REISDORPH PUBLISHING

Volume 3, Number 8 • August 1999
www.reisdorph.com
$7.00

C++BUILDER

Tips & techniques for Borland C++Builder

Developer's Journal

Hosting forms in a main form

by Kent Reisdorph

Almost every serious C++Builder application ha secondary forms in addition to the main form. Sometimes these forms are presented to the user as dialogs and other times as modeless windows. The VCL model makes creating and displaying secondary forms easy.

The problem with this ease of use is that sometimes C++Builder programmers have trouble thinking outside the box. Not all applications can benefit from the modeless window model. Some applications need to display various views within the main form. This article explains how to host a secondary form within your main form. The secondary form will appear to be part of the main form and the user won't even know that a second form is being shown. **Figure A** shows a main form with a second form in the client area.

Understanding the parent/child relationship

The basic idea of this type of application is to make all secondary forms children of the main form. This design is common in other frameworks (such as OWL or MFC), but is not something that you see often in a VCL application. The VCL doesn't allow you to simply set a property in order to make one form the child of another. You must do a little work in order to make this happen.

To have one form host another, you must tell Microsoft Windows that the secondary form is a child of the main form. In C++Builder programming we tend to think of forms as windows and components as child objects. The truth is, though, that from Windows' perspective all forms and components are simply windows. You can specify that any window (a form or a component) be the child of another window. You only need to step out of the VCL box for a moment.

A better mousetrap

One of the advantages to hosting child forms within the main form is

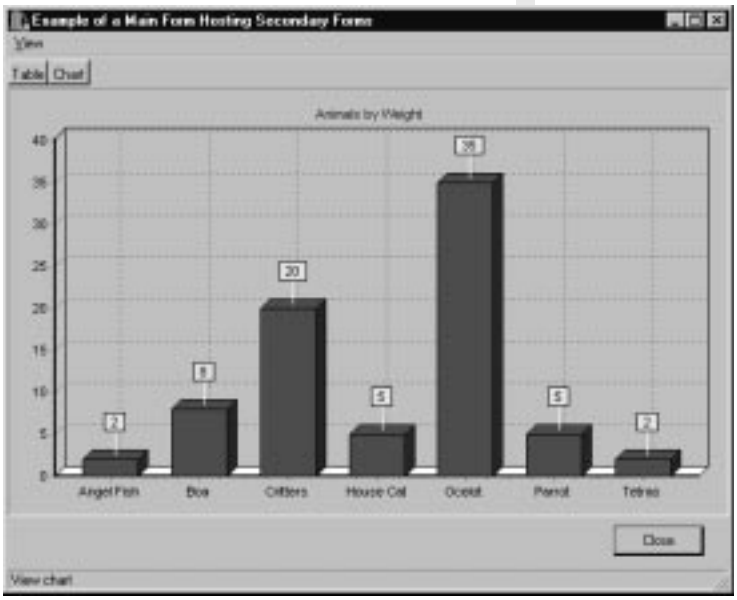


Figure A: *A main form hosting a secondary form in the client area.*

IN THIS ISSUE

7
RAS Update for C++Builder 4

8
Verifying data using CRC

13
Registering AnsiString property editors

16
Determining compiler versions

that you can design your child form just as you would any other secondary form. That is, you create a new form, add components to it, and write the code for the form. This makes it easy to design the child form, and to keep all the code that drives the child form in one central place.

The example program's design

Before I go on, I want to give you some background on the example program for this article. The example, called PARENTING, contains a main form that has a toolbar at the top and a status bar at the bottom. In addition to the main form, the program has two child forms. One child form, called `TTableForm`, displays the ANIMALS.DBF table in a grid. The ANIMALS table is one of the sample database tables that ships with C++Builder. A second child form, `TChartForm` displays the ANIMALS table in a `TChart`. (My apologies to those of you who are using C++Builder 4 Standard as it does not ship with the database components.)

You can choose to view either the table or the chart form by selecting an item from the main menu, or by clicking on one of the toolbar buttons. When you select a form to display, the active form, if any, will be destroyed and the selected form displayed. The child form will be displayed in the client area of the main form below the toolbar and above the status bar. In addition, the child form is resized to always fill the client area of the main form if the main form is resized.

Overriding CreateParams()

As I have said, in order for the main form to host a secondary form you need to set the main form as the secondary form's parent. This is done by overriding the VCL's `CreateParams()` method. `CreateParams()` is called when the VCL creates the underlying window associated with a form. The declaration for `CreateParams()` looks like this:

```
void __fastcall
  CreateParams(TCreateParams& Params);
```

As you can see, `CreateParams()` has a reference to a `TCreateParams` structure as its only parameter. `TCreateParams` is defined in the VCL as follows:

```
struct TCreateParams
{
  char *Caption;
  unsigned Style;
  unsigned ExStyle;
  int X;
  int Y;
  int Width;
  int Height;
  HWND WndParent;
  void *Param;
  tagWNDCLASSA WindowClass;
  char WinClassName[64];
};
```

This structure contains all the information that Windows needs to create a window. (If you have done Windows programming using the API, you will recognize that the members of the `TCreateParams` structure map to a Windows `CREATESTRUCT` structure.) When you override `CreateParams()` you first call the base class's `CreateParams()` method. After that, you can modify the individual members of the `TCreateParams` structure. Here's how a basic overridden `CreateParams()` method might look:

```
void __fastcall TChartForm::CreateParams(
  TCreateParams& Params)
{
  TForm::CreateParams(Params);
  Params.Style =
    WS_CHILD | WS_CLIPSIBLINGS;
  Params.WndParent = MainForm->Handle;
  Params.X = 0;
  Params.Y = 0;
  Params.Width =
    MainForm->ClientRect.Right;
  Params.Height =
    MainForm->ClientRect.Bottom;
}
```

The key points in this code are the lines that set the `Style` and `WndParent` members of the `TCreateParams` structure. `Style` is set to a value that includes the `WS_CHILD` and `WS_CLIPSIBLINGS` window styles. `WS_CHILD` specifies that this window is the child of another window. By definition, a child window has no title bar. At design time the child form will have a title bar, but the title bar will be removed when Windows creates the form at runtime. The `WS_CLIPSIBLINGS` style insures that the various child windows on the main form

don't interfere with one another when the form is painted.

Obviously, a child window must have a parent. You specify the parent by assigning the window handle of the parent window to the TCreateParams structure's WndParent member. As you can see from the preceding code, the WndParent member is set to the Handle property of the main form. Assigning the parent is relatively straightforward, so I won't go into further detail on the subject.

Setting the child form's properties

In addition to the code you see in the CreateParams() method, you must also set some of the child form's properties. Most of the form's properties can be left at their default values. You should, however, set the AutoScroll property to false. This assumes, of course, that your form is designed in such a way that scrolling the form will not be necessary. You should also set the Position property to poDefault, since the size and position of the child window will be set in the CreateParams() method. The Caption and BorderIcons properties will be ignored so you shouldn't have to worry about them. Be sure to leave the BorderStyle property set to bsSizeable, and the BorderWidth property set to 0. If you change these properties, the child form won't fit properly on the main form.

Other components on the form

In many cases, your main form will contain components besides the secondary form. For example, your main form may have a toolbar and a status bar. In that case, you need to account for the toolbar and status bar when you set the X, Y, Width, and Height members of the TCreateParams structure. The child form must fit between the toolbar at the top of the form, and the status bar at the bottom of the form.

Given that, the code that sets the various members of the TCreateParams structure might look like this:

```
Params.X = 0;
Params.Y = MainForm->ToolBar->Height + 1;
Params.Width = MainForm->ClientRect.Right;
Params.Height =
  (MainForm->StatusBar->Top-1) - Params.Y;
```

Note that the Y member is set to the bottom of the toolbar, plus one pixel. The width of the child form is set to the width of the main form's client area, and the height of the child form is calculated based on the top of the child window and the top of the status bar. Basically, the height is set to that part of the main form's client area that falls between the bottom of the toolbar and the top of the status bar.

That is all that is required to make a secondary form the child of the main form. There are a few other features you may want to implement in the child form, but I'll save discussion of those features for later.

Setting up the main form

The main form also needs to be set up to handle a secondary form hosted as a child. First, you must remove the child forms from the application's auto-create list. You will be creating the child forms when needed and don't want them auto-created. In fact, if you don't remove the child forms from the auto-create list they will automatically display when the application starts.

You'll need a variable that keeps track of which child form is currently active. I declared the variable in the main form's public section as follows:

```
TForm* ActiveChild;
```

The ActiveChild variable is public because the child forms need access to the variable. I'll show you how this variable is used in just a bit.

Now you can write the code that will display the child form. First, look at the code, and then I'll explain it.

```
void __fastcall
TMainForm::Chart1Click(TObject *Sender)
{
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}
```

This method is the `OnClick` handler for a menu item on the main form. As you might guess, the handler displays the `TChartForm` child. I first check to see if the `ActiveChild` variable is non-zero. `ActiveChild` will be non-zero if a child window is active. If `ActiveChild` is not zero, I delete the pointer associated with the variable to destroy the active child form. If I don't first destroy the active child form, my program will continue to stack child after child on top of one another.

Next, I create an instance of the `TChartForm` class. I then assign the pointer returned from operator `new` to the `ActiveChild` variable. This way, the `ActiveChild` variable always contains a pointer to the current child form. Finally, I call the `Show()` method to display the child form. The last two lines of code insure that the menu displays a check mark next to the menu item representing either the table or chart view.

In order to complete the discussion of the `ActiveChild` variable, I have to take you back to the child form unit for a moment. Each of the child forms contains an event handler for the `OnClose` event that looks like this:

```
void __fastcall TChartForm::FormClose(
  TObject *Sender, TCloseAction &Action)
{
  MainForm->ActiveChild = 0;
  MainForm->Chart1->Checked = false;
  Action = caFree;
}
```

Note that when the form is destroyed, the main form's `ActiveChild` variable is set to 0. I also uncheck the menu item associated with the child form, and set the `Action` parameter to `caFree`. Setting `Action` to `caFree` tells the VCL to free the memory associated with the form.

You might be wondering why the `FormClose` handler contains those last two lines of code. After all, I just showed you code in the main form that performs these same actions. The answer is that each of the child forms contains a button called Close that can, naturally, be used to close the form. If the form is closed using the Close button, then the memory needs to be freed and the menu item unchecked.

A few extra features

There is at least one feature of the example program that I haven't discussed yet. That is, if the child form is larger than the current client area of the main form, the main form is resized to accommodate the child. That code is placed in the child form's `CreateParams()` method. Earlier, I showed you an example of a basic `CreateParams()` method. I left out the code that resizes the main form because I didn't want to introduce more complex code at that time. You can find the completed `CreateParams()` method for the `TChartForm` class in **Listing B**. The method only differs from that shown earlier in that it contains this code:

```
if (Width > MainForm->ClientWidth)
  MainForm->ClientWidth = Width;
if (Height > (MainForm->StatusBar->Top -
    MainForm->ToolBar->Height))
  MainForm->ClientHeight = Height +
    MainForm->ToolBar->Height +
    MainForm->StatusBar->Height;
```

This code checks to see if the child form's width is greater than main form's `ClientWidth` property. If it is, then the main form's `ClientWidth` property is set to the width of the child form. The next few lines, although a bit more complex, do the same thing for the main form's client height.

The result of this code is that the main form will always be resized to accommodate the child form being displayed.

The example program also accounts for the main form being resized. If the main form is resized, the child form must also be resized so that it continues to fill the client area of the main form. The following code shows the `OnResize` event handler for the main form.

```
void __fastcall
TMainForm::FormResize(TObject *Sender)
{
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
      ActiveChild->Top;
  }
}
```

This code is fairly straightforward, so I don't need to go over every line. Note, though, that I first check the value of the `ActiveChild` variable to be sure that it is non-zero (that is, that it points to a child form). Obviously I don't need to do anything in the `OnResize` event handler if no child form is currently active. The remaining code is a variation of the code you saw in the child form's `CreateParams()` method. It simply calculates the new size for the child window and sets the `Width` and `Height` properties accordingly.

Conclusion

Listing A contains the source code for the example program's main form.

Listing B shows the source code for the `TChartForm` unit. I don't show the headers for these units because they don't contain any meaningful code. I also don't show the code for the `TTableForm` unit because it is identical to the code for the `TChartForm` unit. You can download the example program from our Web site at **www.reisdorph.com**.

Hosting child windows within the main form provides a clean alternative to using MDI, and also to an application that would otherwise display data to the user as modeless forms. Using child forms allows you to design your secondary windows using the form designer, and also helps you keep the code that drives the child form in one place. ◆

Listing A: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"
#include "ChartU.h"
#include "TableU.h"

#pragma resource "*.dfm"
TMainForm *MainForm;

__fastcall TMainForm::TMainForm(TComponent* Owner)
  : TForm(Owner)
{
  // Zero out the ActiveChild variable or it
  // will contain random data.
  ActiveChild = 0;
  // Open the main form's Table component.
  Table->Active = true;
}

void __fastcall TMainForm::Table1Click(TObject
   *Sender)
{
  // If this form is already being displayed
  // then return without doing anything.
  if (Table1->Checked)
    return;
  // Delete the active child if it exists.
  if (ActiveChild) {
    delete ActiveChild;
    ActiveChild = 0;
  }
  // Create an instance of TTableForm.
  TTableForm* form = new TTableForm(this);
  // Assign the DBGrid::DataSource property of the
  // TTableForm's DBGrid to the datasource
  // on the main form.
  form->DBGrid->DataSource = DataSource;
  // Keep track of the active child.
  ActiveChild = form;
  // Show the form.
  form->Show();
  // Update the check marks on the View menu.
  Table1->Checked = true;
  Chart1->Checked = false;
}

void __fastcall TMainForm::Chart1Click(TObject
   *Sender)
{
  // Essentially the same code as described
  // for the Table1Click method above.
  if (Chart1->Checked)
    return;
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}

void __fastcall
TMainForm::FormResize(TObject *Sender)
{
  // If the main form is resized, resize the
  // active child to fill the client area of
  // the main form.
```

```
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
      ActiveChild->Top;
  }
}
```

Listing B: *CHARTU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "ChartU.h"
#include "MainU.h"

#pragma resource "*.dfm"

TChartForm *ChartForm;

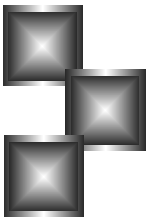
__fastcall
TChartForm::TChartForm(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
TChartForm::CreateParams(TCreateParams& Params)
{
  // Call the base class CreateParams method.
  TForm::CreateParams(Params);
  // Set the style to create a child window.
  Params.Style = WS_CHILD | WS_CLIPSIBLINGS;
  // Set the window's parent as the main form.
  Params.WndParent = MainForm->Handle;
  // The X position of the window is 0.
  Params.X = 0;
  // If the main form is too narrow or too short
  // to accomodate the child form, resize it.
  if (Width > MainForm->ClientWidth)
    MainForm->ClientWidth = Width;
  if (Height > (MainForm->StatusBar->Top -
      MainForm->ToolBar->Height))
    MainForm->ClientHeight = Height +
      MainForm->ToolBar->Height +
      MainForm->StatusBar->Height;
  // The Y position of the child form is just
  // below the main form's toolbar.
  Params.Y = MainForm->ToolBar->Height + 1;
  // The width of the child form is the same as
  // the main form's client width.
  Params.Width = MainForm->ClientRect.Right;
  // Calculate a height based on the bottom of
  // the toolbar, and the top of the status bar.
  Params.Height =
    (MainForm->StatusBar->Top - 1) - Params.Y;
}

void __fastcall TChartForm::FormClose(
  TObject *Sender, TCloseAction &Action)
{
  // Update the main form's ActiveChild property
  // to indicate no child is active.
  MainForm->ActiveChild = 0;
  // The child form might have been closed via
  // the Close button so update the main menu's
  // check marks, and tell the VLC to clean up
  // the memory for this form.
  MainForm->Chart1->Checked = false;
  Action = caFree;
}

void __fastcall
TChartForm::CloseBtnClick(TObject *Sender)
{
  Close();
}
```

RAS update



by Kent Reisdorph

In our March and April issues we presented articles on using Microsoft's Remote Access Services (RAS) to establish dial-up connections. Those articles were written using C++Builder 3. With C++Builder 4, however, I noticed that my RAS applications no longer worked as I expected. One of the symptoms was a runtime error from some of the RAS functions. The return value from those functions was `ERROR_BUFFER_TOO_SMALL`. The textual description of this error is, "The caller's buffer is too small." I spent more than a few hours trying to track down this problem. I started digging around in RAS.H and found some entries like this:

```
RASCONNA
{
  DWORD    dwSize;
  HRASCONN hrasconn;
  CHAR     szEntryName[ RAS_MaxEntryName + 1 ];
#if (WINVER >= 0x400)
  CHAR     szDeviceType[ RAS_MaxDeviceType + 1];
  CHAR     szDeviceName[ RAS_MaxDeviceName + 1];
#endif
#if (WINVER >= 0x401)
  CHAR     szPhonebook [ MAX_PATH ];
  DWORD    dwSubEntry;
#endif
};
```

Hmm… that could certainly lead to a "buffer too small error" if the `WINVER` macro were defined as something other than `0x0400`. Curious, I started hunting in the Windows headers. Here's what I found in WINDEF.H:

```
#ifndef WINVER
#define WINVER 0x0500
#endif /* WINVER */
```

Aha! Presumably Microsoft and Borland were building support for Windows 2000 (which includes many enhancements to RAS) into the Windows headers and those headers shipped with C++Builder 4. In C++Builder 3, `WINVER` was defined as `0x0400` so obviously something changed.

Once I figured out the problem, it did not take long to fix it. If you are building applications that use RAS in C++Builder 4 you must use this construct when including the RAS headers:

```
#pragma warn -dup
#define WINVER 0x400
#include <ras.h>
#include <raserror.h>
#define WINVER 0x500
```

This code temporarily redefines the `WINVER` macro to `0x400` and then sets it back to `0x500` after including the RAS headers. The line that reads, `#pragma warn -dup` suppresses the warning the compiler will issue that says:

```
Redefinition of 'WINVER' is not identical.
```

It's not only RAS.H that suffers from this problem. For a list of affected headers, search the Windows headers for these lines of text:

```
WINVER >= 0x401
WINVER >= 0x0500
WINVER >= 0x500
```

You will no doubt find several places where problems may arise with your existing code.

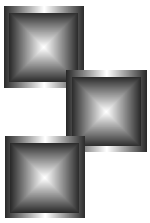
As I have said, this problem caused me a few hours of aggravation before I figured out what was going wrong. I hope that this information will help you avoid the frustration I experienced before I discovered the solution. ◆

Have you moved? Are you going to?

Send any change of address to:

E-mail:
customer_relations@reisdorph.com

Mail:
Reisdorph Publishing
P.O. Box 50213
Colorado Springs, CO 80919



Verifying data using CRC

by John M. Miano

Whenever a data message is transmitted it can be corrupted in ways that cannot be predicted. Even data sitting on a disk can be corrupted by hardware failures, software bugs, or malicious activity.

A key requirement of networking systems and file transfer programs is to ensure that data gets transferred reliably. When a message is received, the amount of data and contents must match what was transmitted exactly. If a message is modified during transmission, the system asks the sender to retransmit the corrupted message.

Note: The code in this article is for use with C++Builder 3. You can download the code for C++Builder 4 from our web site.

Simple Error Detection

The first problem in handling data corruption is how to determine if the data has been modified. An early method used in serial communications was to check the parity of the data bytes. Since ASCII characters only require 7-bits, one bit is left over in each byte and that bit can be used to ensure that each byte has either odd or even parity. In other words, the sender would set or clear the last bit to ensure that each data byte either had either an odd or even number of bits set, depending upon what the receiver was expecting.

The problem with this system is that it only works reliably with errors of one bit. If more than one bit gets corrupted, there is only a 50/50 chance of detecting errors.

Checksum

Another simple method for error detection is to include a checksum value within each message. The sender calculates the checksum for the data before it is transmitted. The receiver also calculates the checksum on the data and, if the checksum values do not match, the receiver knows that the data has been corrupted in some way.

A simple checksum function is to take the sum of each byte in the message. Take a message consisting of the characters "ABCDEF", for example. Before transmitting the message, an application can sum each byte and append the result (495) to the message.

While summing the data bytes allows errors to be detected in some situations, it is possible for many types of errors to slip by. For example, a message consisting of the characters "ACBDEF" differs from the previous string by only two bits, but the sum of the data bytes is the same. The data is obviously incorrect but a checksum won't catch the error.

A Better Check Value

What is really needed for reliable error detection is a function that causes small changes in the input to produce large changes in the check value. In other words, we need a function that essentially produces a random value from a data block.

The most common method for generating a check value is known as the Cyclic Redundancy Check, or CRC. A CRC function can be used to calculate the CRC on a block of data. The CRC function works by treating a data block as a large binary number and dividing by a constant value. The CRC result is the division remainder.

In reality, the process of generating a CRC is not true division, but rather something that looks a lot like division. CRC calculations use binary arithmetic with no carries or borrows. Bit positions are `XOR`ed rather than subtracted.

Figure A show the process for generating the CRC value for the message "AB" (`0100 0001  0100 0010` in binary) using the divisor `1 10001`. Notice in **Figure A** how `XOR` operations are used in place of subtraction as in real division. Also notice that the number of bits in the CRC value is one less than the number of bits in the divisor.

Implementing the CRC

CRC is simple to implement in hardware but if the CRC were implemented in software by performing bit-by-bit pseudo-long division, it would be relatively computationally intensive. Fortunately

long division is not required in a software implementation.

If you look at the individual XOR operations in **Figure A** you can see that at each step the one data bit is shifted into the remainder and XORed. The individual steps are trivial.

Software CRC implementations are invariably implemented with a lookup table, allowing byte-by-byte processing rather than by bit. The application maintains a register containing the CRC value and updates it as each byte is processed.

The following code fragment shows how the CRC is calculated for a buffer from a lookup table. BITCOUNT is the size of the CRC value in bits. I will explain how the lookup table is generated later.

```
for (unsigned int ii=0;ii<length;++ ii)
{
  int index = ((crc_register
    >> (BITCOUNT - 8)) ^ buffer[ii]);
  crc_register = crc_table
    [index & 0xFF] ^ (crc_register << 8);
}
```

The CRC Polynomial

In **Figure A** I made an arbitrary choice for the divisor in the CRC calculation. There are a number of CRC variants in use and one of the ways in which they vary is in the choice of the divisor. The choice of non-zero bits in the divisor determines the types of errors the CRC variant can detect.

Descriptions of the CRC frequently describe it in terms of polynomial division. Instead of representing the CRC divisor as 1 1000 0000 0000 0101 it is represented as x16 + x15 + x2 + 1. This is the reason you will usually see the divisor referred to as the CRC polynomial.

Since the highest order bit is always 1, when the polynomial is shown in hexadecimal the highest order bit is invariably left out. For example, the CRC polynomial shown above would be listed as the 2-byte value 0x8005 rather than 0x18005.

Most CRC functions produce either 16 or 32-bit values (17 or 33-bit polynomials). There are some 12-bit CRC functions, but these are not very common. The number of bits in the divisor limits the size of the block that can reliably detect errors. Divisors with more bits can reliably detect errors in larger data blocks.

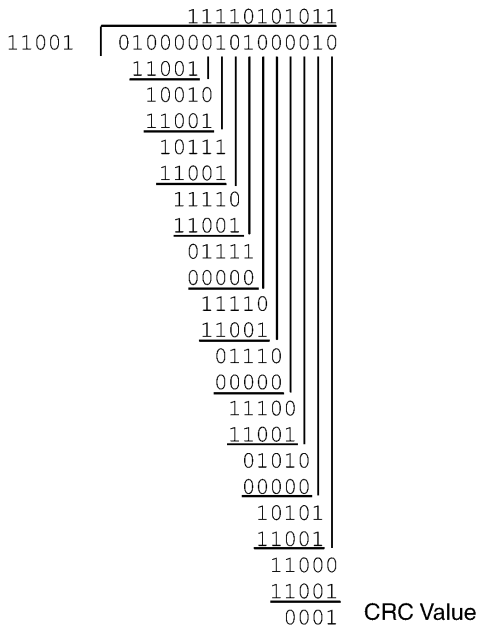


Figure A: *The CRC value for the string "AB" using the divisor 1 10001.*

Other CRC Variations

Besides the polynomial length and polynomial value, there are three other common variations among CRC functions.

Reversed CRC

From an implementation point of view, the most significant difference among CRC variants is the ordering of the bits in the data block. When performing division, start with the most significant bit and work to the least significant.

However, when data is transmitted, such as over a serial line, the least significant bit within a byte often comes first in the data stream. In a hardware implementation, it is easier to process the bits as they arrive.

As a result, there are CRC variants that process data bits both in the natural order and reversed. Since the CRC uses something that looks like division, but is not actually division, the order in which the bits are processed is irrelevant (as long as they are processed consistently in the same order).

This example shows the process for using a table lookup to calculate a reversed CRC. The two versions are almost identical except for way the CRC register is shifted.

```
for (unsigned int ii=0;ii < length; ++ii)
{
  crc_register = crc_table.values[
       (crc_register ^ buffer [ii]) & 0xFF]
    ^ (crc_register >> 8);
}
```

Initialization

Refer back to the CRC process shown in **Figure A**. Suppose the message were corrupted with zero bits inserted at the start of the message. Since zero divided by the polynomial is zero, these extra bits do not change the CRC value. In order to detect this type of error, some CRC variants initialize the CRC register to a value other than zero.

Final Value

Some CRC variants modify the remainder value as well by taking the complement of the value (XOR with –1). The CRC variants that modify the remainder invariably initialize the CRC to a value other than 0. The converse is not true, however.

Classifying CRC Variants

In the previous sections I have identified the five ways common CRC variants differ: polynomial length, polynomial value, reversed or forward, initialization, and finalization. **Table A** shows how the major CRC variants fit into these classifications.

Creating a generic CRC implementation

C++ templates provide a good mechanism for dealing with parameterized variants, such as the ones shown in **Table A**. Here we have a template class whose parameters are modeled on **Table A**.

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
class Crc {
public:
  Crc();
  Crc(const Crc &);
  ~Crc() {}
  Crc &operator=(const Crc &);

  unsigned long value() const;
  void reset();
  void update(const char *buffer,
    unsigned int length);
private:
  struct CrcTable {
    enum { MAXBYTEVALUES = 256 };
    CrcTable();
    unsigned long values [MAXBYTEVALUES];
  };
  static const CrcTable crc_table;
  unsigned long crc_register;
};
```

Creating the Lookup Table

The following function creates the value for an entry in a lookup table for the forward CRC process. You can see that this function divides the input value using the process shown in **Figure A**. Here is the function:

```
unsigned int ForwardTableEntry(
  unsigned long polynomial,
  unsigned int entryindex,
  unsigned int bitcount)
{
  unsigned long result =
    entryindex << (bitcount - 8);
  for (unsigned int ii=0;
       ii<BITSPERBYTE; ++ii) {
    if ((result & bits [bitcount-1]) == 0)
```

Table A: *CRC Variants*

Name	Polynomial Length	Polynomial 1	Reserved	Initialization	Final XOR
CRC-16	16	0x8005	Yes	0	0
X.25	16	0x1021	Yes	-1	-1
CCITT	16	0x1021	No	-1	0
XMODEM	16	0x1021	No	0	0
CRC-32	32	0x04D11DB7	Yes	-1	-1

```
      result <<= 1;
    else
      result = (result << 1) ^ polynomial;
  }
  unsigned long mask =
      ((1UL << (bitcount - 1)) - 1UL) |
      (1UL << (bitcount - 1));
  result &= mask;
  return result;
}
```

Entries for reversed CRC lookup tables are created in a similar manner, except with bit reversals.

```
const unsigned long bits [32] = {
 0x00000001UL, 0x00000002UL, 0x00000004UL,
 0x00000008UL, 0x00000010UL, 0x00000020UL,
 0x00000040UL, 0x00000080UL, 0x00000100UL,
 0x00000200UL, 0x00000400UL, 0x00000800UL,
 0x00001000UL, 0x00002000UL, 0x00004000UL,
 0x00008000UL, 0x00010000UL, 0x00020000UL,
 0x00040000UL, 0x00080000UL, 0x00100000UL,
 0x00200000UL, 0x00400000UL, 0x00800000UL,
 0x01000000UL, 0x02000000UL, 0x04000000UL,
 0x08000000UL, 0x10000000UL, 0x20000000UL,
 0x40000000UL, 0x80000000UL};

unsigned long Reverse(
  unsigned long value,
  unsigned int bitcount)
{
  unsigned long result = 0;
  for (unsigned int jj=0;jj<bitcount;++jj)
  {
    if ((value & bits [jj]) != 0)
      result |= bits [bitcount - jj - 1];
  }
  return result;
}

unsigned long ReverseTableEntry(
  unsigned int polynomial,
  unsigned int entryindex,
  unsigned int bitcount)
{
  unsigned long result = entryindex;
  for (unsigned int ii=0;ii < 8; ++ii)
  {
    if ((result & 1) == 0)
      result >>= 1;
    else
      result = (result >> 1) ^
        Reverse(polynomial, bitcount);
  }
  unsigned long mask =
    ((1UL << (bitcount - 1)) - 1) |
      (1UL << (bitcount - 1));
  result &= mask;
  return result;
}
```

Here is the class constructor and definition for the lookup table:

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
const CrcTable Crc<
  BITCOUNT,
  POLYNOMIAL,REVERSE,INITIAL,
  FINALMASK>::crc_table;

template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
Crc<BITCOUNT,POLYNOMIAL, REVERSE,
  INITIAL,FINALMASK>::CrcTable::CrcTable()
{
  if (REVERSE) {
    for(unsigned int ii=0;
      ii<MAXBYTEVALUES; ++ii) {
      values [ii] =
      ReverseTableEntry(
        POLYNOMIAL, ii, BITCOUNT);
    }
  } else {
    for(unsigned int ii=0;
      ii<MAXBYTEVALUES; ++ii) {
      values [ii] = ForwardTableEntry(
        POLYNOMIAL, ii, BITCOUNT);
    }
  }
}
```

The lookup table is a static member object so that all equivalent instantiations share the same table. The table defines a constructor that initializes it at startup.

The `reset()` function on the following page initializes the CRC using the value specified in the template parameter. Use this function when you want to use a CRC object to calculate the CRC value for more than one block of data.

```
template <
  unsigned int BITCOUNT,
```

```
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
void Crc<BITCOUNT,POLYNOMIAL,
    REVERSE,INITIAL,FINALMASK>::reset()
{
  crc_register = INITIAL;
  return;
}
```

The `update()` function updates the CRC register using a data block:

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
void Crc<BITCOUNT,POLYNOMIAL,
  REVERSE,INITIAL,FINALMASK>
  ::update(const char *buffer,
    unsigned int length)
{
  // The process for updating depends upon
  // whether or not we are using the
   reversed
  // CRC form.
  if (REVERSE)
  {
    for (unsigned int ii=0;ii<length;++ii)
    {
      crc_register = crc_table.values
      [(crc_register ^ buffer[ii]) & 0xFF]
        ^ (crc_register >> 8);
    }
  } else {
    for (unsigned int ii=0;ii<length;++ii)
    {
      unsigned long index = (
       (crc_register >> (BITCOUNT -
        BITSPERBYTE)) ^ buffer[ii]);
      crc_register =
        crc_table.values[index & 0xFF] ^
          (crc_register << BITSPERBYTE);
    }
  }
  return;
}
```

The `value()` function returns the contents of the CRC register XORed with the `FINALMASK` template parameter:

```
template <unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>
unsigned long Crc<BITCOUNT,POLYNOMIAL,
  REVERSE,INITIAL,FINALMASK>::value()
   const
{
  unsigned long result =
    crc_register ^ FINALMASK;
  // The initial value is
  // 1 << BITCOUNT - 1. The convolutions
  // prevent an integer overflow.
  static const unsigned long mask =
    ((1UL << (BITCOUNT - 1)) - 1UL) |
    (1UL << (BITCOUNT - 1));
  result &= mask;
  return result;
}
```

The example program for this article includes the copy constructor, default constructor, and assignment operator. You can download the example from our Web site at **www.reisdorph.com**.

Using the CRC Template Class

You need to create a header file and compilation for each CRC class you want to create from the template. To create a CRC-32 class you would create the header file CRC32.H that defines a type that uses the values in **Table A** as template parameters:

```
#include "crc.h"
typedef Crc <32, 0x04C11DB7,
  true, 0xFFFFFFFF, 0xFFFFFFFF> Crc32;
```

To create the compilation unit CRC32.CPP, all you need are include directives.

```
#include "crc32.h"
#include "crc.cpp"
```

When you compile the CRC32.CPP file you must use the `-Ja` switch either at the command line or using

```
#pragma option -Ja
```

This ensures that the template members get created properly.

The following console application example shows how to use the `Crc32` class to calculate the CRC-32 value for the string "abcdefgh12345678."

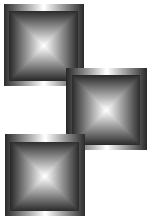
```
#include <iostream>
#include <cstring>
using namespace std;
#include "crc32.h"
const char *msg1 = "abcdefgh";
const char *msg2 = "12345678";
main()
{
  Crc32 crc;
  crc.update(msg1, strlen(msg1));
  crc.update(msg2, strlen(msg2));
  cout << hex << crc.value() << endl;
  return 0;
}
```

Don't be concerned over efficiency because of the tests for reversed CRC in the update function. Since the value in the test is a constant, you can be assured that any compiler will optimize these tests out.

Conclusion

The CRC is an effective and efficient method for detecting corruption in data. A template implementation allows many CRC variants to be implemented with one set of code. The template CRC class shown here allows you easily incorporate data verification in your applications. ◆

Registering AnsiString property editors



by Kent Reisdorph

One of the most powerful features of C++Builder is the ability to create your own components. Certainly not all of you are writing your own components. Even fewer of you have written property editors for your components. On the other hand, if you have written a property editor for a property based on `AnsiString`, you almost certainly ran into serious problems trying to register the property editor.

Simply put, the VCL will not let you register a property editor for a property based on `AnsiString`. This article will explain how to register a property editor for an `AnsiString` property.

As part of this article I'll also show you how to register property editors for the C++ integral data types such as `int`, `char`, `long`, and so on.

Identifying the problem

You've just written a property editor and, of course, you're proud of your creation. All you have to do is register the property editor and try it out for the first time. Naturally, you try to register the property editor with the following code.

```
RegisterPropertyEditor(__typeinfo(String),
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

You compile the package and are immediately greeted with an error from the compiler:

```
__classid requires VCL style class type.
```

At this point most of you would try to work out the problem and, no doubt, throw your hands up in frustration a few miserable hours later. This is the kind of error that is not easily understood, nor remedied.

The problem is that `__typinfo` used in `RegisterPropertyEditor()` requires a VCL-derived class or a Pascal data type. `AnsiString`, being a pure C++ helper class, is not derived from any VCL class and, as such, does not qualify for use with `__typeinfo`. This problem not only affects `AnsiString` property editors, but also property editors for properties based on C++ types such as `int` or `char`.

Now that you see what the problem is, I'll explain the solution.

The solution(s)

There are at least three solutions for the `AnsiString` property editor dilemma. One solution to the `AnsiString` property editor

problem involved deriving a class from TPersistent and declaring an AnsiString as a class member. The class itself was registered as the type for the property and the AnsiString data member held the string data. This solution, while the only one available for a long time, is messy. I won't go into the details, but you can take my word for it.

The remaining two solutions are easier as you'll see in the following sections.

The Van Ditta solution

Several months ago I read a post from Mark Van Ditta in the Borland newsgroups. Mark described how he had solved the AnsiString property editor problem. The solution was both simple and ingenious. Mark was determined to solve the AnsiString property editor problem. He discovered that the __typeinfo keyword was emulating what the Object Pascal TypeInfo() function does—return a pointer to a TTypeInfo structure. (Later I realized that the VCL help for RegisterPropertyEditor() has a hint that pointed in this direction.) With that information Mark created a function that can be used with RegisterPropertyEditor(). The function looked something like this:

```
TTypeInfo* AnsiStringTypeInfo()
{
  TTypeInfo* typeInfo = new TTypeInfo;
  typeInfo->Name = "AnsiString";
  typeInfo->Kind =  tkLString;
  return typeInfo;
}
```

Like I said, simple yet ingenious. This function simply creates a dynamic instance of the TTypeInfo structure, sets the Name and Kind members of the structure, and then returns a pointer to the structure. (You don't need to delete the memory associated with the structure because the VCL will take care of it for you.) The correct call to RegisterPropertyEditor() is shown below.

```
RegisterPropertyEditor(
  AnsiStringTypeInfo(),
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

So long as RegisterPropertyEditor() gets a pointer to a TTypeInfo structure, and as long as the Name and Kind members of the structure contain information that the VCL understands, everything works.

Once you understand the basic idea of the AnsiStringTypeInfo() function, it's easy to create additional functions to register property editors of the integral data types, such as int. The key is in understanding TTypeInfo::Kind. The Kind member of TTypeInfo is an enumeration of type TTypeKind. TTypeKind is defined as follows:

```
enum TTypeKind {
  tkUnknown, tkInteger, tkChar,
  tkEnumeration, tkFloat, tkString, tkSet,
  tkClass, tkMethod, tkWChar, tkLString,
  tkWString, tkVariant, tkArray, tkRecord,
  tkInterface, tkInt64, tkDynArray };
```

Obviously, you can create functions similar to AnsiStringTypeInfo() by using one of the values of TTypeKind along with a corresponding value for the Name member of TTypeInfo. Here's a type info function for a property of type int:

```
TTypeInfo* IntTypeInfo(void)
{
  TTypeInfo* typeInfo = new TTypeInfo;
  typeInfo->Name = "int";
  typeInfo->Kind =  tkInteger;
  return typeInfo;
}
```

Note that I set the Name member to "int" and the Kind member to tkInteger. Some trial and error is required in order to get the Name and Kind members synchronized, but for the most part it's pretty obvious what to pass for the Name member, given a known type. If your property editors don't work then you may have to try different values for the Name member until the property editor works.

The Mitov solution

Another solution was recently presented to me by Boian Mitov. Boian's solution looks like this:

```
TPropInfo* PropInfo = ::GetPropInfo
  (__typeinfo(TMyComponent), "FileName" );

RegisterPropertyEditor(*PropInfo-
   >PropType,
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

This solution is actually a bit cleaner than the first solution presented. In this case a pointer to a `TPropInfo` structure is used. The property information for a particular property is first obtained by a call to `GetPropInfo()`. The result is a pointer to a `TPropInfo` structure. `TPropInfo` is defined in TYPEINFO.HPP:

```
struct TPropInfo
{
  PTypeInfo *PropType;
  void *GetProc;
  void *SetProc;
  void *StoredProc;
  int Index;
  int Default;
  short NameIndex;
  System::ShortString Name;
};
```

Notice that the `PropType` parameter is a pointer to a `PTypeInfo` type. Since `PTypeInfo` is itself a pointer, the `PropType` member is a pointer to a pointer. Since `PropType` is a pointer to a pointer it must be dereferenced before passing it to the `RegisterPropertyEditor()` function.

Here's how the call to `RegisterPropertyEditor()` looks:

```
RegisterPropertyEditor(*PropInfo->PropType,
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

As I have said, this solution is a bit cleaner than the first solution presented, although each works equally well. In the end, choose the method that makes the most sense to you.

Conclusion

Registering `AnsiString` property editors is as vexing as any aspect of writing components in C++Builder. Since the first version of the compiler, component developers have looked to Borland to provide a reasonable solution to this problem. Unfortunately, each new version of C++Builder has failed to address the issue.

Thanks to the efforts of Mark Van Ditta and Boian Mitov, C++Builder programmers have not one, but two solutions to the problem. ◆

About our contributors

John M. Miano is the chief engineer with Colosseum Builders, Inc. in Summit, New Jersey where he works on entertainment and broadcasting systems. He's the author of *Borland C++ Builder How-To.* You can contact John at **Miano@colosseumbuilders.com**.

Kent Reisdorph is a senior software engineer at TurboPower Software and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++ Builder in 21 Days* (ISBN 0-672-31020-1) and *Teach Yourself C++Builder in 14 Days* (ISBN 0-672-31051-1). You can contact Kent at **kentr@turbopower.com**.

C++BUILDER
Tips & techniques for Borland C++Builder
Developer's Journal

C++Builder Developer's Journal is published monthly by Reisdorph Publishing, P.O. Box 50213, Colorado Springs, CO 80919.

Customer Relations

Customer Relations voice (719) 266-0736
Customer Relations fax (719) 260-7151

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to

Reisdorph Publishing
Attn: Customer Relations
P.O. Box 50213
Colorado Springs, CO 80919

Or contact our Customer Relations department via Internet e-mail at customer_relations@reisdorph.com.

Editorial

Editor-in-Chief Kent Reisdorph
Copy Editor Jennifer Franzen
Managing Editor Robert A. DelRossi

You may address tips, special requests, and other correspondence to

Reisdorph Publishing
Attn: Editorial
P.O. Box 50213
Colorado Springs, CO 80919

Or contact us via Internet e-mail at editor@reisdorph.com.

Copyright

Copyright © 1999, Reisdorph Publishing. *C++Builder Developer's Journal* is an independently produced publication of Reisdorph Publishing. All rights reserved. Reproduction in whole or in part in any form or medium without express written permission of Reisdorph Publishing is prohibited. Reisdorph Publishing reserves the right, with respect to submissions, to revise, republish, and authorize its readers to use the tips submitted for personal and commercial use.

Microsoft Windows is a registered trademark of Microsoft Corporation. C++Builder and Borland are registered trademarks of Inprise Corporation. All other product names or services identified throughout this journal are trademarks or registered trademarks of their respective companies.

Price

Domestic $79/yr ($7.00 each)
Outside US $99/yr ($9.00 each)

Back Issues

To order back issues, call Customer Relations at (719) 266-0736. Back issues cost $7.00 each, $9.00 outside the US. You can pay with MasterCard, VISA, Discover, or American Express.

For more information on this journal please visit our Web site at www.reisdorph.com.

Postmaster

Periodicals postage paid in Colorado Springs, CO.
Postmaster: Send address changes to:

Reisdorph Publishing
P.O. Box 50213
Colorado Springs, CO 80919

PERIODICALS MAIL

C++Builder
Technical Support
(888) 683-2378
www.inprise.com

Please include account number from label with any correspondence.

Determining compiler version

by Kent Reisdorph

If you are writing applications or components that must compile under different versions of C++Builder then you may need to determine the version of the compiler in order to conditionally compile sections of code. Each version of the compiler assigns a value to a macro called `__BORLANDC__`. You can test the version of the compiler by checking the value of this macro in your code. For example:

```
#if (__BORLANDC__ >= 0x530)
#pragma package(smart_init)
#endif
```

In this case, the `#pragma package` line is only compiled if the code is being compiled on C++Builder 3 or greater (C++Builder 1 does not use packages, of course). We use code like this to insure that the examples for the journal will compile under any version of C++Builder.

You can easily find out the value of the `__BORLANDC__` macro by running this code under various versions of C++Builder:

```
char buff[10];
sprintf(buff, "%x", __BORLANDC__);
Label1->Caption = buff;
```

Table A shows the values of the `__BORLANDC__` macro for the current versions of C++Builder.

Armed with this information you can easily write both applications and components that work on all current versions of C++Builder. ◆

Table A: *Values of __BORLANDC__ by compiler*

Value	Compiler version
0x520	C++Builder 1
0x530	C++Builder 3
0x540	C++Builder 4

For this month's code be sure to visit:

www.reisdorph.com