Extending Web Solution Builder

By Chad Z Hower

Web Solution Builder (previously codenamed Portcullis/IAG) is a RAD web application development tool for Delphi and C++Builder. It allows the developer to work in a manner consistent with standard Delphi programs. (Web Solution Builder is available for Delphi 2/3 and C++ Builder, from here I'll simply refer to them collectively as "Delphi").

While Web Solution Builder can generate and use Java, ActiveX and JavaScript, it does not require any of these and will not use any of them in the default configuration. It converts all Delphi forms to HTML and transparently interprets the CGI, ISAPI, NSAPI or WinCGI calls for the programmer. The programmer communicates with a Delphi form as s/he would normally and only has to program and debug Delphi: no coding of CGI, ISAPI, NSAPI, WinCGI or HTML is required.

There are two editions of Web Solution Builder: Developer and Enterprise. The main differences between them are scalability and security. Enterprise can be scaled to multiple machines with true dynamic load balancing, and can be completely deployed behind a firewall (while still accessible from the outside), with the web server in front of the firewall, thus securing the application. With Developer, the application must reside on the same machine as the web server.

This article will not attempt to cover Web Solution Builder itself. Web Solution Builder has been designed to be completely extensible so that third party vendors and individual users may create their own components and extensions, or add support for their existing Delphi components. This article will cover techniques to do this. Web Solution Builder contains many APIs, however the one that we will be discussing is called the *Drawbridge* API.

I will show that Web Solution Builder allows you to do such tasks as take a Delphi form (Figure 1) and turn it into a Web application merely by running it (Figure 2) as well as show you how to extend this amazing tool and even write your own new components.

At Work On The Web

When a Web Solution Builder application is developed, each form contains one or more `TIAG_Regions`. A region corresponds to an HTML page. Normally only one per form is used, but with frames and tabs, multiple forms are necessary. Every control that is to be displayed in the browser must be put onto the region. Each time a specific region is requested, the positions of each control, and their visible property will determine how they are rendered into HTML.

Drawbridge Extension Types

There are three ways to create your own components for use with Web Solution Builder. Firstly, you can register an HTML Writer. This is similar to other product offerings: you directly generate all the HTML.

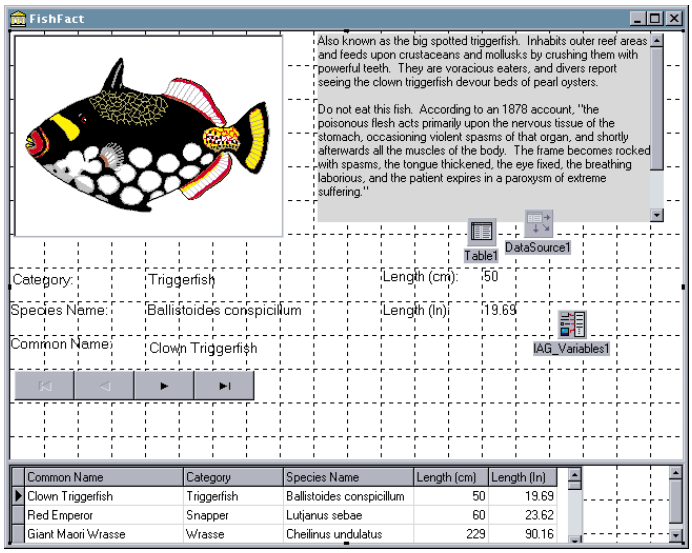
Secondly, you can inherit from `TIAG_Base` "Placeholders". This is a method which should be used when you want to provide HTML functionality in the RAD environment, but not based on an existing Delphi control.

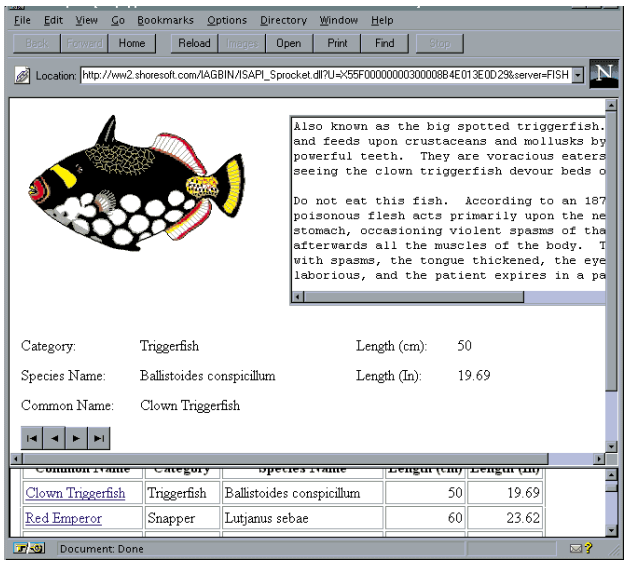
Lastly, you can use the `Extension` API to extend an existing Delphi Control. This method should be used when you wish to add support for a Delphi control which already exists. Collectively these methods are referred to as the `Drawbridge` API.

Registering An HTML Writer

This is useful for small functions, porting code from other products, or porting existing ISAPI, NSAPI or CGI code. This functionality was finalized in version 1.008. HTML.

➤ *Figure 1 (left) Figure 2 (right)*





To register an HTML writer you need to call the function RegisterIAGHTMLHook. Its declaration is shown in Figure 3.

Each time the object specified in psForm is requested, the procedure specified in proc will be called. The parameters that will be passed are the objc item and the form name. This allows for multiple writers to share the same writer procedure.

Once the procedure has been called, you can use Web Solution Builder routines such as WriteHTML, ServeData or DirectTo, or even write directly to a stream. These methods are covered in the Web Solution Builder documentation, so we will not cover them in detail but here is a quick summary. WriteHTML accepts a string, and allows you to incrementally output HTML. ServeData allows you to progmatically serve up Web Solution Builder forms or other registered writers. DirectTo allows you to serve up static HTML contained in a file and Stream allows you to write HTML directly to a stream.

Now we will construct a simple example. Let's register a writer which simply outputs the time and date. First, let's create the writer procedure, see Listing 1.

WriteHTML is a method of iagin, which is a global information object in Web Solution Builder. We use this routine to write our HTML. In this example, it is very straight forward. We write the minimum header, followed by the actual text we want to display. Now we need to register the writer:

```
RegisterIAGHTMLHook('Sample',
  WriterSample, nil);
```

This call should be made in the DPR file, after the Portcullis_Init. The complete source is shown in Listing 2. Figure 4 shows what our final output looks like in the browser.

```
procedure RegisterIAGHTMLHook(const psName: String;
  proc: TProcHTMLHook; objc: TObject);
```

psName: The form name that you wish assign to this hook. This name is used in the form parameter of the URL to reference this routine.

proc: The procedure that will be called when the writer is requested and is of type TProcHTMLHook.

```
ProcHTMLHook = procedure(objc: TObject; const psFormName:
  String);
```

objc: User definable and not used by Web Solution Builder. This has similar functionality to the Tag property of Delphi components. If you do not wish to take advantage of this parameter, pass nil.

➤ *Figure 3*

```
Procedure WriterSample(objc: TObject; const psFormName: String);
Begin
  with iagin do begin
    WriteHTML('<HTML><BODY>');
    WriteHTML('The time is: ' + FormatDateTime('', now));
  end;
end;
```

➤ *Listing 1*

```
writer_Sample.pas
unit writer_sample;
interface
Procedure WriterSample(objc: TObject; const psFormName: String);
implementation
Uses HTMLer, IAG_Server_Info, SysUtils;
Procedure WriterSample;
begin
  with iagin do begin
    WriteHTML('<HTML><BODY>');
    WriteHTML('The time is: ' + FormatDateTime('', now));
    end;
  end;
end;

Writer.dpr
program writer;
uses
  Forms, htmler, IAG_Server_Info,
  main in 'main.pas' {formMain},
  writer_sample in 'writer_sample.pas';
{$R *.RES}
begin
  Application.Initialize;
  Portcullis_Init;
  RegisterIAGHTMLHook('Sample', WriterSample, nil);
  Application.CreateForm(TformMain, formMain);
  {Every Web Solution Builder App needs at least one form, this is a dummy form}
  Application.Run;
end.
```

➤ *Listing 2*

Inheriting From TIAG_Base: "Placeholders"

TIAG_Base is inherited from TGraphicControl, and only renders itself graphically at design time. At run time, TIAG_Base will render to HTML only, and will not waste resources or time generating a graphical image on the form. Placeholders are also read only, and currently cannot accept input. In a future release they will be enhanced to allow input.

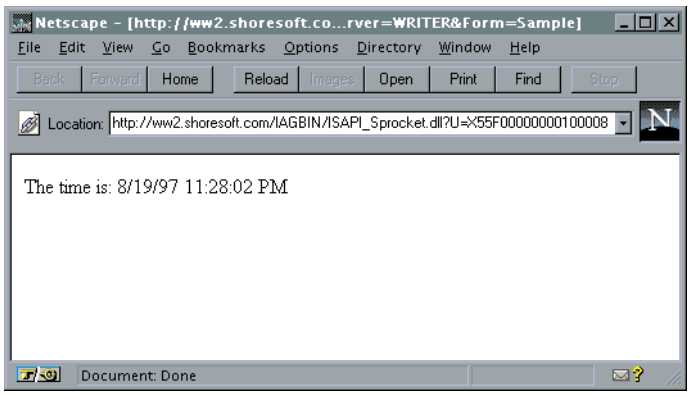
The design time image is not critical nor required. The control is utilized by placing it on a form and sizing it appropriately for its output. Thus, it holds a place for the output, thes are referred to as Placeholder Controls.

This is the best method of creating new controls because they are designed to be as lightweight and fast as possible. Placeholders are also ideal for components which do not have sensible counterparts as Delphi controls. For example, bulleted lists are a standard part of HTML, however there are no

controls in Delphi corresponding to this functionality.

Let's take a quick overview of TIAG_Base, since this is the base for all placeholder controls. The import ant declarations of TIAG_Base are shown in Listing 3.

The minimal implementation of a placeholder control requires the following steps. First, inherit a new class from TIAG_Base; then, override and implement the WriteSelf function and finally, register the control to the Delphi palette.



➤ *Figure 4*

➤ *Listing 3*

```
TIAG_Base = class(TGraphicControl)
protected
  property Font: Tfont read Ffont write SetFont;
  property FontSize: integer read FiFontSize write FiFontSize;
  { 0 = Default Font Size - Do no font output
    These are here if you want to publish these. It will be your responsibility
    to render their attributes if you do publish these }
  procedure ClearRect;
  { Can be called in DesignPaint. Will clear area and fill with parent color }
Public
  Property Text: String write SetText;
  { Do not use the text property, it conflicts with the inherited one. This is
    used to issue an exception if you try to use it }
  Procedure DesignPaint; Virtual;
  { Since these controls only paint at design time, override this method to draw
    it. This method will only be called at design time, so you need not worry
    about this. If you call then inherited DesignPaint it will clear the region
    and fill it with clWindow. You are not required to call the inherited
    DesignPaint. To output, merely use the control's canvas }
  Function WriteSelf: string; Virtual; Abstract;
  { This function will be called when it is the controls turn to write itself
    out. It must return the HTML representation in the result. The control will
    be given the width and height specified in the Width and Height properties.
    The HTML returned should not exceed this size when rendered. If size varies,
    you should make the control bigger than the output will be }
Published
  property Visible;
  { Controls whether or not control is rendered to HTML.
    WriteSelf will not be called if false, however a place will be rendered in
    the final output where the control resides }
```

➤ *Listing 4*

```
Type
  TIAG_List = class(TIAG_Base)
  private
  protected
    procedure DesignPaint; override;
  public
    Function WriteSelf: string; Override;
  published
  end;
```

➤ *Listing 5*

```
function TIAG_List.WriteSelf;
var i: Integer;
begin
  if Numbered then result := '<ol>'
    else result := '<ul>;
  for I := 0 to Items.Count - 1 do
    AppendStr(result, '<LI>' + Items[i] + '</LI>' + #13 + #10);
  if Numbered then AppendStr(result, '</ol>')
    else AppendStr(result, '</ul>');
end;
```

Let's build a simple example (Listing 4). We mentioned earlier that bulleted lists have no corresponding Delphi control. We will build a bulleted list placeholder control for our example. The first step is to create a new unit, and inherit from TIAG_Base. We will also override the DesignPaint and the WriteSelf methods.

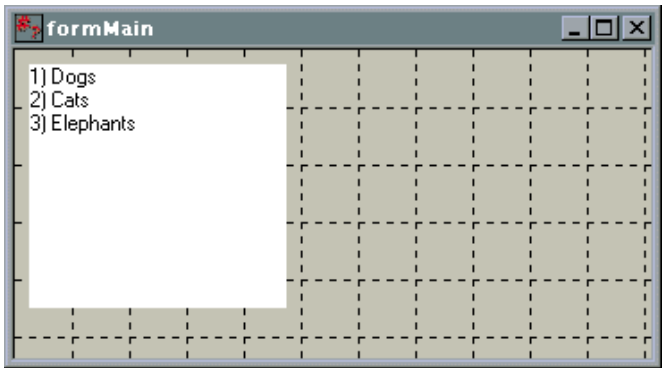
Before we go any further, we need to add some properties so that we can describe the features of an HTML list. These properties are: the items in the list and Bulleted or Numbered. I will add them to TIAG_List as Items (TStringList) and Numbered (Boolean). I will not cover adding these properties and their associated code as they consist of standard Delphi techniques. The complete source code is available in Listing 7. We still have to implement the two methods we overrode in the beginning, WriteSelf and DesignPaint.

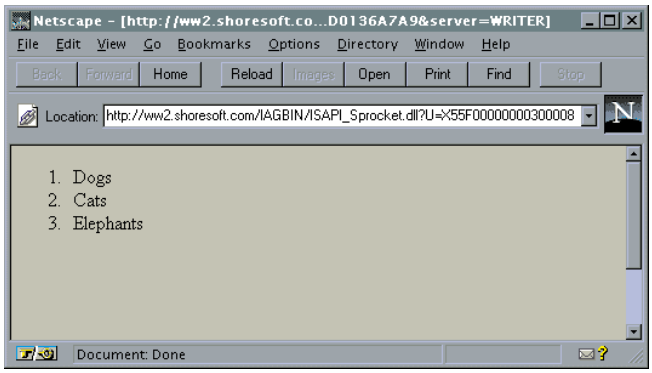
WriteSelf

In the WriteSelf function (Listing 5) the task is to generate the HTML for the control, and return it as the result. The output is dynamic based on the properties that the user has set. In the above example we first test to see if they have selected a numbered list or a bulleted list. Based on the property of Numbered, we set the result to <ol> (The HTML tag to begin a numbered list) or <ul> (The HTML tag to begin a un-numbered list (ie bulleted list)). From there on, we will use the AppendStr procedure on the Result variable to continue to build the complete HTML output.

Tip: AppendStr is much more efficient than using string concatenation on an individual variable. For example: AppendStr(Result, 'More') is better than Result := Result + 'more'.

After we have determined the list type, we will then write out each individual item. For each item we will prefix the text with <li> (HTML for begin list item) and suffix it with (HTML for end list item). To make the output readable (ie if you select View | Source in your browser) for debugging





➤ *Figure 5 (left)*
Figure 6 (right)

purposes, we will also suffix each item with an end of line sequence (CR + LF). Finally, we need to terminate the list. This is done by using the slash form of the begin tag that we used earlier. Thus we again test to see if it is numbered or bulleted, and add either `</ol>` or `</ul>`.

DesignPaint

`DesignPaint` is not required to be implemented, however it gives the programmer a better visualization of what the form will look like at run time. It is generally a good idea to implement `DesignPaint`, however, if you do not, the default `DesignPaint` will fill the space of the control. The output of `DesignPaint` need not look exactly like the actual HTML will (besides, HTML looks different in different browsers), but it should provide a decent representation.

In our `DesignPaint` we first call the inherited `DesignPaint` to provide a textured background. Then we iterate through the items and create a string with either a number prefix, or an asterisk (our representation of a bullet), and use `TextOut` to write it on the controls Canvas. Finally, we use `FrameRect` to draw a border on the control. Figure 5 shows what `TIAG_List` looks like at design time and Figure 6 shows what it looks like in the browser. If we do a `View | Source`, this is what we see (excerpt of section):

```
<ol><LI>Dogs</LI>
<LI>Cats</LI>
<LI>Elephants</LI>
</ol>
```

Completed TIAG_List

Our control is now complete! All that remains is to register it and

➤ *Listing 6*

```
procedure TIAG_List.DesignPaint;
var i: Integer;
    s: string;
begin
  inherited DesignPaint;
  with Canvas do begin
    for i := 0 to Items.Count - 1 do begin
      if Numbered then s := IntToStr(I + 1) + ') + Items[i]);
        else s := '*') + Items[i];
      TextOut(0, i * TextHeight('Ty'), s);
    end;
    FrameRect(Rect(0, 0, Width - 1, Height - 1));
  end;
end;
```

➤ *Listing 7*

```
Unit IAG_List;
interface
Uses HTMLer, SysUtils, Classes;
Type
  TIAG_List = class(TIAG_Base)
  private
    FbNumbered: Boolean;
    FslstItems: TStringList;
    procedure SetItems(Value: TStringList);
  protected
    procedure DesignPaint; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    Function WriteSelf: string; Override;
  published
    property Items: TStringList
      read FslstItems write SetItems;
    property Numbered: Boolean
      read FbNumbered write FbNumbered;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('IAG Placeholders', [TIAG_List]);
end;
constructor TIAG_List.Create;
begin
  inherited Create(AOwner);
  FslstItems := TStringList.Create;
  Height := 24;
  Width := 116;
end;
destructor TIAG_List.Destroy;
begin
  FslstItems.Free;
  inherited Destroy;
end;
procedure TIAG_List.SetItems;
begin
  FslstItems.Assign(Value);
  Invalidate;
end;
function TIAG_List.WriteSelf;
var i: Integer;
begin
  if Numbered then result := '<ol>'
    else result := '<ul>';
  for I := 0 to Items.Count - 1 do
    AppendStr(result, '<LI>'+Items[i]+'</LI>'+#13+#10);
  if Numbered then AppendStr(result, '</ol>')
    else AppendStr(result, '</ul>');
end;
procedure TIAG_List.DesignPaint;
var i: Integer;
begin
  inherited DesignPaint;
  with Canvas do begin
    for i := 0 to Items.Count - 1 do
      TextOut(0, i * TextHeight('Ty'),
        sIIF(Numbered, IntToStr(i)+ ') ', '*') + Items[i]);
    FrameRect(Rect(0, 0, Width - 1, Height - 1));
  end;
end;
```

```
TiagControl = class
  clas: TClass;
  { Class of control to register }
  OutputType: TiagOutputType;
  { Type of output which control renders
    outpHTML: Outputs raw HTML, outpImageBitmap: Outputs a Bitmap }
  ImageSource: TiagImageSource;
  { imgsNone: Is not an Image
    imgsHandle: Image has a Handle
      Return handle in .Bitmap
    imgsFile: Image is in a file
      .FileName will contain a filename, save the bitmap to this }
  ControlType: TiagControlType;
  { Type of control
    ctypDisplay: Display only. Does not accept input.
    CtypInput: Not yet supported, ctypSubmit: Not yet supported }
  AutoLayout: Boolean;
  { If true, control will be positioned in page and given room by its width and
    height. If false control will be vertically placed only if Default is True }
  ProcRender: TProcRender;
  { This procedure is called when the control needs to render itself.
    Page: This is the TIAG_Region which is being generated.
    ctrl: This is the instance of the control which is requested for rendering
    strm: if not nil, the control must send its output to this stream
    sFile: if strm is nil control should write its output to filename specified.
    * strm and File are mutually exclusive, if strm is nil, then sFile will
    contain a name. If strm is not nil, then FileName should be ignored.
    * For controls of type outpHTML, strm will never be nil }
  ProcSetData: TProcSetData;
  ProcSubmit: TProcSubmit;
  { These two pointers are for Submit and interactive controls. This
    functionality will be completed in a future release of Web Solution Builder }
  constructor Create; Dynamic;
  { Standard Delphi Constructor }
end;

Supporting Types for the Extension API
TiagOutputType = (outpHTML, outpImageBitmap);
TiagImageSource = (imgsNone, imgsHandle, imgsFile);
TiagControlType = (ctypDisplay, ctypInput, ctypSubmit);
TProcRender =
  Procedure(page: TIAG_Region; ctrl: Tcontrol; var rinfo: TRenderInfo);
```

➤ *Listing 8*

```
iagc := TiagControl.Create; { Create new instance of TiagControl for register routine }
with iagc do begin
  clas := Timage; { Tell it that we are registering TImage }
  OutputType := outpImageBitmap; { TImage outputs a bitmap }
  ImageSource := imgsHandle; { render procedure will pass back a Bitmap Handle }
  ControlType := ctypDisplay; { It is a display control }
  ProcRender := RenderImage; { This is a pointer to the render procedure. This is
    the procedure which will return the bitmap handle }
  ProcSetData := nil;
  ProcSubmit := nil; { Procedure is Dispaly only, these procedures are not used
    for Display Controls }
  RegisterIAGControl(iagc); { Register the control. This step passes the
    TiagControl instance to the Drawbridge Extension API and allows it to be uses
    in a Web Solution Builder Application }
end;
```

➤ *Listing 9*

```
TRenderInfo = record
  Bitmap: Hbitmap;
  { outpImageBitmap : imagsHandle, Used to return a bitmap handle }
  FileName: String;
  { outpImageBitmap : imagsFile, procedure need to write to filename that is
    passed in FileName }
  strm: Tstream; { outpHTML outputs to stream }
end;
```

➤ *Listing 10*

compile it into the component palette. Again, since these are standard Delphi techniques, I will not cover this here. The complete source is available in Listing 7.

Descending From TIAG_Base Descendants

In addition to inheriting from `TIAG_Base`, you may wish to inherit from descendants of `TIAG_Base`. An example of this would be to inherit from `TIAG_Applet` and add properties to control specific functionality for a specific Java applet.

Extending Delphi Controls

An extension to a Delphi control can be implemented without the need for source code of the control being supported. This method should be used when the control already contains a great deal of functionality that could not be easily rewritten or encapsulated, such as charts and grid.

To extend an existing control, you need to describe the control, provide input/output, and register it with Web Solution Builder. For our example, we will extend the `TImage` and `TDBImage` to be supported. These controls are already supported, but are internally done the same as you see here, through the `Drawbridge Extension` API.

The first step is to describe the control. This is done by creating an instance of `TiagControl`, setting it's properties, and finally registering it. `TiagControl` is shown in Listing 8. Let's take the first step and create the `TiagControl` object for `TImage` (Listing 9).

The registering of the control should be done in the initialization section of the unit. When you need support for a control in an application, be sure to add the unit in a uses clause in the application somewhere, otherwise the initialization section will never be called, and the control will not be supported.

Render Procedure

The second step is to define the output. This is done by creating a render procedure. The render procedure is specified in the `TiagControl`. The render procedure prototype is as follows:

```
TProcRender = Procedure(page:
  TIAG_Region; ctrl: TControl;
  var rinfo: TRenderInfo);
```

You will need to create a procedure with a compatible parameter list. When the render procedure is called, page will specify the region which is doing the rendering and

➤ *Listing 11*

```
procedure RenderImage;
begin
  rinfo.Bitmap :=  TImage(ctrl).
   Picture.Bitmap.Handle;
   { Return the bitmap handle
     to the TImage}
end;
```

```
Unit iag_images;
Interface
Uses
  Controls, Drawbridge, HTMLer;
procedure RenderDBImage(page: TIAG_Region; ctrl: TControl;
  var rinfo: TRenderInfo);
procedure RenderImage(page: TIAG_Region; ctrl: TControl;
  var rinfo: TRenderInfo);
{ Multiple controls can share same Render Procedure but would need to detect
  class of ctrl. Normally it's easier to use unique Render procs }
Implementation
Uses DBCtrls, ExtCtrls;
procedure RenderDBImage;
begin
  rinfo.Bitmap := TDBImage(ctrl).Picture.Bitmap.Handle; { handle to TDBImage }
end;
procedure RenderImage;
begin
  rinfo.Bitmap := TImage(ctrl).Picture.Bitmap.Handle; { handle to Timage }
end;
var iagc: TiagControl;
initialization
  iagc := TiagControl.Create;
  with iagc do begin
    clas := TDBImage; { Class that we are registering }
    OutputType := outpImageBitmap; { It outputs a bitmap }
    ImageSource := imgsHandle; { render procedure passed back a Bitmap Handle }
    ControlType := ctypDisplay; { It is a display control }
    ProcRender := RenderDBImage;   { Render procedure }
    ProcSetData := nil; { Dispaly only, this proc not used for Display Controls }
    ProcSubmit := nil;  { Display only, this proc not used for Display Controls }
    RegisterIAGControl(iagc); { Register the control }
  end;
  { ... See comments above... }
  iagc := TiagControl.Create;
  with iagc do begin
    clas := Timage;
    OutputType := outpImageBitmap;
    ImageSource := imgsHandle;
    ControlType := ctypDisplay;
    ProcRender := RenderImage;
    ProcSetData := nil;
    ProcSubmit := nil;
    RegisterIAGControl(iagc);
  end;
end.
```

➤ *Listing 12*

control will specify the instance of the control which needs to be rendered. `Rinfo` is an input/output class which is passed by address. A class is used so that the `TProcRender` can be expanded in functionality while maintaining backward compatibility with all versions of `Drawbridge` extensions. `Rinfo` may contain information which you will need, to perform the render, and is also used to return data from the `Render` procedure. `Rinfo` is of type `TRenderInfo` and `TRenderInfo` is defined in Listing 10.

We already specified `RenderImage` in step one as our render procedure, so we merely need to write a render procedure with this name now (Listing 11). The complete listing for `IAG_Images` is shown in Listing 12.

Chad Z. Hower (aka Dr.Pepper, czhower@shoresoft.com) is a Principal Development Consultant at Shoreline Software. In his spare time, Chad likes to program. He has the license plates *Delphi2* and *CPPBldr* on his vehicles!