field in the lookup dataset whose value it should display. Lookup fields are always read-only.

Note: The *Calculated* and *Lookup* properties of a field component are mutually exclusive. Setting one to *true* automatically sets the other to *false*.

For example, consider a mail-order application that enables an operator use a lookup field to determine automatically the city and state that correspond to a zip code a customer provides. In that case, the column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those in the *ZipTable.City* and *ZipTable.State* columns for the record where *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note: The use of the *Table.Field* is applicable if the dataset resides in a data module. If the dataset resides on a form, only the field name is necessary.

To create a lookup field:

1. Start the Fields editor by double-clicking the dataset (*TTable*, *TQuery*) component.
2. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
3. Choose a data type for the field from the Type combo box.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
5. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
6. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes. The lookup dataset must reside on the same form or in the same data module as the dataset for the field component.
7. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons.
8. In the Lookup Keys drop-down list, choose a field in the lookup dataset to match against the Key Fields field you specified in step 6. To specify more than one field, enter field names directly instead. Separate multiple field names with semicolons.
9. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating. To return values from more than one field in the lookup dataset, enter field names directly instead. Separate multiple field names with semicolons.

Another way to define a lookup field is to complete steps 1-4 above and then select the lookup field in the Fields editor. The boxes in the Fields editor are displayed as properties in the Object Inspector, therefore, you can set the *LookupDataSet*, *LookupKeyFields*, and *LookupResultField* properties there. For the lookup field to be activated, the *Lookup* property must be set to *true*.

Note: Field lookup happens before field calculations are performed. You can therefore base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

Deleting persistent field components

You delete persistent field components to access a subset of available columns in a table, and to define your own persistent fields to replace a column in a table. To remove one or more persistent field components in a dataset:

1. Start the Fields editor by double-clicking the dataset (*TTable*, *TQuery*) component.
2. Select one or more fields to remove in the Fields editor list box.
3. Press Del.

Note: You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate persistent field components that you delete by accident, but any changes previously made to its properties or events are lost.

If you remove all persistent field components for a dataset, C++Builder regenerates all dynamic field components for every column in the database table underlying the dataset.

Setting field properties and events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a TDBGrid, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

Press the >> button to read through topics in sequence.

Other places to look:

Setting display and edit properties of fields

Creating attribute sets for field components

Associating attribute sets with field components

Removing attribute set associations

Working with field properties at runtime

Setting display and edit properties of fields

To edit the properties of a selected persistent field component, switch to the Properties page in the Object Inspector window. The following table summarizes properties that can be edited.

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field's contents in a data-aware component or column.
Calculated	*true*: field value is calculated by a *CalcFields* method at runtime.*false* (the default): field value is determined from the current record.
Currency	Numeric fields only. *true*: displays monetary values.*false* (the default): does not display monetary values.
DisplayFormat	Numeric fields only. Specifies the format of data displayed in a data-aware component.
DisplayLabel	Specifies the column name for a field in a data-aware grid component.
DisplayWidth	Specifies the width, in characters, of a grid column that displays this field.
EditFormat	Numeric fields only. Specifies the edit format of data in a data-aware component.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear in the field (hyphens, parentheses, and so on).
FieldName	Specifies the actual name of a column in the table from which the field derives its value and data type.
Index	Specifies the order of the field in a dataset.
KeyFields	Specifies which field in the current dataset to match in the lookup dataset when *Lookup* is *true*.
Lookup	*true*: Displays lookup field values in data-aware components.*false* (the default): Does not display lookup field values.
LookupDataSet	Specifies the table used to look up field values when *Lookup* is *true*.
LookupKeyFields	Specifies the field or fields in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset whose value you want copied into the lookup field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the component name of the field in C++Builder.
Precision	Numeric fields only. Specifies the number of decimal places of the value to store before rounding begins.
ReadOnly	*true*: Displays field values in data-aware components, but prevents editing.*false* (the default): Permits display and editing of field values.
Required	*true*: Specifies that a non-NULL value for a field is required.*false* (the default): Specifies that a non-NULL value for a field is not required.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField*.
Tag	Long integer bucket available for programmer use in every component as needed.
Transliterate	*true* (the default): Specifies translations to and from the respective locales of the *Source* and *Destination* properties of the dataset will be done.*false*: Specifies these translations will not be done.
Visible	*true* (the default): Permits display of field in a data-aware grid component.*false*: Prevents display of field in a data-aware grid component.User-defined components can make display

decisions based on this property.

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as Calculated, require additional programming steps to be useful. Others, such as DisplayFormat, EditFormat, and EditMask, are interrelated; their settings must be coordinated.

You can also use and manipulate field component properties at runtime.

Using default formats for numeric, date, and time fields

C++Builder provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Routine	**Used by**
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*
FormatCurr	*TCurrencyField*

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Windows 95 or Windows NT Control Panel. For example, using the default settings for the United States, a TFloatField column with the Currency property set to *true* sets the DisplayFormat property for the value 1234.56 to $1234.56, while the EditFormat is 1234.56.

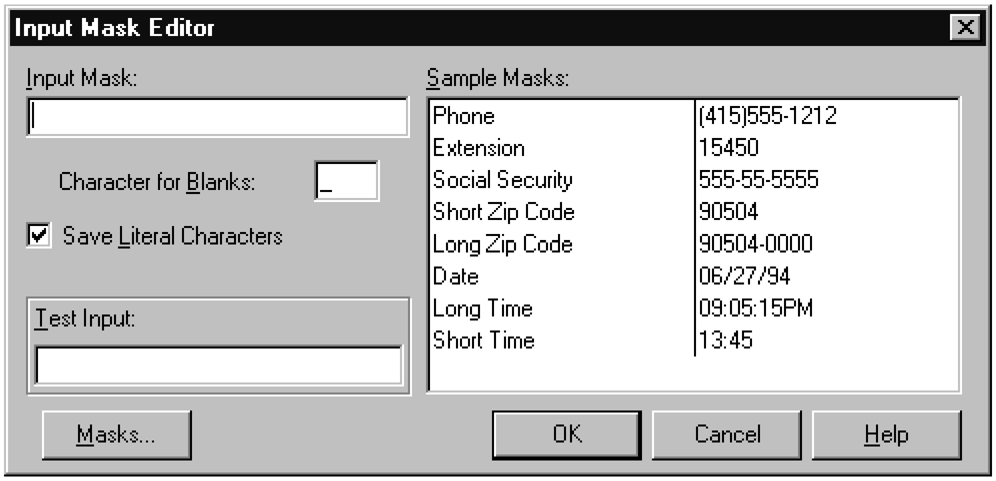
At design time or runtime, you can edit the DisplayFormat and EditFormat properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

Controlling, or masking, user input

The EditMask property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, and *TDateTimeField* components. You can use existing masks, or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can also enter masks directly into the *EditMask* field in the Object Inspector. For a complete discussion of edit masks, their structure, and the symbols they can contain, see the "EditMask property" in the VCL Reference.

To enter an input mask, invoke the Input Mask editor for a field component:

1. Select the component in the Fields editor or Object Inspector.
2. Click the Properties page in the Object Inspector.
3. Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.



Character for Blanks: _

✓ Save Literal Characters

Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	06/27/94
Long Time	09:05:15PM
Short Time	13:45

The Input Mask edit box enables you to create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks--if you have created one--into the Sample Masks grid for easy selection.

Note: For TStringField components, the EditMask property is also its display format.

Creating attribute sets for field components

When several persistent fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets in the Data Dictionary can be easily applied to other fields.

To create an attribute set based on a field component in a dataset:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the Object Inspector.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing *Save attributes as* instead of *Save attributes* from the context menu.

Note: You can also create attribute sets directly from the Database Explorer. When you create an attribute set from the Database Explorer, the set is not applied to any fields. You can specify two additional attributes in the set: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) which is automatically placed on a form when a field based on the attribute set is dragged onto the form. For more information, see the online help for the Database Explorer.

Associating attribute sets with field components

When several persistent fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Right-click the Fields editor list box and choose Associate attributes.
4. Select or enter the attribute set to apply from the Attribute set name dialog box. If an attribute set exists in the Data Dictionary, that set name appears in the edit box.

Removing attribute set associations

If you change your mind about associating an attribute set with a persistent field component, you can easily remove the attribute set from the field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to remove an attribute set.
3. Right-click the Fields editor list box and choose Unassociate attributes.

After you remove an attribute set from a field component, the attributes remain the same as they were when they were associated. You can either use the Object Inspector to set its properties, or you can associate a different attribute set with the component.

Working with field properties at runtime

You can use and manipulate the properties of field components at runtime. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *true*:

```
CustomersCityStateZip->ReadOnly = true;
```

And this statement changes field ordering by setting the Index property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip->Index = 3;
```

Handling field events

Like all C++Builder components, field components have event handlers associated with them. By writing event handlers, you control events that affect data entered in fields using data-aware controls.   The following table lists the events associated with field components:

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and OnSetText events are primarily useful to programmers who want to do custom formatting that goes beyond C++Builder's built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to validate data-entry in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component from the Fields editor or the Object Inspector drop-down list.
2. Select the Events page in the Object Inspector.
3. Double-click the values column for the event handler to display its source code window.
4. Create or edit the handler code.

Working with field methods at runtime

Field component methods available at runtime let you convert field values from one data type to another, and let you set focus on the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). You can validate the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
   CustomersCompany->FocusControl();
```

Note: In this example, you need to create a persistent field for the *Company* field.

The following table lists other field component methods. To access online help for these methods, you can enter the method in the Code editor and then press F1.

Method name	Used for
Assign	Copies data from one field to another. Fields must be the same data type and size.
AssignValue	Sets the field to *Value* using one of the field component conversion functions (depending on the type of value).
Clear	Clears the field, sets *Value* to NULL.
Create	Allocates memory to create a component and initializes its data as needed.
Destroy	Destroys the object component or control and releases memory allocated to it (*Free* is more commonly used for this purpose).
FocusControl	Sets a form's focus to the first data-aware component associated with a field.
Free	Destroys the object and frees its associated memory (use after *Create*).
GetData	Used to obtain "raw" data from the field.
IsValidChar	Used by data-aware controls to determine if a particular character entered in the field is valid for the field.
SetData	Assigns "raw" data to the field.

Displaying, converting, and accessing field values

Data-aware controls such as TDBEdit and TDBGrid automatically display the values associated with field components when a dataset's *Active* property is *true*. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications.

Standard controls (such as *TEdit*) can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part.

Press the >> button to read through topics in sequence.

Other places to look:

Displaying values in standard controls

Converting values

Accessing values with the default dataset method

Accessing values with a dataset's Fields property

Accessing values with a dataset's FieldByName method

Displaying values in standard controls

An application can access the value of a database column through the *Value* property of a field component. For example, the following statement assigns the value of the *CustomersCompany* field to the text in a TEdit control:

```
   Edit3->Text = CustomersCompany->Value;
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in functions for handling conversions.

Note: You can also use variants to access and set field values. Variants are a flexible data type.

Converting values

Conversion functions attempt to convert one data type to another. For example, the *AsString* function converts numeric and Boolean values to string representations. The following table lists field component conversion functions, and which functions are recommended for field components by field-component type:

Function	***TStringField***	***TIntegerField***	***TSmallintField***	***TWordField***	***TFloatField***	***TCurrencyField***	***TBCDField***	***TDateTimeField***	***TDateField***	***TTimeField***	***TBooleanField***	***TBytesField***	***TVarBytesField***	***TBlobField***	***TMemoField***	***TGraphicField***
AsVariant	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsString	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsInteger	3	3	3	3												
AsFloat	3				3	3	3									
AsCurrency																
AsDateTime	3							3	3	3						
AsBoolean	3										3					

Note that the *AsVariant* method can be used for translating among all data types. When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or date-time format only if the string value is in a recognizable date-time format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. This may be useful, for example, when you want to calculate elapsed time. The following table lists permissible conversions that produce special results:

Conversion	**Result**
String to Boolean	Converts "true," "false," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime to Float	Converts date to number of days since December 31, 1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "true" or "false."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception. Permissible conversions are listed in the Field component conversion functions table.

You use a conversion function as you would use any method belonging to a component: append the function name to the end of the component name wherever it occurs in an assignment statement. Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1->Text = CustomersCustNo->AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
CustomersCustNo->AsInteger = StrToInt(Edit1->Text);
```

An exception occurs if an unsupported conversion is performed at runtime.

Accessing values with the default dataset method

The preferred method for accessing a field's value is to use variants with the default dataset method, *FieldValues*. For example, the following statement takes the value from an edit box and transfers it into the *CustNo* field in the *Customers* table:

```
   Customers->FieldByName("CustNo")->AsString = Edit2->Text;
```

Because *FieldValues* is the default method for a dataset, you do not need to specify its method name explicitly. The following statement, however, is identical to the previous one:

```
   Customers->FieldValues["CustNo"] = Edit2->Text;
```

For more information about variants, see the C++Builder Programmer's Guide. For more information about using the *FieldByName* method, see "Accessing values with a dataset's FieldByName method".

Accessing values with a dataset's Fields property

You can access the value of a field with the Fields property of the dataset component to which the field belongs. Accessing field values with a dataset's *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property, you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using the field component's conversion routine.

For example, the following statement assigns the current value of the seventh column (Country) in the Customers table to an edit control:

```
Edit1->Text = CustTable->Fields[6]->AsString;
```

Conversely, you can assign a value to a field by setting the Fields property of the dataset to the desired field. For example,

```
{
 Customers->Edit();
 Customers.Fields[6]->AsString = Edit1->Text;
 Customers->Post();
}
```

Accessing values with a dataset's FieldByName method

You can also access the value of a field with a dataset's FieldByName method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion function, such as AsString or AsInteger. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

Conversely, you can assign a value to a field:

```
{
 Customers->Edit();
 Customers->FieldByName("CustNo")->AsString = Edit2->Text;
 Customers->Post();
};
```

Managing transactions

This topic describes how to manage transactions in a C++Builder database application using implicit and explicit control. It focuses on

- Using the methods and properties of the *TDatabase* component that enable explicit transaction control.
- Using passthrough SQL with *TQuery* components to control transactions.

This topic also discusses how local transactions are handled. Local transactions are transactions made against local Paradox and dBASE tables.

When you create a database application using C++Builder, it provides transaction control for all database access, even against local Paradox and dBase tables. A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all the actions fail.

Transactions ensure database consistency even if there are hardware failures. It also maintains the integrity of data while permitting concurrent multiuser access.

For example, an application might update the ORDERS table to indicate that an order for a certain item was taken, and then update the INVENTORY table to reflect the reduction in inventory available. If a hardware failure occurred after the first update but before the second, the database would be in an inconsistent state because the inventory would not reflect the order entered. Under transaction control, both statements would be committed at the same time. If one statement failed, then both would be undone (rolled back).

Press the >> button to read through topics in sequence.

How transactions differ from cached updates

Explicitly controlling transactions

Transactions against a local table

How transactions differ from cached updates

In the BDE, when a transaction is active, updates are immediately sent to the underlying tables. Thus errors (such as integrity constraint violations, and so on) are instantly reported to the clients. Because updates are immediately sent to the underlying tables, the updates are visible to other transactions. And because each modified record is locked, other users cannot interfere.

This behavior differs from that of cached updates, where updates are not sent to the underlying table until the commit time. Hence no errors are reported until the commit time. No record locks are held until the user decides to commit the updates. The locks are held only during the commit process. If errors occur during the commit process, clients are given an option to abort the commit process. If clients abort a commit process, the original state of the table is restored.

The main advantage of cached updates is that the locks are held only during the commit time, thereby increasing the access time of SQL servers for other system transactions. Transactions lock out other users after record is changed, and local transactions limit the user to changing only the maximum number of records that can be locked. Cached updates avoid these problems, but permit another user to change data underneath you.

Implicitly controlling transactions

By default, C++Builder provides implicit transaction control for your applications through the Borland Database Engine (BDE). When an application is under implicit transaction control, C++Builder uses a separate transaction for each record in a dataset that is written to the underlying database. It commits each individual write operation, such as *Post* and *AppendRecord*.

Using implicit transaction control is easy. It guarantees both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop client applications in a multi-user environment, particularly when your applications run against a remote SQL server, such as Sybase, Oracle, Microsoft SQL, InterBase,® or remote ODBC-compliant databases such as Access and FoxPro, you should control transactions explicitly.

Note: You may be able to use cached updates to minimize the number of transactions you use in your applications.

Explicitly controlling transactions

There are two mutually exclusive ways to control transactions explicitly in a C++Builder database application:

- Use the methods and properties of the *TDatabase* component.
- Use passthrough SQL in a *TQuery* component. Passthrough SQL is only meaningful in the C++Builder Client/Server Suite, where you use SQL Links to pass SQL statements directly to remote SQL or ODBC servers.

The main advantage to using the methods and properties of a database component to control transactions is that it provides a clean, portable application that is not dependent on a particular database or server.

The main advantage of passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

Controlling transactions using a database component

The following table lists the methods and properties of a *TDatabase* component that are specific to transaction management, and describes how they are used:

Method or property	Purpose
Commit	Commits data changes and ends the transaction.
InTransaction	Indicates whether a transaction is in progress.
Rollback	Undoes data changes and ends the transaction.
StartTransaction	Starts a transaction.
TransIsolation	Specifies the transaction isolation level for a transaction.

StartTransaction, *Commit*, and *Rollback* are methods your application can call at runtime to start transactions, control the duration of transactions, and save or discard changes made to the database.

InTransaction is a property you can call at runtime to see if a transaction is started, but not committed or rolled back. If *true*, a transaction is started, but not completed. If *false*, a transaction is not currently in progress.

TransIsolation is a database component property that enables you to control how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Starting a transaction

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. Each statement is considered part of a group. Write changes made by any statement in the group must be successfully committed to the database, or every change made by every write statement made in the group are undone.

Grouping statements is useful when statements are dependent upon one another. Consider a bank transaction at an Automated Teller Machine (ATM). When a customer decides to transfer money from a savings account to a checking account, two changes must take place in the bank's database records:

1. The savings account must be debited.
2. The checking account must be credited.

If, for any reason, one of these actions cannot be completed, then neither action should take place. Because these actions are so closely related, they should take place within a single transaction.

To start a transaction in a C++Builder application, call a database component's *StartTransaction* method. The *StartTransaction* method begins a transaction at the isolation level specified by the *TransIsolation* property. If a transaction is currently active, an exception is raised. For a database component named DatabaseInterbase, the syntax for *StartTransaction* is:

```
DatabaseInterBase->StartTransaction();
```

All subsequent database actions take place in the context of the newly started transaction until the transaction is explicitly terminated by a subsequent call to *Commit* or *Rollback*. Modifications are not stored permanently until the *Commit* method is called. If the *Rollback* method is called to cancel changes that occurred during the session, the modifications are discarded.

How long should you keep a transaction going? Ideally, only as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit your changes.

Committing database changes

To make database changes permanent, a transaction must be committed using a database component's *Commit* method. Executing a commit statement saves database changes and ends the transaction. For example, the following statement ends the transaction started in the previous code example:

```
DatabaseInterBase->Commit();
```

Calling the *Commit* method writes all modifications since the last call to *StartTransaction* to the database and ends the current transaction. If no transaction is active, an exception is raised.

Note: *Commit* should always be attempted in a **try...catch** statement. If a transaction cannot commit successfully, you can attempt to handle the error, and perhaps retry the operation.

Discarding database changes

To discard database changes, a transaction must roll back its changes using *Rollback*. *Rollback* undoes a transaction's changes and ends the transaction. *Rollback* undoes any modifications made within the current transaction, those made since the last call to *StartTransaction*. If a transaction is not active when this method is called, an exception is raised. The following statement rolls back a transaction:

```
DatabaseInterBase->Rollback();
```

Rollback usually occurs in

- Exception handling code when you cannot recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Specifying how a transaction interacts with other simultaneous transactions

TransIsolation specifies the transaction isolation level for a database component's transactions. *Transaction isolation level* determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transaction's changes to a table. Before changing or setting *TransIsolation*, you should be familiar with transactions and transactions management in C++Builder.

The default setting for *TransIsolation* is *tiReadCommitted*. The following table summarizes possible values for *TransIsolation* and describes what they mean:

Isolation level	Meaning
tiDirtyRead	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
tiReadCommitted	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
tiRepeatableRead	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then C++Builder will use the next highest isolation level. For a detailed description of how each isolation level is implemented, see your server documentation.

TransIsolationsetting	Paradox and dBASE	Oracle	Sybase and Microsoft SQL servers	InterBase
Dirty read	Dirty read	Read committed	Read committed	Read committed
Read committed (Default)	Not supported	Read committed	Read committed	Read committed
Repeatable read	Not supported	Repeatable read (READ ONLY)	Not supported	Repeatable Read

Note: When using transactions with local Paradox and dBASE tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

Sending an SQL transaction control statement to a remote server

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To be able to use passthrough SQL to control a transaction you must

- Use the C++Builder Client/Server Suite.
- Install the proper SQL Links drivers. If you chose the Standard installation when installing C++Builder, all SQL Links drivers are already properly installed.
- Configure your network protocol correctly. See your network administrator for more information.
- Have access to a database on a remote server.
- Use the BDE Configuration utility to set SQLPASSTHROUGHMODE to NOT SHARED.

Setting SQLPASSTHROUGHMODE

SQLPASSTHROUGHMODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHROUGHMODE is set to SHARED AUTOCOMMIT. If however, you want to pass SQL transaction control statements to your server, you must use the BDE Configuration utility to set the BDE SQLPASSTHROUGHMODE to NOT SHARED. For more information about SQLPASSTHROUGH modes, see the online Help for the BDE Configuration utility.

Note: When SQLPASSTHROUGHMODE is NOT SHARED, you must use separate database components for *TQuery* components that pass SQL transaction statements to the server and those other dataset components that do not.

Transactions against a local table

The BDE supports local transactions against Paradox and dBASE tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Understanding the limitations of local transactions

The following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Transactions cannot be run against temporary tables.
- Transactions cannot be run against the BDE ASCII driver.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
- Several tables are open.
- The cursor is closed on a table to which no changes were made.

Upsizing and deploying an application

This topic addresses some of the key issues for upsizing a C++Builder database application. Upsizing means two things for C++Builder:

- Moving database sources from a local server, desktop database, or an SQL database to a database on a remote server.
- Modifying your application to address client/server considerations.

Detailed treatment of upsizing issues is beyond the scope of this section. For example, upsizing requires you to evaluate connectivity and network usage issues which are always unique to your organization and its needs. This topic, however, addresses some of the most important aspects of upsizing as they relate to a C++Builder database application, and points to some of the tools available in C++Builder that make upsizing easier.

Press the >> button to read through this topic in sequence.

Upsizing, or migrating, a database

Upsizing a database involves the following steps and considerations:

- Defining metadata on the server, based on the existing desktop database structure.
- Moving the data from the desktop to the server.
- Addressing issues such as:
- Data type differences
- Data security and integrity
- Transaction control
- Data access rights
- Data validation
- Locking differences

C++Builder provides three ways to upsize a database:

- Use the Database Desktop (DBD) to copy a table from the desktop to SQL format on a remote server. For more information about the DBD, see the DBD online Help.
- Use a TBatchMove component
- Use the Data Migration wizard, or Data Pump, to copy information from one database location to another on a one-time basis.

All of these options copy table structures and data from the desktop source to the server destination. Depending on the database, it may be necessary to change the tables created by these methods. For example, the data type mappings may not be exactly as desired.

Additionally, you must add to the database any of the following features that your application can use or that it requires

- Integrity constraints (primary and foreign keys)
- Indexes
- Check constraints
- Stored procedures and triggers
- Other server-specific features

Depending on the database, it may be most efficient to define the database structure (metadata) on the server first by using an SQL script and the server's data definition tools and then moving the data using one of the two methods previously mentioned. If you define the table structure manually, then Database Desktop and TBatchMove will copy only the data.

When upsizing a database, it is important to know the characteristics of your database server to ensure optimal performance. The characteristics of servers, such as their indexing schemes, locking mechanisms, and query optimizer can have a powerful impact on performance.

Upsizing an application

In principle, a C++Builder application designed to access local data can access data on a remote server with few changes to the application itself. This is especially true if you designed your C++Builder application and local data sources with an eye to upsizing at a later date.

If your server database mirrors your local database, upsizing your application can be as easy as changing the DatabaseName property of TTable or TQuery components in the application to point to server data.

In practice, however, there are a number of important differences between accessing local and remote data sources. Client applications accessing a remote database server must address some issues that are not relevant to desktop applications.

Any C++Builder application can use either TTable or TQuery for data access. Desktop applications frequently use TTable components. When upsizing to a SQL server, it may be more efficient to use TQuery components in some instances. For example, TQuery may be preferable for applications that retrieve large numbers of records from database tables.

If an application uses mathematical or aggregate functions, it may be more efficient to perform these functions on the server with stored procedures. The use of stored procedures may be faster because servers are typically more powerful. This also reduces the amount of network traffic required, particularly for functions that process a large number of rows.

For example, an application might need to compute the standard deviation of values of a large number of records. If this function were performed on the client, all the values would have to be retrieved from the server to the client, resulting in a lot of network traffic. If the function were performed by a stored procedure, all the computation would be performed on the server, so the application would only retrieve the answer from the server.

Moving data between databases on a one-time basis

Use the Data Migration wizard, or Data Pump, to move data (both database schema and content) between databases. Both source and target can be either a PC database or a SQL database server.

To use the data pump,

1. Create an alias for both the source and target databases by using the Borland Database Engine (BDE) Configuration Utility or SQL/Database Explorer. If using the BDE Configuration Utility, follow the directions for creating an alias. Select the driver for the source or target database and set all the parameters. For information about creating aliases, refer to the BDE Configuration Utility online Help files. If using the Database Explorer, refer to its online Help for information.
2. Select the Data Migration wizard, or Data Pump, from the C++Builder program group. In the Data Pump, select both the source and target alias. The source can be either an alias or a directory; however, SQL server databases always require an alias and may require a login.
3. Select the tables that you want to move from the source database.
4. View a preliminary report to determine how the data will appear when you move it to the target database.
5. Modify any data types, indexes, or referential integrity that are not supported on the target database.
6. Upsize the data.
7. View the final status report to determine the sequence in which data objects have been upsized, what data objects were upsized, and how they appear on the target. You can now update or modify the data directly on the target.

Before you move any data, you must understand how the data was created on the source database and how it will appear on the target database. Depending upon the structure of your data on the source database and the database objects that are supported on the target database, some modifications may be required.

Note: When moving data to a Microsoft SQL Server, be sure to set the configuration parameters MAX QUERY TIME and TIMEOUT to 20. This ensures data movement.

When moving data to a Paradox table, any indexes from the source relation will not be transferred. This is due to the design of Paradox tables, requiring a primary index prior to creating secondary indexes.

Deploying support for remote server access

Whenever you make an application available to end users, you must deploy the application. Deployment involves preparing a package for your users that enables them to run your application and access data with it. Typically, your application's installation utility handles deployment.

When you upsize your application to use a remote server, there are additional files and DLLs you need to deploy. For example, to access remote database servers, your application must use the appropriate Borland SQL Links drivers and support files or ODBC drivers. These are not part of the Borland Database Engine (BDE), and require separate installation and licensing.

For the latest information about the files you need for deploying on a remote server, see the online file DEPLOY.TXT.

Working with stored procedures

This topic describes how to use the *TStoredProc* dataset component in your Borland C++Builder database applications. Stored procedure components encapsulate stored procedures in a database on database servers such as Sybase, Microsoft SQL Server, Oracle, and InterBase.

A *stored procedure* is a set of semi-procedural statements, stored as part of a server's database metadata (like tables, indexes, and domains). A stored procedure performs a frequently-repeated database-related task on the server and passes results to a client application, such as a C++Builder database application. Stored procedures can receive input parameters from and return values to an application. The stored procedure component enables C++Builder applications to execute these stored procedures.

Often, operations that act upon large numbers of rows in database tables--or that use aggregate or mathematical functions--are candidates for stored procedures. By moving these repetitive and calculation-intensive tasks to the server, you improve the performance of your database application by

- Taking advantage of the server's usually greater processing power and speed.
- Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your C++Builder application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result--a single value representing the standard deviation--it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

Note: InterBase supports two types of stored procedures, one of which is called a select procedure because it is called within the context of a SELECT statement. To use an InterBase select procedure in C++Builder, use a *TQuery component* instead of a *TStoredProc* component.

See your server's database documentation for more information about its support for stored procedures.

Press the >> button to read through topics in sequence.

Using stored procedures

To create a stored procedure component for a stored procedure on a database server,

1. Place a stored procedure component from the Data Access page of the Component palette in a data module.
2. Set the *DatabaseName* property of the stored procedure component to the name of the database in which the stored procedure is defined. *DatabaseName* must be a BDE alias. Access the SQL Explorer from Database|Explore to view, create, and modify BDE aliases.
3. Set the *StoredProcName* property to the name of the stored procedure to use, or select its name from the drop-down list for the property.
4. Specify input parameters, if necessary, in the *Params* property. You can use the Parameters editor to set input parameters.

Note: The Parameters editor also lets you see the output parameters used by the procedure to return results to your application.

Understanding input parameters

Use input parameters to pass values from an application to a stored procedure.

Many stored procedures require you to pass them a series of input arguments, or *parameters*, to specify what and how to process. In C++Builder, the Parameters editor retrieves information about input and output parameters from the server. For some servers, all of the information required to run the stored procedure may not be accessible.

While a *TStoredProc* component encapsulates access to the stored procedure, the implementation of the stored procedure may differ from server to server. For example, the client, in this case C++Builder, requires the server to get information about input and output parameters from a system table and return this information to the client application. If the system table is in a readable format, the application can access the parameter information. If the application is in a compiled format, some of the parameter information may need to be entered manually.

In this case, you may need to enter information about the type of parameter (input, output, result) and/or the data type of the parameter to use the procedure. You will also need to enter the value that you will be passing to the stored procedure as input parameters.

The order in which the input parameters are displayed is significant, and is determined by the stored procedure definition on the server. If you are not sure of the ordering of the input and output parameters for a stored procedure, use the Parameters editor.

Setting input parameters at design time

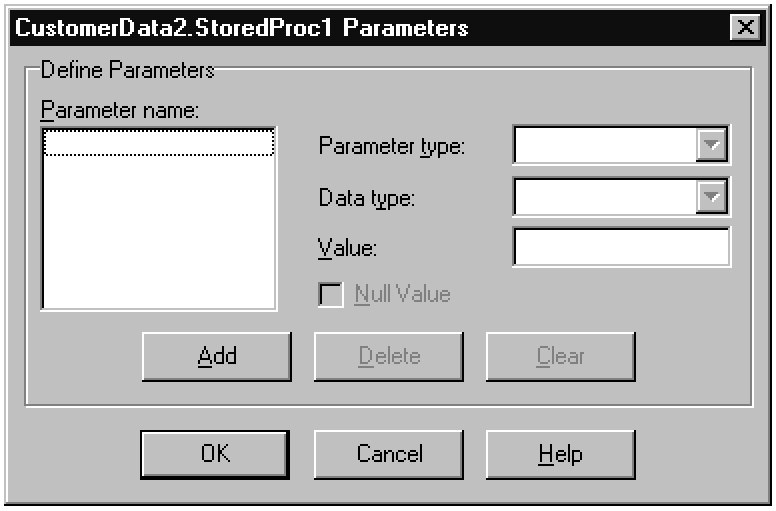
Setting parameters at runtime

Setting input parameters at design time

At design time, the easiest and safest way to view and edit input parameters is to invoke the Parameters editor. The Parameters editor lists input parameters in the correct order, and lets you assign values to them.

To invoke the Parameters editor, click the ellipsis in the *Params* property value box in the Object Inspector or

1. Select the stored procedure component.
2. Invoke the context menu.
3. Choose Define Parameters.



The Parameter name list box displays all input, output, and result parameters for the procedure. Information on input and output parameters is retrieved from the server. For some servers, not all parameter information is accessible. In this case, you must provide the missing parameter information. If you are sure that the stored procedure is valid, but no parameter names are displayed, use SQL Explorer to look for the stored procedure and inspect its text.

The Parameter type combo box describes whether a parameter selected in the list box is an input, input/output, output, or result parameter. If a server's stored procedure allows it, a single parameter may be both an input and output. If the parameter type information is missing, you must set the parameter type for it.

Note: Sybase, MS-SQL, and Informix servers generally do not return information on parameter types.

The Data type combo box lists the data type for a parameter selected in the list box. The data type can be any standard SQL data type except BLOB and arrays of data types. If the data type has not been provided from the server, you must set the data type. Some servers will also allow you to override the default data type here.

Note: Informix servers generally do not return information on data types.

Input parameters are passed by value from your application to a stored procedure. The Value edit box enables you to enter a value for a selected input parameter, and the NULL Value check box enables you to set a NULL value for the selected input parameter if its data type supports NULL values. Values should be assigned based on the declared data type. Each input parameter must be assigned either a value or a NULL value.

You can use the Add button to add parameters to a stored procedure definition. Use the Delete button to remove parameters, and the Clear button to remove all parameters from the list. If the stored procedure named in the *StoredProcName* property is valid, these buttons will be disabled because you cannot modify this data. These buttons are enabled when the stored procedure is not named, is invalid, or if C++Builder is unable to retrieve this information from the server. If you are sure that the stored procedure is valid, but no parameter names are displayed, use SQL Explorer to look for the stored procedure and inspect its text and then add parameters as appropriate.

You will not be able to add, delete, or clear parameters for servers that pass parameter information to C++Builder except if you are working with Oracle overloaded stored procedures

To signal the end of parameter definition, choose OK.

Important:

Defining parameters at design time also prepares the stored procedure for execution. A stored procedure must be prepared before it can be executed at runtime.

Setting parameters at runtime

To assign values to parameters at runtime, access the *Params* property directly. *Params* is an array of parameter strings. For example, the following code assigns the text of an edit box to the first string in the array:

```
StoredProc1->Params[0]->Items[0]->AsString = Edit1->Text;
```

You can also access parameters by name using the *ParamByName* method:

```
StoredProc1->ParamByName("Company")->AsString = Edit1->Text;
```

The *ParamBindMode* property determines how the elements of the *Params* array will be matched with stored procedure parameters. If the *ParamBindMode* property is set to *pbByName* (the default), parameters will be bound based on their names in the stored procedure. If *ParamBindMode* is set to *pbByNumber*, parameters will be bound based on the order in which they are defined in the stored procedure. Use the *pbByNumber* setting if you are building your parameters list and you don't want to use the parameter names defined in the stored procedure, or if you could not retrieve a list of parameter names from the stored procedure but know what arguments it expects.

```
StoredProc1->ParamBindMode = pbByName;
```

Preparing and executing a stored procedure

To use a stored procedure, you must prepare and execute it.

You can prepare a stored procedure at

- Design time, by choosing OK in the Parameters editor.
- Runtime, by calling the *Prepare* method of the stored procedure component.

For example, the following code prepares a stored procedure for execution:

```
StoredProc1->Prepare();
```

Note: You can prepare a stored procedure both at runtime and at design time. Input parameters are passed by value from C++Builder to the stored procedure, so if your application changes parameter information at runtime, such as when using Oracle overloaded procedures, you should prepare the procedure again.

When you execute a stored procedure, it returns either output parameters or a result set. There are two possible return types: singleton returns, which return a single value or set of values, and result sets, which return many values, much like a query.

To execute a prepared stored procedure, call the ExecProc method.

```
StoredProc1->Params[0]->Items[0]->AsString = Edit1->Text;
StoredProc1->Prepare();
StoredProc1->ExecProc();
```

Working with output parameters and result sets

A stored procedure returns values in an array of output parameters. Return values can be either a singleton result or a result set with a cursor, if the server supports it.

To access a stored procedure's output parameters at runtime, you can index into the *Params string list*, or you can use the ParamByName method to access the values. The following statements set the value of an edit box based on output parameters:

```
Edit1->Text = StoredProc1->Params[6]->Items[0]->AsString;
Edit1->Text = StoredProc1->ParamByName("Company")->AsString;
```

Note: If a stored procedure returns a result set, you may find it more useful to access and display return values in standard data-aware controls.

On some servers, such as Sybase, stored procedures can return a result set in much the same way as a query. Applications can use data aware controls to display the output of such stored procedures.

To display the results from a stored procedure in data-aware controls,

1. Place a datasource component on the data module.
2. Set the *DataSet* property of the datasource to the name of the stored procedure component from which to receive data.
3. Place a data-aware control on the form. Set the *DataSource* property of the data-aware control to the name of the datasource component.

The data-aware control can now display the results from a stored procedure when the stored procedure is active.

Working with Oracle overloaded stored procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's Overload property enables an application to specify the procedure to execute.

If Overload is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then C++Builder will execute the first stored procedure with the overloaded name; if it is two (2), it will execute the second, and so on.

Note: Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.<$paratext[.Head1]>

Working with queries

This topic describes the *TQuery* dataset component which enables you to use SQL statements to access data. It assumes you are familiar with the general discussion of datasets and data sources.

A query component encapsulates an SQL statement that is used in a client application to retrieve, insert, update, and delete data from one or more database tables. SQL is an industry-standard relational database language that is used by most remote, server-based databases, such as Sybase, Oracle, InterBase, and Microsoft SQL Server. C++Builder query components can be used with remote database servers (C++Builder Client/Server Suite only), with desktop databases such as Paradox and dBASE, and with OBDC-compliant databases such as Microsoft Access and FoxPro.

Press the >> button to read through topics in sequence.

Other places to look:

Using queries effectively

To use the query component effectively, you must be familiar with:

- SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard.
- The Borland Database Engine (BDE).

If you are an experienced desktop database developer moving to server-based applications, see "Queries for desktop developers" for an introduction to queries . If you are new to SQL, you may want to purchase one of the many fine third party books that cover SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

If you are an experienced database server developer, but are new to building C++Builder clients, then you are already familiar with SQL and your server, but you may be unfamiliar with the BDE. See "Queries for server developers" for an introduction to queries and the BDE.

Queries for desktop developers

As a desktop developer you are already familiar with the basic table, record, and field paradigm used by C++Builder and the BDE. You feel very comfortable using a TTable component to gain access to every field in every data record in a dataset. You know that when you set a table's *DataSet* property, you specify the database table to access.

Chances are you have also used a *TTable*'s range and filter properties and methods to limit the number of records available at any given time in your applications. Applying a range temporarily limits data access to a block of contiguously indexed records that fall within prescribed boundary conditions, such as returning all records for employees whose last names are greater than or equal to "Jones" and less than or equal to "Smith". Setting a filter temporarily restricts data access to a set of records that is usually noncontiguous and that meets filter criteria, such as returning only those customer records that have a California mailing address.

A query behaves in many ways very much like a table filter, except that you use the query component's SQL property (and sometime the Params property, too) to identify the records in a dataset to retrieve, insert, delete, or update. In some way a query is even more powerful than a filter because it lets you access:

- More than one table at a time (called a "join" in SQL).
- A specified subset of rows and columns in its underlying table(s), rather than always returning all rows and columns. This improves both performance and security. Memory is not wasted on unnecessary data, and you can prevent access to fields a user should not view or modify.

Queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called parameterized queries. When you use parameterized queries, the actual values currently assigned to the parameters must be inserted into the query by the BDE before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at run time.

Most often you use queries to select the data that a user should see in your application, just as you do when you use a table component. Queries, however, can also perform update, insert, and delete operations instead of retrieving records for display. When you use a query to perform insert, update, and delete operations, the query ordinarily does not return records for viewing. In this way a query differs from a table.

Queries for server developers

As a server developer you are already familiar with SQL and with the capabilities of your database server. To you a query is the SQL statement you use to access data. You know how to use and manipulate this statement and how to use optional parameters with it.

The SQL statement and its parameters are the most important parts of a query component. The query component's SQL property is used to provide the SQL statement to use for data access, and the component's Params property is an optional array of parameters to bind into the query. In C++Builder, however, a query component is much more than an SQL statement and its parameters. A query component is also the interface between your client application and the BDE.

A client application uses the properties and method of a query component to manipulate an SQL statement and its parameters, to specify the database to query, to prepare and unprepare queries with parameters, and to execute the query. A query component's methods call the BDE, which, in turn process your query requests, and communicate with the database server through an SQL Links driver. The server passes a result set, if appropriate, back to the BDE, and the BDE returns its to your application through the query component.

When you work with a query component, you should be aware that some of the terminology used to describe BDE features can be confusing at first because it has very different meanings to you as an SQL database programmer. For example, the BDE commonly uses the term "alias" to refer to a shorthand name for the path to the database server. The BDE alias is stored in a configuration file, and is set in the query component's *DatabaseName* property. (You can still use table aliases in your SQL statements.)

Similarly, the BDE help documentation, available online in \Borland\Common Files\BDE32.HLP, often refers to queries that use parameters as "parameterized queries" where you are more likely to think of SQL statements that use bound variables or parameter bindings.

Note: BDE terminology is used throughout the help system because you will encounter it throughout Borland's documentation. Whenever confusion may result from using BDE terms, however, explanations are provided.

What databases can you access with a query component?

A TQuery component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL supports the SQL-92 standard with the exception of some DDL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase (C++Builder Client/Server only). You must have installed the appropriate SQL Link to access a remote server. Any standard SQL syntax supported by these server is allowed. For information on SQL syntax, limitations, and extensions, see your server documentation.

C++Builder also supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you create a heterogeneous query, the BDE uses Local SQL to process the query.

Using a query component: an overview

To use a query component in an application, follow these general steps at design time:

1. Place a query component from the Data Access tab of the Component palette in a data module, and set its *Name* property appropriately for your application.
2. Set the *DatabaseName* property of the component to the name of the database to query. *DatabaseName* can be a BDE alias or, for local database access, an explicit directory path and database name. If your application uses a database component, then you can set *DatabaseName* to the name of a local BDE alias defined in that database component's *DatabaseName* property.
3. Specify an SQL statement in the SQL property of the component, and optionally specify any parameters for the statement in the Params property.
4. Place a data source component from the Data Access tab of the Component palette in the data module, and set its *DataSet* property to the name of the query component. The data source module is used to return the results of the query (called a result set) from the query to data-aware components for display.

To execute a query for the first time at runtime, follow these general steps:

1. Close the query component.
2. Provide an SQL statement in the *SQL* property either because you did not set the *SQL* property at design time, or because you want to change the SQL statement already provided. To use the design-time statement as is, skip this step.
3. Set parameters and parameter values in the *Params* property either directly, or by using the *ParamByName* method. If a query does not contain parameters, or the parameters set at design time are unchanged, skip this step.
4. Call Prepare to initialize the BDE and bind parameter values into the query. Calling *Prepare* is optional, though highly recommended.
5. Call Open for queries that return a result set, or call ExecSQL for queries that do not return a result set.

After you execute a query for the first time, then as long as you do not modify the SQL statement, an application can repeatedly close and reopen or reexecute a query without repreparing it.

Setting the SQL property

Use the *SQL* property to specify the SQL query statement to execute. At design time a query is prepared and executed automatically when you set the query component's *Active* property to *true*. At run time, a query is prepared with a call to Prepare, and executed when the application calls the component's *Open* or *ExecSQL* methods.

The *SQL* property is a *TString* object, which is an array of text strings and a set of properties, events, and methods that manipulate them. The strings in *SQL* are automatically concatenated to produce the SQL statement to execute. You can provide a statement in as few or as many separate strings as you desire. One advantage to using a series of strings is that you can divide the SQL statement into logical units (for example, putting the WHERE clause for a SELECT statement into its own string), so that it is easier to modify and debug a query.

The SQL statement can be a query that contains hard-coded field names and values, or it can be a parameterized query that contains replaceable parameters that represent field names and values that must be bound into the statement before it is executed. For example, this statement is hard-coded:

```
SELECT * FROM CUSTOMER WHERE CUSTNO = 1231
```

Hard-coded statements are useful when applications execute exact, known queries each time they run. At design time or runtime you can easily replace one hard-code query with another hard-coded or parameterized query as needed. Whenever the SQL property is changed the query is automatically closed and unprepared.

Note: When column names in a query contain spaces or special characters, the column name must be enclosed in quotes and must be preceded by a table reference and a period. For example, BIOLIFE."Species Name".

A parameterized query contains one or more placeholder parameters, application variables that stand in for field names or comparison values such as those found in the WHERE clause of a SELECT statement. Using parameterized queries enables you to change the value without rewriting the application. Parameter values must be bound into the SQL statement before it is executed for the first time. C++Builder does this automatically for you if you do not explicitly call the *Prepare* method before executing a query.

This statement is a parameterized query:

```
SELECT * FROM CUSTOMER WHERE CUSTNO = :Number
```

The variable Number, indicated by the leading colon, is a parameter that fills in for a comparison value that must be provided at run time and that may vary each time the statement is executed. The actual value for *Number* is provided in the query component's *Params* property.

Tip:

It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names ensures that if a query uses its *DataSource* property to provide values for parameters, it can match the variable name to valid field names.

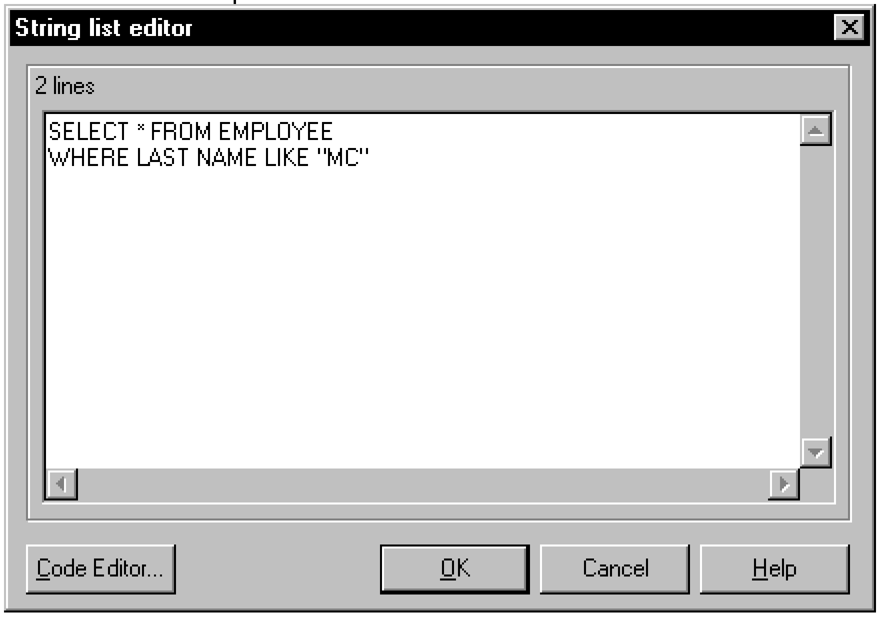
Specifying the SQL property at design time

Specifying the SQL property at runtime

Specifying the SQL property at design time

You can specify the *SQL* property at design time using the String List editor. To invoke the String List editor for the *SQL* property,

- Double-click on the *SQL* property value column, or
- Click its ellipsis button.



```
SELECT * FROM EMPLOYEE
WHERE LAST NAME LIKE "MC"
```

You can enter an SQL statement in as many or as few lines as you want. Entering a statement on multiple lines, however, makes it easier to read, change, and debug. Choose OK to assign the text you enter to the *SQL* property.

Normally, the SQL property can contain only one complete SQL statement at a time, although these statements can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some servers also support "batch" syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements in the *SQL* property.

Note: With C++Builder Client/Server Suite, you can also use the Visual Query Builder to construct a query based on a visible representation of tables and fields in a database. To use the Visual Query Builder, select a query component, right-click it to invoke the context menu, and choose Query Builder. To learn how to use the Query Builder, open it and see to its online Help.

Specifying the SQL property at runtime

There are two ways to set the *SQL* property at runtime. An application can set the *SQL* property directly, or it can call the *SQL* property's *LoadFromFile* method to read an SQL statement from a file.

Setting the SQL property directly

To set the *SQL* property at run time,

1. Call *Close* to inactive the query. It is always safe to call *Close* even for queries that are currently inactive, and you should call *Close* so that you can reopen or reexecute the query with its new SQL statement.
2. Call the *Clear* method for the *SQL* property to delete its current SQL statement.
3. Call the *Add* method for the *SQL* property to insert and append one or more strings to the *SQL* property to create a new SQL statement.
4. Call *Open* or *ExecSQL* to execute the query.

The following code illustrates these steps:

```
CustomerQuery->Close(); // Close the query if it's active
CustomerQuery->SQL->Clear(); // Delete the current SQL statement, if there is one
CustomerQuery->SQL->Add("SELECT * FROM ORDERS");
CustomerQuery->SQL->Add("WHERE COMPANY = 'Sight Diver'");
CustomerQuery->Open();
```

Note: If a query uses parameters, you should also set their initial values and call the Prepare method before opening or executing a query.

Loading the SQL property from a file

You can also use the LoadFromFile method to assign an SQL statement in a text file to the SQL property. For example,

```
CustomerQuery->Close();
CustomerQuery->SQL->Clear();
CustomerQuery->SQL->LoadFromFile("C:\ORDERS.TXT");
CustomerQuery->Open();
```

Note: If the SQL statement contained in the file is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query.

Setting parameters

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace column names or data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO COUNTRY (NAME, CAPITAL, POPULATION)
 VALUES (:name, :capital, :population)
```

In this SQL statement, *:name*, *:capital*, and *:population* are placeholders for actual values supplied to the statement at run time by your application. Before a parameterized query is executed for the first time, your application should call the Prepare method to bind the current values for the parameters to the SQL statement. Binding means that the BDE and the server allocate resources for the statement and its parameters that improve the execution speed of the query.

Supplying parameters at design time

Supplying parameters at runtime

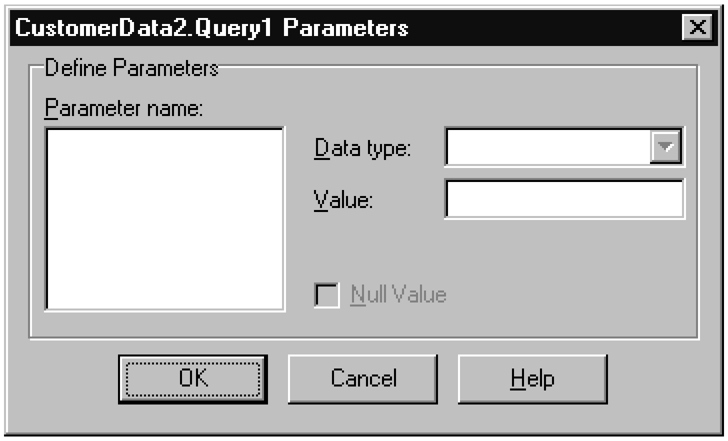
Using a data source to bind parameters

Supplying parameters at design time

At design time, the easiest and safest way to enter query parameters is to invoke the Query Parameters editor. The Query Parameters editor lists parameters in the correct order and lets you assign values to them. When you define parameters at design time, your query is automatically reprepared for you.

To invoke the Query Parameters editor,

1. Select the query component.
2. Right-click the component to invoke the context menu.
3. Choose Define Parameters.



For queries without previously defined parameters, the Parameter name list box is empty. If parameters are already defined for a query, then the parameter names are displayed in the list box.

The Data type combo box lists the BDE data type for a parameter selected in the list box. You must set a data type for each parameter. In general, BDE data types conform to server data types. For specific BDE-to-server data type mappings, see the BDE help in \Borland\Common Files\BDE32.HLP.

The Value edit box enables you to enter a value for a selected parameter, and the Null Value check box enables you set a null value for the selected parameter if its data type permits null values. You must enter a value for each parameter, even if that value is null.

To signal the end of parameter definition, choose OK.

Supplying parameters at runtime

To create parameters at run time, you can use the

- ParamByName method to assign values to parameters based on the parameters' names.
- Params property to assign values to parameters based on the parameters' ordinal location.

For example, the following code uses *ParamByName* to assign the text of an edit box to the *Company* field:

```
Query1->ParamByName("Company")->AsString = Edit1->Text;
```

The same code can be rewritten using the *Params* property (it assumes that *Company* is the first parameter in the query):

```
Query1->Params->Items[0]->AsString = Edit1->Text;
```

Assigning values to parameters based on parameter name

ParamByName is a function that enables an application to assign values to parameters based on the parameters' names. This is better and safer than using the *Params* property directly because you don't have to know the actual order of the parameters to which you assign values. Instead of providing the ordinal location of a parameter as you must when indexing directly into the array of strings in the *Params* property, you pass the parameter name as an argument to *ParamByName*.

For example, suppose a query component named CountryQuery has the following statement for its SQL property:

```
INSERT INTO COUNTRY (NAME, CAPITAL, POPULATION)
 VALUES (:name, :capital, :population)
```

The following statements assign parameter values to a set of specified parameters:

```
CountryQuery->ParamByName("name")->AsString = "Lichtenstein";
CountryQuery->ParamByName("capital")->AsString = "Vaduz";
CountryQuery->ParamByName("population")->AsInteger = 420000;
```

Assigning values to parameters based on parameter order

When you create a parameterized query, C++Builder creates an array of parameter object for the query in the *Params* property. Params is a zero-based array of TParam objects with an element for each parameter in the query. The first parameter is Params[0], the second Params[1], and so on.

For example, given this query statement:

```
INSERT INTO COUNTRY (NAME, CAPITAL, POPULATION)
 VALUES (:name, :capital, :population)
```

then an application could specify values for each of these parameters as follows:

```
CountryQuery->Params->Items[0]->AsString = "Lichtenstein";
CountryQuery->Params->Items[1]->AsString = "Vaduz";
CountryQuery->Params->Items[2]->AsInteger = 420000;
```

These statements bind the value "Lichtenstein" to the *:name* parameter, "Vaduz" to the *:capital* parameter, and 420000 to the *:population* parameter.

Using a data source to bind parameters

If parameter values for a parameterized query are not bound at design time, C++Builder attempts to supply values for them based on the query component's *DataSource* property. *DataSource* specifies a different table or query component that C++Builder can search for field names that match the names of unbound parameters. This search dataset must be created and populated before you create the query component that uses it. If matches are found in the search dataset, C++Builder binds the parameter values to the values of the fields in the current record pointed to by the data source.

You can create a simple application to understand how to use the *DataSource* property to link a query in a master-detail form. Suppose the data module for this application is called, *LinkModule*, and that it contains a query component called *OrdersQuery* that has the following SQL property:

```
SELECT CustNo, OrderNo, SaleDate
 FROM Orders
 WHERE CustNo = :CustNo
```

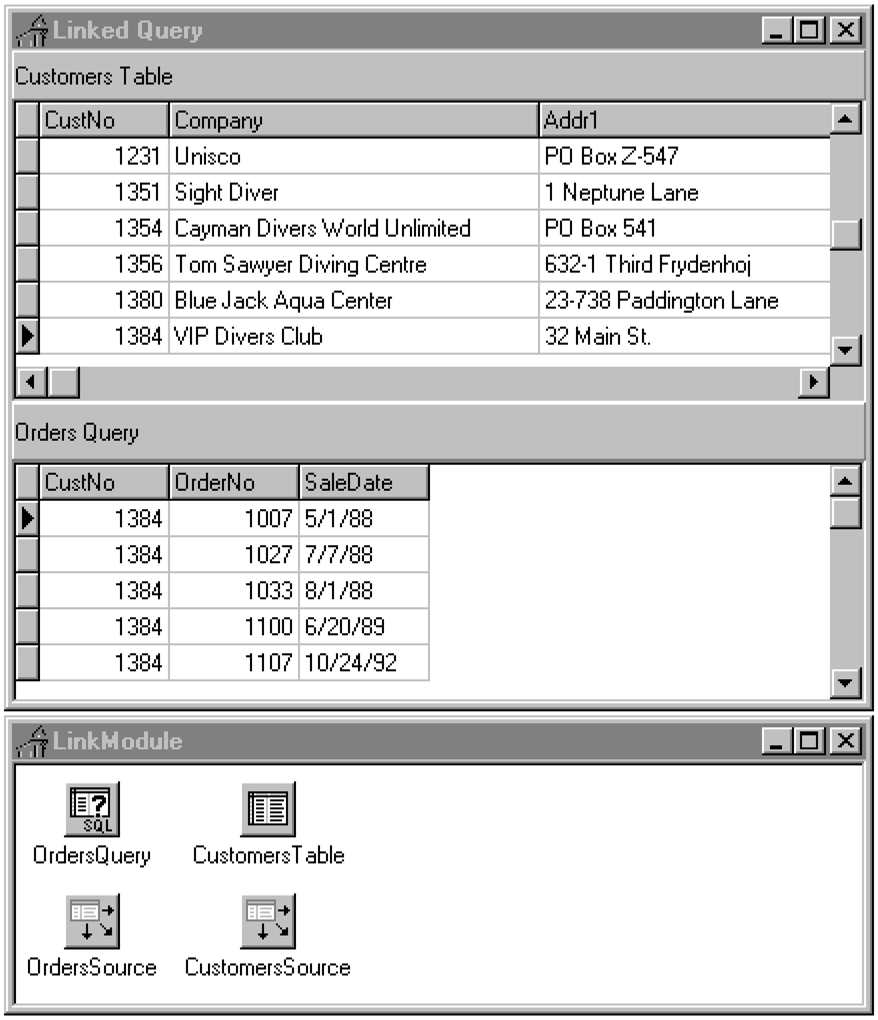
The *LinkModule* data module also contains

- A TTable dataset component named *CustomersTable* linked to the CUSTOMER.DB table.
- A TDataSource component named *OrdersSource*. The DataSet property of *OrdersSource* points to *OrdersQuery*.
- A TDataSource named *CustomersSource* linked to *CustomersTable*. The *DataSource* property of the *OrdersQuery* component is also set to *CustomersSource*. This is the setting that makes *OrdersQuery* a linked query.

Suppose, too, that this application has a form, named *LinkedQuery*. This form contains two data grids, a *Customers Table* grid linked to *CustomersSource*, and an *OrdersQuery* grid linked to *OrdersSource*.

The following figure illustrates how this application appears at design time.

Sample master/detail query form and data module at design time


Linked Query

Customers Table

CustNo	Company	Addr1
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.

Orders Query

CustNo	OrderNo	SaleDate
1384	1007	5/1/88
1384	1027	7/7/88
1384	1033	8/1/88
1384	1100	6/20/89
1384	1107	10/24/92

LinkModule

OrdersQuery CustomersTable

OrdersSource CustomersSource

Note: If you build this application, create a table component and its data source before creating the query component.

If you compile this application, at run time the *:CustNo* parameter in the SQL statement for *OrdersQuery* is not assigned a value, so C++Builder tries to match the parameter by name against a column in table pointed to by *CustomersSource*. *CustomersSource* gets its data from *CustomersTable*, which, in turn, derives its data from the CUSTOMER.DB table. Because CUSTOMER.DB contains a column called "CustNo," the value from the *CustNo* field in the current record of the *CustomersTable* dataset is assigned to the *:CustNo* parameter for the *OrdersQuery* SQL statement. The grids are linked in a master-detail relationship. At run time, each time you select a different record in the Customers Table grid, the *OrdersQuery* SELECT statement executes to retrieve all orders based on the current customer number.

Executing a query

After you specify an SQL statement in the SQL property, and after you set any parameters for the query, you can execute the query. When a query is executed, the BDE receives and processes SQL statements from your application.

Note: Before you execute a query for the first time, you may want to call the Prepare method to improve query performance. Prepare initializes the BDE and the database server, each of which preallocates system resources for the query.

You can execute both static and dynamic SQL statements at design time and at runtime.

Executing a query at design time

To execute a query at design time, set the query component's *Active* property to *true*.

The results of the query, if any, are displayed in any data-aware controls associated with the query component.

Executing a query at runtime

To execute a query at runtime, use one of the following methods:

- *Open* executes a query that returns a result set, such as SELECT.
- *ExecSQL* executes a query that does not return a result set, such as INSERT, UPDATE, or DELETE.

Note: If you do not know at design time whether a query will return a result set at run time, code both types of query execution statements in a **try...catch** block. Put a call to the *Open* method in the **try** clause, and put the *ExecSQL* method in the **catch** clause. To avoid executing a statement twice (for example, in the case of an empty result set), you should test for an empty result set before executing the *ExecSQL* method in the **catch** clause.

Executing a query that returns a result set

To execute a query that returns a result set, such as a SELECT statement, follow these steps:

- Call *Close* to ensure that the query is not already open. If a query is already open you cannot open it again without first closing it. Closing a query and reopening it fetches a new version of data from the server.
- Call *Open* to execute the query.

For example,

```
CustomerQuery->Close();
CustomerQuery->Open(); // Returns a result set
```

For information on navigating within a result set, see "Disabling bidirectional cursors". For information on editing and updating a result set, see "Working with result sets".

Executing a query without a result set

To execute a query that does not return a result set, such as INSERT, UPDATE, or DELETE, follow these steps:

- Call *Close* to ensure that the query is not already open. If a query is already open you cannot open it again without first closing it. Closing a query and reopening it fetches a new version of data from the server.
- Call *ExecSQL* to execute the query.

For example,

```
CustomerQuery->Close();
CustomerQuery->ExecSQL(); // Does not return a result set
```

Preparing a query

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the BDE for parsing, resource allocation, and optimization. The BDE, in turn, notifies the database server to prepare for the query. The server, too, may allocate resources for the query. These operations improve query performance, making your application faster.

If you do not prepare a query before executing it, then C++Builder prepares it for you each and every time you call Open or ExecSQL. For queries that are executed only once this is fine. For parameterized queries that are executed many times, such as an INSERT statement, query performance can be significantly improved if an application explicitly prepares the query itself.

You can prevent C++Builder from repreparing a repeatedly executed query by calling the query component's *Prepare* method once before first opening or executing the query. For example,

```
CustomerQuery->Close;
if (!CustomerQuery->Prepared)
 CustomerQuery->Prepare();
CustomerQuery->Open;
```

This example checks the query component's *Prepared* property to determine if a query is already prepared. If not, it calls the *Prepare* method before calling *Open*.

Unpreparing a query to release resources

The *UnPrepare* method sets the Prepared property to *false*. *UnPrepare*

- Ensures that the SQL property is reprepared prior to executing it again.
- Notifies the BDE to release the internal resources allocated for the statement.
- Notified the server to release any resources it has allocated for the statement.

To unprepare a query,

```
CustomerQuery->UnPrepare();
```

Note: When you change the text of the *SQL* property for a query, C++Builder automatically closes and unprepares the query.

Creating heterogenous queries

C++Builder supports *heterogeneous queries*, queries made against tables in more than one database. A heterogeneous query may join tables on different servers, and even different types of servers. For example, a heterogeneous query might involve a table in a Oracle database, a table in a Sybase database, and a local dBASE table. When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL, so extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query, follow these steps:

1. Define a BDE standard alias that references a local directory, and set the *DatabaseName* property of the query component to that alias. To define BDE aliases use the Database Explorer.
2. Define separate BDE aliases for each database accessed in the query.
3. Specify the SQL statement to execute in the SQL property. Precede each table name in the SQL statement with the BDE alias for the database where that table can be found.
4. Set any parameters for the query in the Params property.
5. Call Prepare to prepare the query for execution prior to executing it for the first time.
6. Call Open or ExecSQL depending on the type of query to execute.

For example, suppose you define an alias called Oracle1 for an Oracle database that has a CUSTOMER table, and Sybase1 for a Sybase database that has an ORDERS table. A simple query against these two tables would be

```
SELECT CUSTOMER.CUSTNO, ORDERS.ORDERNO
FROM '':Oracle1:CUSTOMER'', '':Sybase1:ORDERS''
```

Improving query performance

Following are steps you can take to improve query execution speed:

- Set a query's UniDirectional property to *true* if you do not need to navigate backward through a result set (SQL-92 does not, itself, permit backward navigation through a result set). By default, *UniDirectional* is *false* because the BDE supports bidirectional cursors by default.
- Prepare the query before execution. This is especially helpful when you plan to execute a single query several times. You need only prepare the query once, before its first use.

Disabling bidirectional cursors

The *UniDirectional* property determines whether or not BDE bidirectional cursors are enabled for a query in C++Builder. When a query returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source.

UniDirectional is *false* by default, meaning that the cursor for a result set can navigate both forward and backward through its records. Bidirectional cursor support requires some additional processing overhead, and can slow some queries. To improve query performance, you may be able to set *UniDirectional* to *true*, restricting a cursor to forward movement through a result set.

If you do not need to be able to navigate backward through a result set, you can set *UniDirectional* to *true* for a query. Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if (!CustomerQuery->Prepared)
{
 CustomerQuery->UniDirectional = true;
 CustomerQuery->Prepare();
}
CustomerQuery->Open(); // Returns a result set with a one-way cursor
```

Working with result sets

By default, the result set returned by a query is read-only. Your application can display field values from the result set in data-aware controls, but users cannot edit those values. To enable editing of a result set, your application must request a "live" result set.

Enabling editing of a result set

Local SQL syntax requirements for a live result set

Remote server SQL syntax requirements for a live result set

Enabling editing of a result set

To request a result set that users can edit in data-aware controls, set a query component's *RequestLive* property to *true*. Setting *RequestLive* to *true* does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether or not a query uses the local SQL parser or a server's SQL parser. Heterogeneous joins and queries executed against Paradox or dBASE are parsed by the BDE using local SQL. Queries against a remote database server are parsed by the server.

If an application requests and receives a live result set, C++Builder sets the *CanModify* property for the query component to *true*.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for pass-through SQL queries made against a remote server.

Local SQL syntax requirements for a live result set

For queries that use the local SQL parser, the BDE offers expanded support for updatable, live result sets for both single table and multi-table queries. The local SQL parser is used when a query is made against one or more dBASE or Paradox tables, or one or more remote server tables when those table names in the query are preceded by a BDE database alias. The following sections describe the restrictions that must be met to return a live result set for local SQL.

Restrictions on live queries

A live result set for a query against a single table or view is returned if the query does not contain any of the following:

- JOIN, UNION, INTERSECT or MINUS clauses.
- A DISTINCT clause in the SELECT statement.
- Aggregate functions.
- Base tables or views that are not updatable.
- GROUP BY or HAVING clauses.
- Subqueries.
- ORDER BY clauses not based on an index.

Restrictions on live joins

A live result set for a join is returned if:

- Only two tables are involved in the join.
- All joins are left-to-right outer joins.
- All joins on Paradox and dBASE tables can be satisfied by existing indexes.
- Output ordering is not defined.
- Each table in the join is a base table.
- All restrictions that apply to live queries are also met.

Remote server SQL syntax requirements for a live result set

For queries that use passthrough SQL, which includes all queries made solely against remote database servers, live result sets are restricted to the standards defined by SQL-92 and any additional, server-imposed restrictions.

SQL-92 restrictions that must be met in order to return a live result set are as follows. A query cannot contain

- A DISTINCT clause in the SELECT statement.
- Aggregate functions
- References to more than one base table or updatable view.
- GROUP BY or HAVING clauses.
- Subqueries that reference the table in the FROM clause.
- Correlated subqueries.

Restrictions on updating a live result set

If a query returns a live result set, you may not be able to update the result set directly if the result set contains linked fields or you switch indexes before attempting an update. If these conditions exist, you may be able to treat the result set as a read-only result set, and update it accordingly.

Updating a read-only result set

Applications can update data returned in a read-only result set if they are using cached updates. To update a read-only result set associated with a query component:

1. Add a TUpdateSQL component to the data module in your application to essentially give you the ability to post updates to a read-only dataset.
2. Enter the SQL update statement for the result set to the update component's ModifySQL, InsertSQL, or DeleteSQL properties.
3. Set the query component's *CachedUpdate* property to *true*.

Working with tables

This topic describes how to use the TTable dataset component to support the use of complex tables in your database applications. A table component inherits many of its fundamental properties and methods from *TDataSet*. Therefore, you should be familiar with the general discussion on datasets before reading about the unique properties and methods of table components.

Press the >> button to read through topics in sequence.

Other places to look:

Accessing data in databases

Working with queries

Working with stored procedures

What are table components?

A table component is the most fundamental and flexible dataset component class in C++Builder. It gives you access to every row (record) and column (field) in an underlying database table, whether it is from Paradox, dBASE, an ODBC-compliant database such as Microsoft Access, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

A *TTable* component retrieves data from a physical database table and provides live access to the database table via the Borland Database Engine (BDE) to one or more data-aware controls (such as *TDBGrid*) through a TDataSource component. A *TTable* component also sends data received from a data-aware component to a physical database via the BDE.

Table components let you

- View and edit data in every column and row of a table
- Work with a range of rows in a table
- Search for records in a table
- Copy or delete a table
- Sort records in a table
- Create master-detail relationships
- Filter records in a table based on criteria you specify

Choosing between table and query components

Choosing between table and query components

Table components are full-featured, flexible, and easily used data access components that are sufficient for many database applications. They differ from query components in two significant ways:

- A query component can access more than one table at a time. A table component can access only one table at a time.
- A query component can access a subset of rows and columns in its underlying table(s). A table component can restrict row access by setting ranges and/or filters.

If you have worked with databases previously, you are probably familiar with the table model already. Query components offer different capabilities from table components and are useful when you need to

- Query data in multiple tables to produce a single result set (this operation is called a "join").
- Restrict data access to a subset of rows and columns across tables.
- Write applications that require SQL syntax for compatibility with other SQL databases or applications.

Accessing all data with a table component

The following steps are the general steps for accessing all or your data through a table component. You may need to set additional properties due to application requirements. To create and use a table component at design time,

1. Place a table component from the Data Access tab of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the DatabaseName of the component to the location of the database to access. This can be a defined BDE alias or an explicit directory path to the database.
3. Set the TableName property to the name of the table in the database. You can select tables from the drop-down list if the *DatabaseName* property is already specified.
4. Optionally specify the TableType of the underlying database table.
5. Place a data source component from the Data Access tab of the Component palette in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.
6. Place a data-aware control (such as *TDBGrid*) from the Data Controls tab of the Component palette on the form, and set its *DataSource* property to the name of the data source component. The data-aware component is used to display data from the table on a form.
7. Set the table component's *Active* property to *true* to view data in the data-aware control.

Specifying a database table

The DatabaseName property specifies where C++Builder looks for a database table. *DatabaseName* can be a defined Borland Database Engine (BDE) alias, or an explicit directory path to the database. For Paradox and dBASE tables, an explicit specification is just a directory path; for SQL tables, you must use a BDE alias.

Hint:

You can add, delete, and modify aliases by running the BDE Configuration Utility from the C++Builder program group and selecting the Aliases tab.

The advantage of setting *DatabaseName* to a BDE alias is that you can change the data source for an entire application by simply changing the alias definition in the BDE Configuration Utility. For more information on using the BDE Configuration Utility, see its online Help. For more information on the DatabaseName property, see the online VCL Reference.

If your application uses database components to control database transactions, set *DatabaseName* to a local alias defined for the database component instead.

Note: You cannot set *DatabaseName* if the table component's *Active* property is *true*. To set or change *DatabaseName*, set *Active* to *false* or use the *Close* method to put a dataset in Inactive state before changing the *DatabaseName* property.

Specifying a table name

The *TableName* property specifies the table in a database with which a table component is associated. Before you set *TableName*, set the DatabaseName property. When *DatabaseName* is set at design time, you can choose a table from the drop-down list for the *TableName* property.

At runtime, you can set or change the table associated with a table component by

1. Closing the table by calling its Close method or setting its Active property to *false*.
2. Assigning a valid table name to the *TableName* property.

For example, the following code changes the table name for the *OrderOrCustTable* table component based on its current table name:

```
OrderOrCustTable->Active = false; //close the table
if (OrderOrCustTable->TableName == "CUSTOMER.DB")
 OrderOrCustTable->TableName = "ORDERS.DB";
else
 {
   OrderOrCustTable->TableName = "CUSTOMER.DB";
   OrderOrCustTable->Active = true; //Reopen with a new table
 }
```

Specifying a table type

TableType specifies the type of the underlying database table. The *TTable* component must be closed or the *Active* property must be *false* in order to change the *TableType* property. If TableType is set to ttDefault (the default), the table's file extension determines the table type:

Extension	Table type
.DB or no extension	Paradox table (*ttParadox*)
.DBF	dBASE table (*ttDBase*)
.TXT	ASCII table (*ttASCII*)

If the value of TableType is not ttDefault, then the table is always the specified TableType, regardless of file-name extension.

Note: This property is not used for SQL table access.

Opening and closing a table

To view data in a data-aware control, such as *TDBGrid*, open a table and put it in Browse state by setting its *Active* property to *true* or by using the *Open* method. To close a dataset and put it in Inactive state, set its *Active* property to *false* or call its *Close* method.

Note: *Post* is not called implicitly by setting the *Active* property to *false*. Use the *BeforeClose* event to post any pending edits explicitly or they will be lost.

Opening and closing datasets

Creating a table programmatically

You can use the *CreateTable* method to create a table programmatically. The following shows how to create a table programmatically.

```
{TTable* Table1;
Table1=new TTable(this);
Table1->Active = false;
Table1->DatabaseName = "DBDemos";
Table1->TableName = "Testtbl.db";
Table1->TableType = ttParadox;
Table1->FieldDefs->Clear();
Table1->FieldDefs->Add("Field1", ftInteger, 0, false);
Table1->FieldDefs->Add("Field2", ftInteger, 0, false);
Table1->IndexDefs->Clear();
TIndexOptions MyIndexOptions;
MyIndexOptions << ixPrimary << ixUnique;
Table1->IndexDefs->Add("Field1Index", "Field1", MyIndexOptions);
Table1->CreateTable();}
```

Caution: The CreateTable method will overwrite an existing table without warning. Add an exception handler to determine if the table currently exists.

Controlling access to a table

By default, the *ReadOnly* property for a table component is set to *false*. When the table is opened, therefore, it requests read and write privileges for the underlying database table. Depending on the characteristics of the underlying table, the requested write privilege may not be granted (for example when you request write access to an SQL table on a remote server for which you only have read access).

After you open a table at runtime, your application examines the table's *CanModify* property to test whether or not the server has granted write access to the database. If *CanModify* is *false*, the application cannot write to the database and the dataset cannot be put into Edit or Insert state. If *CanModify* is *true*, your application can write to the database unless other factors interfere, such as SQL access privileges.

Note: Set the *Active* property to *false* or call the *Close* method before changing the *ReadOnly* property.

Caution: When the *ReadOnly* property is set to true, the *CanModify* property is automatically set to *false*. Conversely, when the *ReadOnly* property is set to false, the *CanModify* property is automatically set to *true* if the database allows read and write privileges for the dataset and the underlying table.

To gain sole read/write access to a Paradox or dBASE table, set a table component's *Exclusive* property to *true* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. If other users are accessing the table when you try to open it, your exception handler will have to wait for those users to release it. If you do not provide an exception handler and another user already has the table open, your application will be terminated. Set *Exclusive* to *true* only when you must have complete control over the table.

Note: Set the *Active* property to *false* before changing the *Exclusive* property to prevent an exception. Do not set both *Active* and *Exclusive* to *true* in the Object Inspector. Since the Object Inspector will have the table open, that will prevent your program from opening it.

The following statements open a table for exclusive access, providing an exception handler if another user is accessing the table:

```
CustomersTable->Active = false; //Close the table
CustomersTable->Exclusive = true; //Set request for exclusive lock
try
 {CustomersTable->Active = true;} //Now open the table
catch (EDatabaseErrorix)
 {...}
```

Note: You can attempt to set *Exclusive* on SQL tables, but some servers may not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

Searching for records

You can search a table for specific records using the generic search methods *Locate* and *Lookup* or the *Find* methods *FindFirst*, *FindLast*, *FindNext*, and *FindPrior*. These methods enable you to search any type of columns in any table, whether or not they are indexed or keyed.

- Locate moves the cursor to the first row matching a specified set of criteria
- Lookup returns the values from the first row that matches specified search criteria, but does not move the cursor to that row.
- Find methods are used with filters and move the cursor to the first row matching the search criteria when filtering is not enabled.
- Goto search methods enable you to search for a record based on indexed fields, referred to as a *key*, and make the first record found the new current record.

Moving the cursor to the first row matching search criteria

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-sensitive or if it can use partial-key matching.

Returning values from rows that match search criteria

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it. In its simplest form, you pass *Lookup* the name of column to search, the field value to match, and the field or fields to return.

Moving among records that match filter criteria

Find methods used with filters provide the ability to go to the first, last, next, and previous records based on filter criteria when the dataset is not currently being filtered. When filtering is not enabled, the methods temporarily set the *Filtered* property to *true* then use the code in either the *OnFilterRecord* event handler or the condition in the *Filter* property to find the record. To locate records that match the filter, call the *FindFirst*, *FindLast*, *FindNext*, or *FindPrior* methods.

Searching for and moving to records based on indexed fields

Although using *Locate* to search for data meeting certain criteria is the preferred method for conducting a search due to better performance, table components also support a set of *Goto* search methods that enable you to search for a record based on indexed fields, referred to as a *key*, and make the first record found the new current record.

For Paradox and dBASE tables, the key must always be an index, which you can specify in a table component's *IndexName* property. For SQL tables, the key can also be a list of fields you specify in the *IndexFieldNames* property. You can also specify a field list for Paradox or dBASE tables, but the fields must have indexes defined on them.

Tip:

To search nonindexed fields in a Paradox or dBASE table, use *Locate*. Alternatively, you can use a TQuery component and a SELECT statement to search nonindexed fields in Paradox and dBASE fields.

Note: You can create an index "on the fly" in C++Builder by using the *AddIndex* method.

There are six *Goto* methods:

Method	Description
SetKey	Clears the existing elements from the search key buffer and puts a table into *dsSetKey* state, so your application can search for values in database tables.
EditKey	Modifies the contents of the search key buffer and puts a table into *dsSetKey* state while preserving previous search values in the *Fields* property. This method is useful only when searching on multiple fields after calling *SetKey*. For more information, see "Repeating or extending a search".
GotoKey	Searches for the first record in a dataset that exactly matches the values in the *Fields* property, moves the cursor to that record if one is found, and returns *true*. If the record is not found, the cursor is not moved, and the function returns *false*.
GotoNearest	Searches for the closest match to a record based on the index fields that are greater than or equal to the *IndexFields* property, and moves the cursor to that record. *GotoNearest* only works with fields of string data type. The search begins at the first record in the table, not at the current cursor position.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single function.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single function. *FindNearest* only works with string data types.

To execute a search, follow these steps:

1. If you are not using a table's primary index, specify the index to use for the search in *IndexName* or, for SQL tables, list the fields to use as a key in *IndexFieldNames*. If you use a table's primary index, you do not need to set these properties.
2. Open the table by setting the *Active* property to *true* or by calling the dataset's *Open* method.
3. Put the table in *SetKey* state with *SetKey*.
4. Specify the value(s) to search on in *Fields*. *Fields* is a string list that you index with ordinal numbers corresponding to each column. The first column number in a table is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, moves to the first record containing a field value that exactly matches the text in an edit box on a form:

```
void __fastcall TForm1::CustTableCancelBtnClick(TObject *Sender)
{
 Table1->SetKey();
 Table1->Fields[0]->AsString = Edit1->Text;
 if (!Table1->GotoKey())
   ShowMessage("Record not found");
}
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for columns of string data type. For example,

```
Table1->SetKey();
Table1->Fields[0]->AsString = "Sm";
Table1->GotoNearest();
```

If a record exists with Sm as the first two characters, the cursor is positioned on that record. If such a record does not exist, the cursor is positioned on the record immediately following where the cursor would have been positioned if there had been a match.

The *FindKey* and *FindNearest* methods each take a single argument: a comma-delimited array of values, where each value corresponds to an index column in the underlying table. Values can be literals, variables, or null. If the number of values in the argument is less than the number of columns in the underlying table, the remaining values are assumed to be null.

For example, if *Table1* is indexed on its first column, then the following *FindKey* statement

```
Table1->FindKey(new TVarRec(Edit1->Text), vtPChar);
```

performs the same function as next statements:

```
Table1->SetKey();
Table1->FieldByName("Company")->AsString = Edit1->Text;
Table1->GotoKey();
```

Similarly, the following *FindNearest* statement

```
Table1->FindNearest(new TVarRec(Edit1->Text), vtPChar);
```

performs the same function as these statements:

```
Table1->SetKey();
Table1->FieldByName("Company")->AsString = Edit1->Text;
Table1->GotoNearest();
```

Both *Find* functions work by default on the primary index column. To search the table for values in other indexes, you must specify the field name in the table's IndexFieldNames property or the name of the index in the IndexName property.

Specifying the current record after a successful search

By default, the *KeyExclusive* property is *false* and positions the cursor on the first record that matches the search criteria after a successful search. If you prefer, you can set the *KeyExclusive* property for a table component to *true* to position the cursor on the next record after the first matching record.

Searching on an index with more than one key column

If a table has more than one key column, and you want to search for values in a sub-set of that key, set KeyFieldCount to the number of columns on which you are searching. For example, if a table has a three-column primary key, and you want to search only the first column, set KeyFieldCount to 1.

For tables with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Searching on secondary, or alternate, indexes

If you want to search on an index other than the primary index for a table, you must specify the name of the index to use in the *IndexName* property for the table. A table must be closed when you specify a value for *IndexName*. For example, if the CUSTOMER table had a secondary index named "CityIndex" and you want to search for a value using this index and the *Goto* methods, you need to set the value of the table's IndexName property to "CityIndex" prior to commencing the search. The following code provides an example:

```
Table1->Close();
Table1->IndexName = "CityIndex";
Table1->Open();
Table1->SetKey();
Table1->FieldByName("City")->AsString = Edit1->Text;
Table1->GotoNearest();
```

Instead of specifying an index name, you can list fields to use as a key in the *IndexFieldNames* property.

For Paradox and dBASE tables, the fields you list must be indexed, or an exception is raised when you execute the search. For SQL tables, the fields you list need not be indexed.

Repeating or extending a search

Each time you call SetKey, it clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey*. For example, to extend the above search to find a record with a specified city name in a specified country, use the following code:

```
Table1->EditKey();
Table1->FieldByName("Country")->AsString = Edit2->Text;
Table1->GotoNearest();
```

Working with a subset of data

Production tables can be huge, so applications often need to limit the number of rows with which they work. For table components, there are two ways to limit records retrieved by an application: by using filters and ranges.

Understanding the differences between ranges and filters

Filtering datasets

Setting range values

Using Range values

Understanding the differences between ranges and filters

A range is a dynamic, runtime mask, set by your application, that temporarily restricts visible records in a dataset based on range criteria you can set and change as your application runs. If you cancel a range assignment, your application can immediately access all records in the table underlying the dataset.

A *filter* is an event handler called by your application in response to an *OnFilterRecord* event for each record in the dataset. The filter determines whether to accept a record for application access or filter it out. You can apply filters to a dataset when it is first opened. C++Builder can also create filters on the fly when you use *Locate* and *Lookup* to conduct nonindexed searches. Filters are applied to every record retrieved in a dataset. You can filter a dataset in three ways:

- Setting the Filter property of the dataset.
- Restricting record visibility at the time of record retrieval using an OnFilterRecord event handler.
- Finding a record in a dataset that matches search values using the Locate method for the dataset.

When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

Setting Range values

Using Range values

Filtering datasets

Setting Range values

You can use the range value table component methods to enable an application to work with a subset of the data in a database table by limiting the number of rows, or records, retrieved. With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the IndexFieldNames property.

Setting start-range values

SetRangeStart indicates that subsequent assignments to field values specifies the start of the range of rows to include in the dataset. This enables an application to filter the data that is visible to it. Any column values not specified are not considered. The corresponding method *EditRangeStart* indicates to keep existing range values and update with the succeeding assignments. *EditRangeStart* differs from *SetRangeStart* in that the latter clears all the elements of the search key buffer to the default values (NULL). *EditRangeStart* leaves the elements of the search key buffer with their current values. Call *ApplyRange* to apply the new range and filter the dataset.

Setting end-range values

SetRangeEnd indicates that subsequent assignments to field values will specify the end of the range of rows to include in the dataset. This enables an application to filter the data that is visible to it. Any column values not specified are not considered. The corresponding method *EditRangeEnd* indicates to keep existing range values and update with the succeeding assignments. *SetRangeEnd* differs from *EditRangeEnd* in that it clears all the elements of the range filter to the default values (or NULL). *EditRangeEnd* leaves the elements of the range filter with their current values. Call *ApplyRange* to apply the new range and filter the dataset.

Setting start- and end-range values

SetRange([Start Values], [End Values]) combines the functionality of the *SetRangeStart, SetRangeEnd*, and *ApplyRange* methods. *SetRange* assigns the elements of *StartValues* to the beginning index key, the elements of *EndValues* to the ending index key, and then calls *ApplyRange*. This enables an application to filter the data visible to the dataset. If either *StartValues* or *EndValues* has fewer elements than the number of fields in the current index, the remaining entries are set to null.

Applying a range

The *ApplyRange* method is used to cause a range selection to take effect based on the start and end ranges established with the *SetRangeStart* and *SetRangeEnd* methods or the *EditRangeStart* and *EditRangeEnd* methods. Calling this method will make a subset of records from the database table accessible to the application. If neither *SetRangeStart* nor *SetRangeEnd* is called, the range starts at the beginning of the table.

Note: When comparing fields for range purposes, a NULL field is always the lesser value.

Canceling a range

The CancelRange method removes the range limitations for the table that were previously established by calling the *ApplyRange* or *SetRange* methods. Calling this method will restore display of all rows of data for the dataset.

Using Range values

Using Range values

For example, suppose your application uses a table component named *Customers* linked to the CUSTOMER table, and that you created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named StartVal and EndVal, used to specify starting and ending values for a range. If so, the following code could be used to create and apply a range:

```
Customers->SetRangeStart();
Customers->FieldByName("CustNo")->AsString = StartVal->Text;
Customers->SetRangeEnd();
if (EndVal->Text != "")
 Customers->FieldByName("CustNo")->AsString = EndVal->Text;
Customers->ApplyRange();
```

This code checks that the text entered in EndVal is not null before assigning any values to *Fields*. If the text entered for StartVal is null, then all records from the beginning of the table will be included, since all values are greater than null. However, if the text entered for EndVal is null, no records are included, since none are less than null.

This code could be rewritten using the SetRange function as follows:

```
if (EndVal != "")
 Customers->SetRange(OPENARRAY(TVarRec, (StartVal->Text)), OPENARRAY(TVarRec, (EndVal-
>Text)));
Customers->ApplyRange();
```

Setting a range based on a subset of indexed fields

If an index is composed of one or more string fields, the *SetRange* methods support setting a range based on a subset of indexed columns, known as partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Table1->SetRangeStart();
Table1->FieldByName("LastName")->AsString = "Smith";
Table1->SetRangeEnd();
Table1->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith." The value specification could also be

```
Table1->FieldByName("LastName")->AsString = "Sm";
```

This statement includes records that have *LastName* greater than or equal to Sm. The following statement includes records with a *LastName* greater than or equal to "Smith" and a *FirstName* greater than or equal to "J":

```
Table1->FieldByName("LastName")->AsString = "Smith";
Table1->FieldByName("FirstName")->AsString = "J";
```

Including or excluding records that match range values

By default, *KeyExclusive* is *false* and a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. If you prefer, you can set the *KeyExclusive* property for a table component to *true* to exclude records equal to the specified starting and ending ranges.

Sorting records

An index determines the display order of records in a table. In general, C++Builder displays records in ascending order based on a primary index (for dBASE tables without a primary index sort order is based on physical record order). This default behavior does not require application intervention. If you want a different sort order, however, you must specify:

- A secondary, or alternate, index.
- A list of columns on which to sort (SQL only).

Using secondary, or alternate, indexes

To sort on an index other than the primary index for a table, you can specify the name of the index to use in the *IndexName* property for a table. To determine which indexes are available for your dataset, call the *GetIndexNames* method.

Note: There are special requirements for setting a secondary index for a dBASE table.

Specifying a different sort order with IndexName

To specify that a table should be sorted using a secondary, or alternate, index, specify the index name in the table component's *IndexName* property. A table must be closed when you specify a value for *IndexName*. At design time you can specify this name in the Object Inspector, and at runtime you can access the property in your code. For example, the following code sets the index for *CustomersTable* to *ByCompany*:

```
CustomersTable->IndexName = "ByCompany";
```

Caution: *IndexFieldNames* and *IndexName* are mutually exclusive. Setting one property will clear values set for the other.

Retrieving a list of available indexes with GetIndexNames

At runtime, your application can call the *GetIndexNames* method to retrieve a list of available indexes for a table. *GetIndexNames* returns a string list containing valid index names. For example, the following code determines the list of indexes available for the *CustomersTable* dataset:

```
TList *IndexList = new TList();
...
CustomersTable->GetIndexNames(IndexList);
```

Note: For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. If you need to return to using a primary index on a Paradox table after using a secondary index, set the table's *IndexName* property to a null string, as follows:

```
IndexName "";
```

Specifying a dBASE index file

For dBASE tables that use non-production indexes, you must set the *IndexFiles* property to the name of the index file(s) to use before you set *IndexName*. At design time, you can click the ellipsis button in the *IndexFiles* property value in the Object Inspector to invoke the Index Files editor.

To see a list of available index files, choose Add and select one or more index files from the list. A dBASE index file can contain multiple indexes. To select an index from the index file, select the index name from the *IndexName* drop-down list in the Object Inspector. You can also specify multiple indexes in the file by entering desired index names, separated by semicolons.

You can also set *IndexFiles* and *IndexName* at runtime. For example, the following code sets the *IndexFiles* for the *AnimalsTable* table component to ANIMALS.MDX, and then sets *IndexName* to NAME:

```
AnimalsTable->IndexFiles->Strings[0] = "ANIMALS.MDX";
AnimalsTable->IndexName = "NAME";
```

Specifying sort order for SQL tables

In SQL, sort order of rows is determined by the ORDER BY clause. You can specify the index used by this clause either with the

- *IndexName* property, to specify an existing index, or
- *IndexFieldNames* property, to create a pseudo-index based on a subset of columns in the table.

Caution: *IndexName* and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other.

Specifying fields to be used as an index

IndexFieldNames is a string list property used with an SQL server to identify fields to be used an index for the table. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. If you have too many column names or the names are too long to fit within the 255-character limit, use column numbers instead of names. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
PhoneTable->IndexFieldNames = "LastName;FirstName";
```

Note: Use the *IndexName* property for dBASE tables. If you use *IndexFieldNames* on dBASE tables, C++Builder attempts to find an index that uses the columns you specify. If it cannot find such an index, it may raise an exception.

Examining the field list for an index

When your application uses an index at runtime, it can examine the

- *IndexFieldCount* property, to determine the number of columns, or fields, in the current index. If you are using the primary index, this value will be one.
- *IndexFields* property, to give you access to information about each column, or field, of the current index.

IndexFields is string list containing the column names for the index. The following code fragment illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
{
 String ListofIndexFields[20];
 for (i = 0; CustomersTable->IndexFieldCount-1; i++)
    ListOfIndexFields[i] = CustomersTable->IndexFields[i]->AsString;
}
```

Note: *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

Caution: If the component is not active, the value of *IndexFieldCount* will be zero and the information in *IndexFields* will not be valid.

Copying a table and its data

You can duplicate a table's structure and data with a table component's *BatchMove* method. For example, the following statement makes a copy of the database table and data underlying a table component:

```
CustomersTable->BatchMove(CustomerTable,batCopy);
```

Note: A table's *BatchMove* method encapsulates a part of the functionality of batch move components. Batch move components can also append, delete, and update records, and are useful for moving data from one type of table to another.

Modifying data in a table

Modifying data

Deleting records and tables

You can delete all rows (records) of data from the database table specified by the *TableName* property by calling a table component's *EmptyTable* method at runtime. Before calling the *EmptyTable* method, the *DatabaseName*, *TableName*, and *TableType* properties must be assigned values. If the table is open at the time the *EmptyTable* method is called, it must have been opened with the *Exclusive* property set to *true*. For SQL tables, this method only succeeds if you have DELETE privileges for the table. For example, the following statement deletes all records in a dataset:

```
PhoneTable->EmptyTable();
```

Caution: Data deleted with *EmptyTable* is gone forever.

You can delete an existing database table by calling a table component's *DeleteTable* method. Before calling the *DeleteTable* method, the *DatabaseName*, *TableName*, and *TableType* properties must be assigned values and the table must be closed. For example, the following statement removes the table underlying a dataset:

```
CustomersTable->Close();
CustomersTable->DeleteTable();
```

Caution: When you delete a table with *DeleteTable*, the table and all its data are gone forever.

Synchronizing tables linked to the same database table

If more than one table component is linked to the same database table through their *DatabaseName* and *TableName* properties and the tables do not share a data source component, then each table has its own view of the data and its own current record. As users access records through each table component, the components' current records will differ.

You can force the current record for each of these table components to be the same with the *GotoCurrent* method. *GotoCurrent* sets its own table's current record to the current record of another table component. Both tables must have the same *DatabaseName* and *TableName* or a "table mismatch" exception is raised. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne->GotoCurrent(CustomerTableTwo);
```

Tip:

If your application needs to synchronize table components in this manner, put the components in a data module and add a **#include** statement of its header file to each form that accesses the tables.

If you must synchronize table components on separate forms, you must add one form's header file as a **#include** statement in the source unit of the other form, and you must qualify at least one of the table names with its form name. For example

```
CustomersTable->GotoCurrent(Form2->CustomerTableTwo);
```

Creating master-detail forms

A table component's MasterSource and MasterFields properties can be used to establish one-to-many relationships between two tables.

The MasterSource property is used to specify a data source from which the detail table will get information on which records to select based on the current record in the master table. For instance, if you link two tables in a master-detail relationship, the detail table can track the events occurring in the master table through the master table's data source component as specified in the *MasterSource* property.

The *MasterFields* property specifies the column(s) common to both tables used to establish the link. Each time the current record in the master table changes, the new values in those columns are used to select corresponding records from the detail table for display. To link tables based on multiple column names, use a semicolon delimited list. Note that you must index the fields in the detail table first:

```
   Table1->MasterFields = "OrderNo;ItemNo";
```

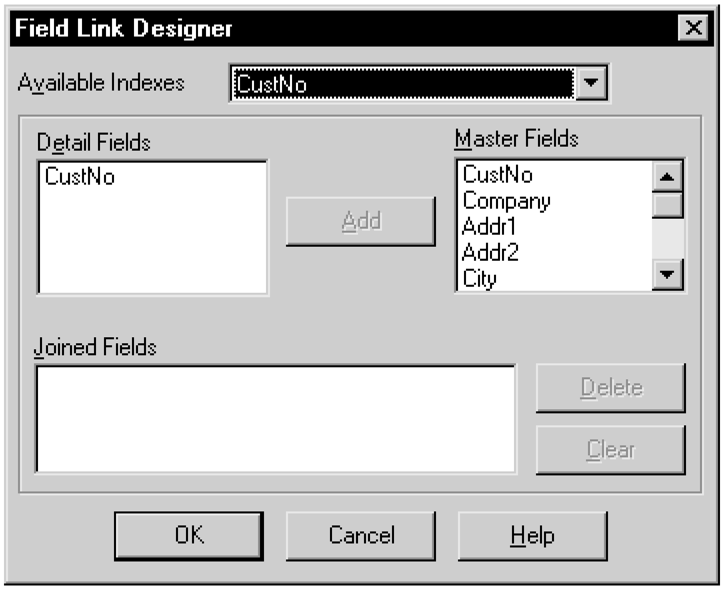
At design time, use the Field Link designer to set the *MasterFields* property.

Example

Linking master and detail tables

Use the Field Link Designer to link master and detail tables. At design time, after specifying the master table in the *MasterSource* property, double-click on the MasterFields property value or click the ellipsis button in the *MasterFields* property value in the Object Inspector to invoke the Field Link designer.

Field Link designer



Select the field to use to link the detail table in the Detail Fields list, and the field to use to link the master table in the Master Fields list. Choose Add. The selected fields are be displayed in the Joined Fields list box. For example,

```
CustNo-> CustNo
```

The detail table displays only indexed fields in the Detail Fields list. The Available Indexes combo box shows the currently selected index used to join the tables. Unless you specify a different index name in the table's *IndexName* property, the default index used for the link is the primary index for the table. If you want to create a link on a field in the detail table that is not in the currently selected index, other available indexes defined on the table can be selected from the drop-down list.

For tables on a database server, the Available Indexes combo box will not appear, and you must manually select the detail and master fields to join in the Detail Fields and Master Fields list boxes.

Example

Building an example master-detail form

The following procedure creates a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable* table, and the detail table is *OrdersTable*.

1. Create a new data module. In the data module, place two TTable and two TDataSource components from the Data Access tab of the Component palette.
2. Set the properties of the *Table1* component as follows:
   - DatabaseName: BCDEMOS
   - TableName: CUSTOMER.DB
   - Name: CustomersTable
3. Set the properties of the *Table2* component as follows:
   - DatabaseName: BCDEMOS
   - TableName: ORDERS.DB
   - Name: OrdersTable
4. Set the properties of the *DataSource1* component as follows:
   - *DataSet*: CustomersTable
   - *Name*: CustomersSource
5. Set the properties of the *DataSource2* component as follows:
   - *DataSet*: OrdersTable
   - *Name*: OrdersSource
6. On a form, place two TDBGrid components from the Data Controls tab of the Component palette. Place focus in the form source unit's editor.
7. Choose File|Include Unit to specify that the form should use the data module.
8. Set the DataSource property of the first grid component to "DataModule2.CustomersSource" and set the *DataSource* property of the second grid to "DataModule2.OrdersSource."
9. Select *OrdersTable* from the data module and set the MasterSource property to "CustomersSource." This step links the CUSTOMER table (the master, or control, table) to the ORDERS table (the detail table).
10. Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link designer to set the following properties:
    - In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
    - Select *CustNo* in both the Detail Fields and Master Fields field lists.
    - Click the Add button to add this join condition. In the Joined Fields list, "CustNo -> CustNo" appears.
    - Choose OK to commit your selections and exit the Field Link Designer.
11. Set the *Active* properties of *CustomersTable* and *OrdersTable* to *true* to display data in the grids on the form.
12. Compile and run the application.

If you run the application now, you will see that the tables are linked together, and when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.