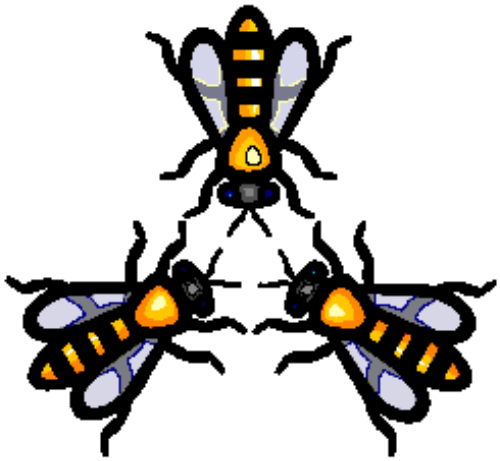
Volume 8, Number 6 • June 2004
www.bcbjournal.org
ISSN 1093-2097

EnCoded Communications Group

C++BUILDER Developer's Journal

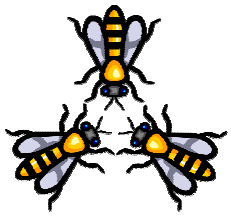
A Monthly Publication Offering Tips & Techniques for Borland C++Builder



IN THIS ISSUE:

Welcome to the New Journal

By Damon Chandler



Dear Fellow Developers,

As many of you know, C++ at Borland has been through some dramatic changes in the last year, changes which have affected us all in one way or another. The Journal is no exception.

The C++Builder Developer's Journal was launched seven years ago by Kent Reisdorph and Tim Gooch, and it was first published by The Cobb Group. In mid 1999, Kent began producing the Journal under Reisdorph Publishing, and approximately three years later, these duties were passed to David Bridges (Bridges Publishing). In December 2003, the production tasks were split: Susan Culligan of Baseline Grid Publications handled the publication details and Dave continued his role as Editor-in-Chief. Now, as the Journal embarks upon its 85th issue, these duties have passed to me.

I've been with the Journal for nearly four years—Kent brought me on as an author in September of 2000, and Dave kindly promoted me to a Contributing Editor in 2002. As I now take on the task of production, my goal is not only to keep the Journal alive and strong, but to bring a sense of *community* to the project. If you think about it, the Journal's readers comprise one of the largest groups of C++Builder developers around. We are, in fact, a community of users who are linked by this common product. I'd like to strengthen this link.

To this end, the Journal has a new, revamped website, http://bcbjournal.org, designed to facilitate your ability to access the Journal. We've also added online forums, http://forums.bcbjournal.org, which allow you to and connect with other members of the community, comment on the Journal's content, and submit article proposals (I sincerely encourage you to publish and share your tips and techniques by writing for the Journal). The format of the Journal's articles has also been revamped: Starting with this issue, HTML versions of the articles are available in an easier-to-read print-preview-type format. The new issues also contain easier-to-read code listings, print-quality screenshots, and references to related articles, where applicable. In addition, we've added a new Developer's Poll that allows you to compare your opinions to those of other members of the community, and in the works is a Q&A section in which our editors answer reader-submitted questions. Remember, this is your journal, so any suggestions for improvements are greatly appreciated.

I'd like to personally welcome you to the C++Builder Developer's Journal, and I thank you for your continued support.

Best regards,

Damon Chandler
Editor-in-Chief
editor@bcbjournal.org

Debugging Aids

By Malcolm Smith

How much time do you spend debugging your code? In simple applications the ratio of debugging to productive coding is probably quite low, but as the projects become more and more complex these figures can change radically. This article is going to discuss several aids I use on a daily basis to help resolve logic problems and often better fine tune certain regions of my code.

All of the utilities mentioned in this article rely on an external viewer to capture the output. I personally use DEBUGVIEW from SYSINTERNALS [1]; it's free and has many great features. The reason I use this approach is because the tools presented here are generally used for applications that have no GUI (such as COM DLLs and Win32 Service applications). More importantly, these utilities are often used in multi-threaded code (GUI and non-GUI) where I need to see the output immediately rather than during post-analysis of a generated text file. (The generation of text-based log files from multiple threads is a topic for a future article.)

Helper categories

The helpers discussed in this article fall into three categories:

1. General output
2. Assertion Logging
3. Tracer / Profiler

General Output

The Win32 API already contains a simple debugging method called `OutputDebugString()` defined as:

```
VOID OutputDebugString(
  LPCTSTR lpOutputString
 );
```

The purpose of this method is to send a message to the application or system debugger.

When you run your application through C++Builder's F9 option, it is running in the context of a debugger. All calls made to `OutputDebugString()` will be captured by the IDE's debugger. The strings are visible in the Event Viewer, accessible via the View | Debug Windows | Event Viewer menus; or by pressing Ctrl+Alt+V. When your application is running outside of the IDE the debug messages can be captured by external debugging applications such as DEBUGVIEW.

`OutputDebugString()` has one serious drawback—you cannot explicitly pass an `AnsiString`. What if you want to output the value of one or more variables? A first attempt might come up with something like:

```
AnsiString Msg = "Value: " + IntToStr(x);
OutputDebugString(Msg.c_str());
```

The problem with this approach is that you end up having to write a minimum of two lines of code and each of those extra variable declarations will pollute your release builds. My replacement, known as `AnsiOutputDebug()`, looks like this:

```
inline void __fastcall AnsiOutputDebug(
  AnsiString Message)
{
OutputDebugString(Message.c_str());
}
```

As you can see, the method takes an `AnsiString` parameter which means I get all the benefits of the various overloaded constructors that it provides in a single call. The previous example now becomes:

```
AnsiOutputDebug("Value: " + IntToStr(x));
```

The next thing we need to do is remove all traces of this function call from our product when producing a Release Build. This can be accomplished via some macro magic. The following code snippet turns our `OutputDebugString()` into a comment during a Release Build:

```
#ifdef _DEBUG
  inline void __fastcall AnsiOutputDebug(
    AnsiString Message)
  {
  OutputDebugString(Message.c_str());
  }
#else
  #define mjf_dslash(s)  s##s
  #define mjf_comment    mjf_dslash(/)

  #define AnsiOutputDebug(p)  mjf_comment
#endif
```

This macro ensures that `AnsiOutputDebug()` is used only in Debug Builds. Release Builds will replace each call to `AnsiOutputDebug()` with a comment. Can you work out how?

- `AnsiOutputDebug` is converted to `mjf_comment`
- `mjf_comment` is converted to `mjf_dslash(/)`
- `mjf_dslash` is converted to `/##/` which the pre-processor converts to `//` (the start of a comment).

I exclusively use `AnsiOutputDebug()` because I am assured the code is converted to a comment when I build my applications in Release Mode.

If you were to run a test project in Release Mode using `OutputDebugString()` you will find the messages are still being generated (you can see them being captured by DEBUGVIEW). In time-critical applications I can't afford to have the application wasting time doing things that are not required.

By the way, a small tip for you: If you decide to use the above-mentioned DEBUGVIEW application for all monitoring of debug strings, then be aware you will not see any output if you run your application within the IDE. I once was caught by this while debugging a DLL using a host application that I was running within the IDE.

Assertions

An assertion is a runtime check to ensure that a particular logical state is as expected. Simple examples include checking that a pointer is not `NULL` or that the value of a variable is within certain limits. If the runtime check evaluates to `TRUE` then no action is taken, otherwise the error condition is reported. How this error is reported depends on the approach used.

The standard C/C++ runtime provides `assert()` as well as the macros `_ASSERT()` and `_ASSERTE()`. I won't go into all the specifics of these, but they can be about read at MSDN—see [2] and [3]. I highly recommend you read these links to get a full appreciation of the options available.

So if I'm not going to talk about the above assertion methods, what am I going to talk about? My own macros of course. I have two macros: `qassert()` ("quiet assert") and `dassert()` ("debugger assert"). Here are the macros including the conversion to comments in a Release Build:

```
#ifdef _DEBUG

#define _quiet_assert(__cond, \
 __file, __line) \
  ShowMessage("Condition: " + \
    AnsiString(__cond) + \
    "\nFile: " + AnsiString(__file) + \
    "\nLine: " +  IntToStr(__line))

#define qassert(p) ((p) ? (void)0 :
  _quiet_assert(#p, __FILE__, __LINE__))

#define _dbgr_assert(__cond, \
  __file, __line) \
  AnsiOutputDebug("Condition: " + \
    AnsiString(__cond) + \
    ", File: " + AnsiString(__file) + \
    ", Line: " + IntToStr(__line))

#define dassert(p) ((p) ? (void)0 :
  _dbgr_assert(#p, __FILE__, __LINE__))

#else

  #define mjf_dslash(s) s##s
  #define mjf_comment   mjf_dslash(/)
  #define qassert(p)    mjf_comment
  #define dassert(p)    mjf_comment

#endif
```

In debug builds the `qassert()` and `dassert()` macros are translated to `_quiet_assert()` and

_dbgr_assert(), respectively. You'll also notice dassert() makes use of the AnsiOutputDebug() function presented earlier.

The qassert() and dassert() macros are essentially the same: qassert() is designed to call ShowMessage() and dassert() calls AnsiOutputDebug(). For this reason I'll explain only how qassert() works.

Let's start with qassert() itself:

```
#define qassert(p) ((p) ? (void)0 :
  _quiet_assert(#p, __FILE__, __LINE__))
```

This macro makes use of the ?: operator which acts as an if-then-else statement. If the expression (p) is TRUE then (void)0 is selected by the preprocessor, otherwise _quiet_assert(#p, __FILE__, __LINE__) is used. The first statement is translated to a no-op and the second passes p as a literal string to _quiet_assert() along with the name of the source file and the current line number. The _quiet_assert() macro is then translated into a call to ShowMessage().

In many cases I place qassert() calls where I want nothing more than a message box to be displayed indicating the failed condition, the file-name and the line number. This macro is simple enough for my needs and it saves me the trouble of adding the extra header and library dependencies associated with _ASSERTE().

As for dassert(), I use this when I need to perform the same kind of assertion checking in non GUI applications such as COM DLLs and Win32 Service Applications.

I should point out something very important at this stage: assert() will abort your application after displaying the error condition. It was designed this way because a program's behavior is undefined if the state of the application is invalid. I prefer to use my macro and not cause the application to terminate. The biggest reason for this is that I often have several links to COM objects (or other resources) and I want to make sure everything is shutdown in a controlled manner.

Tracer / Profiler

Profilers are not actually debugging tools, but I've lumped it into this article because the class I am about to describe serves the purpose of tracing your code's logic and it provides timing information for each executed method (or distinct block of code).

The class is called TTAT (Turn Around Time). It was born when I needed to track multiple threads within a DLL to determine where a bottleneck was located. I have used external profilers in the past that did not require manual insertion of method calls or macros into your code but I have since moved away from them because I always found that the code executed much slower than under normal conditions. Since tracking down issues such as deadlocks is easier to reproduce when the code is running at full throttle, I prefer to use my own methods. **Listing A** provides the class definition.

Listing A: Definition of TTAT

```
class TTAT
{
private:

  AnsiString     FTitle;
  bool           FInclusive;
  DWORD          FMinToReport;
  bool           FCondition;
  bool           FLogInOut;
  bool           HRPCAvail;
  LARGE_INTEGER FStart;
  DWORD          FTickStart;

public:
  TTAT(AnsiString ATitle, DWORD MinToReport,
       bool AInclusive, bool ACondition,
       bool ALogInOut = false);

  ~TTAT(void);
};
```

You'll notice in **Listing A** that there are no member methods to call on this class. So how is it used? It uses the *Resource Acquisition Is Initialization* (RAII) design pattern. Simply put, we use the constructor and destructor of the class to do the work for us when it is constructed and destructed (goes out of scope) respectively. Consider the following simplified example:

```
void __fastcall TfrmMain::Foo1(void)
{
TTAT tat("Calling Foo2", 100, false,
           true, false);

// call Foo2() and perform more work
}

void __fastcall TfrmMain::Foo2(void)
{
// a lengthy task occurs here
}
```

Ignoring the constructor parameters for now (we'll come back to them) what we have here is a class that is constructed at the beginning of `Foo1()` and implicitly destructed when this same method goes out of scope. A second way to use this class is as follows:

```
void __fastcall TfrmMain::Foo1(void)
{
for(int i = 0; i < 100; i++)
  {
  if(condition1 == true)
    {
    TTAT tat("test 1", 100, false,
           true, false);

    // code for test 1 here
    }

  if(condition2 == true)
    {
    TTAT tat("test 2", 100, false,
           true, false);

    // code for test 2 here
    }
  }
}
```

In this situation we have created two explicit blocks of code within the `for` loop. This construct is useful when you want to isolate certain sections of code within the same method.

Remember I said this class was helpful for both tracing code as well as performing timing operations? Well, in this last example we will be able to trace the program logic (when `condition1` and `condition2` are `TRUE`) as well as determine the execution time of the code between the respective curly braces. It's time to see how this all works, starting with the constructor.

```
TTAT(AnsiString ATitle,
     DWORD MinToReport,
     bool AInclusive,
     bool ACondition,
     bool ALogInOut = false);
```

`ATitle` is used to help distinguish one log entry from another.

`MinToReport` is the minimum execution time to report. If the time taken to execute the block of code is below this value (see the next parameter) then no log entry would be made. This comes in handy when profiling multi-threaded code. Just place these objects in the suspect locations and let this parameter filter out all of the speedy execution paths.

`AInclusive` is used in conjunction with `MinToReport`. If this value is `TRUE` then a log entry will be made if the execution time is greater-than-or-equal-to `MinToReport`; otherwise it will be made if the execution time exceeds `MinToReport`.

`ACondition` also controls whether or not a log entry is made. If you have no specific conditions pass `TRUE`. Any expression resulting in a Boolean return value can be used for this parameter.

Finally, `ALogInOut` instructs the class to log when the block of code was entered and when it returned. This is extremely useful for logging multiple nested methods and recursive functions.

Most of the private member variables of `TTAT` are used to copy the values passed into the constructor for later use within the destructor. The remaining variables are used for calculating the timing details.

TTAT supports two methods of measuring execution time. Any system running a high resolution performance counter (HRPC) with Windows 95 and above will be able to take advantage of two API calls: QueryPerformanceCounter() and QueryPerformanceFrequency(). All other systems will fall back to GetTickCount() for their results. Here's how the constructor uses these functions:

```
TTAT::TTAT(AnsiString ATitle,
  DWORD MinToReport, bool AInclusive,
  bool ACondition, bool ALogInOut)
{
FTitle = ATitle.Unique();
FMinToReport = MinToReport;
FInclusive = AInclusive;
FCondition = ACondition;
FLogInOut = ALogInOut;

// best choice
HRPCAvail =
  QueryPerformanceCounter(&FStart);

// next best
if(!HRPCAvail)
  FTickStart = ::GetTickCount();

if(FLogInOut)
  AnsiOutputDebug("<<<<< In: " + FTitle);
}
```

The first five lines copy the constructor parameters. (A special note should be made about the copying a unique instance of an AnsiString in multiple threads due to reference counting issues.) The next line checks to see if an HRPC is available. If it is available then HRPCAvail is set to TRUE and FStart is assigned the counter's present value. If no HRPC is available then TTAT reverts to GetTickCount().

So what is an HRPC? Unlike GetTickCount() which has a resolution of around 10 ms, an HRPC has a much higher resolution that's equal to the value returned by a call to QueryPerformanceFrequency(). This value indicates the number of counts performed per second and it cannot change while the system is running.

The last line in the constructor outputs a message (if FLogInOut is TRUE) to indicate that the block of code being monitored is about to execute.

After TTAT's constructor has completed your block of code will execute as normal. Once the block of code goes out of scope, TTAT will also go out of scope, causing the destructor to be called:

```
TTAT::~TTAT(void)
{
double TimeDiff;

if(HRPCAvail)
  {
  LARGE_INTEGER FStop;
  QueryPerformanceCounter(&FStop);

  // get the HRPC frequency
  LARGE_INTEGER FQPF;
  QueryPerformanceFrequency(&FQPF);

  // convert to milliseconds
  TimeDiff = static_cast<double>
    (FStop.QuadPart - FStart.QuadPart) *
    1000 / static_cast<double>
      (FQPF.QuadPart);
  }
else
  {
  DWORD FTickEnd = ::GetTickCount();
  TimeDiff = FTickEnd - FTickStart;
  }

if(FCondition && (
   (FInclusive && TimeDiff >= FMinToReport)
   || (TimeDiff > FMinToReport)
  ))
  AnsiOutputDebug(FTitle + " TAT = " +
    FloatToStrF(TimeDiff, ffGeneral, 15, 4)
    + " ms");

if(FLogInOut)
  AnsiOutputDebug(">>>>> Out: " + FTitle);
}
```

The destructor determines the method used to calculate the time elapsed by first checking if HRPCAvail is TRUE. If it is, then the current counter's value is determined along with the system's counter frequency by using QueryPerformanceCounter(). This result is then used to calculate the elapsed time with a conversion to milliseconds.

Calculating the elapsed time requires a quick look at `LARGE_INTEGER`, the data-type used by the performance counter functions. WINNT.H defines `LARGE_INTEGER` as a structure containing a `LONGLONG` variable named `QuadPart`. `LONGLONG` is a `typedef` for `__int64`. When there is no HRPC available `TTAT` resorts to using `GetTickCount()`; in this case there is no conversion to milliseconds required.

The destructor next checks if a message needs to be sent to the debugger based on the parameters passed to the constructor. If a message is to be output, the elapsed period is formatted using the `FloatToStrF()` method.

Finally, if the user of `TTAT` needs to track nested calls, another message is sent to the debugger to indicate which `TTAT` object has gone out of scope.

Let's have a quick look at one of the examples provided in the demo application accompanying this article. **Listing B** provides a typical recursive approach to calculating the factorial of a number (I've set the test value to 3 to keep the log short so feel free to play with larger values).

At this point I'll encourage you to visit the sample application for more examples. It contains several buttons that perform different timing/tracing scenarios. You can view the results by using C++Builder's Event Viewer or DEBUGVIEW as mentioned earlier.

Conclusion

In this article we've looked at several aids to more easily track program flow, produce some basic profiling information and, more importantly, turn all of these into comments when producing Release Builds of our products.

The tools provided are useful in all types of applications—single/multi-threaded, GUI- or non-GUI-based. In future articles I hope to explore other techniques I use to trace complicated projects but for now I hope you'll find the information in this article useful in debugging and profiling your current projects.

You can download the code for this article from http://www.bcbjournal.org.

Contact Malcolm at msmith@bcbjournal.org.

References

1. http://tinyurl.com/3yauv
2. http://tinyurl.com/34bg9
3. http://tinyurl.com/2h9gv

Listing B: An example recursive trace

```
void __fastcall TfrmMain::btnTest2Click
(TObject *Sender)
{
AnsiOutputDebug("-> factorial(3) = ?");

int Result = Factorial(3);

AnsiOutputDebug("-> Result = " +
  IntToStr(Result));
}

int __fastcall TfrmMain::Factorial(int n)
{
TTAT tat("Factorial:" + IntToStr(n), 0,
         true, true, true);

if(n < 2)
  {
  AnsiOutputDebug("Factorial(1)=1");
  return 1;
  }

return n * Factorial(n-1);
}
```

Share your thoughts about the Journal with other readers by using our online forums: http://forums.bcbjournal.org.

A TDataSource Component Editor

by Bob Swart

In the January 2004 issue, I wrote about C++Builder property editors, and demonstrated how we can write our own property editor in C++ (or Delphi) and then add them to the IDE. This time, I'll show how we can enhance the design-time power of C++Builder even further by creating component editors.

Component Editors

Component editors are like property editors; they are used to enhance the integrated development environment of Delphi and C++Builder. And, as with property editors, they are basically derived from a single base class where some abstract methods need to be overridden and redefined in order to give the component editor the desired behaviour. In contrast to property editors however, component editors are component-specific, not property-specific, entities. They are bound to a particular component type, and they are generally executed by a click of the right mouse button on the component (when dropped on a form). This way of activation is a bit different than property editors, but other than that, the process of writing your own component editor is essentially the same.

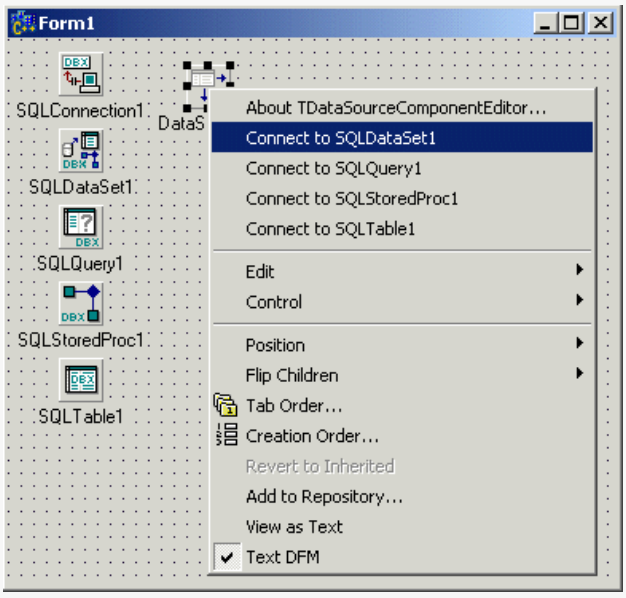
A component editor is created for each component that is selected in the form designer based on the component's type (see also `GetComponentEditor()` and `RegisterComponentEditor()` in `DesignIntf.pas`). When the component is double-clicked, the `Edit()` method is called. When the context menu for the component is invoked, the `GetVerbCount()` and `GetVerb()` methods are called to build the menu. If one of the verbs is selected, `ExecuteVerb()` is called. `Copy()` is called whenever the component is pasted to the clipboard. You only need to create a component editor if you wish to add verbs to the context menu, change the default double-click behaviour, or paste an additional clipboard format. The interface definition for the `IComponentEditor` interface contains the six virtual methods (`Edit`, `ExecuteVerb()`, `GetVerb()`, `GetVerbCount()`, `PrepareItem()`, and `Copy()`), and as with a property editor, we can override any of these six virtual methods to build the special behavior inside our component editor.

Default Component Editor

Apart from the general `TComponentEditor` type, there is a default component editor—called `TDefaultEditor`–that is used by most components (unless another component editor is installed to override the default one). The `TDefaultEditor` class implements `Edit()` to search the properties of the component and to generate the (or navigate to an already existing) `OnCreate`, `OnChange` or `OnClick` event (whichever it finds first), or the first alphabetic event handler that's available.

Whenever the component editor modifies the component it must call the `Modified()` method of the IDE's designer to inform the designer that (the component on) the form has been modified.

Figure A



TDataSource component editor.

If we only use the component editor to display some information (like a general about box, for example) there is no need to inform the designer.

Example: TDataSourceComponentEditor

With this information at hand, we can build a useful component editor. Consider the `TDataSource` component, that lives for one main purpose: to be connected to a `TDataSet` (or `TDataSet`-derived) component and act as connector between this `TDataSet` and other data-aware components (including other `TDataSets`).

Although we can use the Object Inspector to assign a value to the `DataSet` property, sometimes it's quicker if you do not have to switch from your form or data module to the Object Inspector, but if you can simply assign a value to the `DataSet` property by connecting it to one of the available `TDataSets` on the current form or data module. This is the behaviour that I want to implement in my `TDataSource` Component Editor (called `TDataSourceComponentEditor`).

First, we need to decide which base class to use. For the `TDataSource` component, it appears that some (default) behavior is already implemented if you double-click on the component, so the `TDefaultEditor` is the best choice as parent class.

We then need to override and implement a number of methods. Let's start with `GetVerbCount()`, which returns the number of menu entries that appear when we right-click on the `TDataSource` component. I want at to include one menu entry for an About box. And for every `TDataSet` (or derived) component I would like to have a dynamic menu entry that says "Connect to ..." followed by the name of this `TDataSet` component. This means that we can simply right-click on a `TDataSource` component and it will show a "Connect to ..." option for all available datasets (which makes it quick and easy to connect them to the `TDataSource`). In order to find out how many datasets are available, we need to look at the current form or data module. The component itself is owned by this form or data module, so we only have to look at the owner of the current component, and then walk through all components owned by the owner. This is defined by the owner's `Components` array, which holds `ComponentCount` components (counting from 0 to `ComponentCount` minus 1).

Once we've defined how many menu entries to show (returned by the `GetVerbCount()`), it's not difficult to return the individual menu entries (with `GetVerb()`). Even the actual `ExecuteVerb()` is straightforward: just walk through the components of the owner, find the right `TDataSet` component (specified by the index number)

Listing A: DSCompEdit.h

```
#ifndef DSCompEditH
#define DSCompEditH

#include "DesignIntf.hpp"
#include "VCLEditors.hpp"

class TDataSourceComponentEditor:
  public TDefaultEditor
{
  typedef TDefaultEditor inherited;

public:
  virtual void __fastcall Edit(void);
  virtual int __fastcall GetVerbCount(void);
  virtual AnsiString __fastcall GetVerb(int index);
  virtual void __fastcall ExecuteVerb(int index);

public:
  #pragma option push -w-inl
  inline __fastcall virtual
    TDataSourceComponentEditor(
      Classes::TComponent* AComponent,
      _di_IDesigner ADesigner):
    TDefaultEditor(AComponent, ADesigner) { }
  #pragma option pop

public:
  #pragma option push -w-inl
  inline __fastcall virtual
    ~TDataSourceComponentEditor(void) { }
  #pragma option pop
};

#endif  // DSCompEditH
```

and assign it to the DataSet property of the current TDataSource component.

As final touch, I've also overridden the Edit() method in order to show an About box if you double-click on the TDataSource component (note that this would hide the default behaviour, so I've also called the TDefaultEditor::Edit() method to compensate for that). The definition of the TDataSourceComponentEditor, with the four virtual methods that I've implemented, can be seen in **Listing A**.

The actual implementation of the TDataSourceComponentEditor can be seen in **Listing B**. Note that in order to determine whether or not a component is a TDataSet (or derived) component, I use dynamic_cast, which returns a valid pointer if the component is a TDataSet (or a derived type) but a NULL pointer otherwise. This is an effective test that proves very powerful, since it will, in one step, identify TDataSet or derived classes (which are the only classes that can be assigned to the DataSet property of the TDataSource component).

Finally, notice the Register() method (which must be placed in the namespace that corresponds to the file-name), where we call the RegisterComponentEditor() method to register the TDataSourceComponentEditor for the TDataSource component.

After you've installed this component editor, for example in the dclusr.dpk package (see the article of January 2004 for more installation details), you can see the component editor in action by right-clicking on the TDataSource component. See, for example, **Figure A** where I've used four different dbExpress datasets to illustrate the behavior.

Listing B: DSCompEdit.cpp

```
#include "DSCompEdit.h"
#include "DB.hpp"
#pragma package(smart_init)

void __fastcall
  TDataSourceComponentEditor::Edit(void)
{
  MessageDlg("TDataSourceComponentEditor",
    mtInformation, TMsgDlgButtons() << mbOK, 0);
  TDefaultEditor::Edit(); // inherited
}

int __fastcall
  TDataSourceComponentEditor::GetVerbCount(void)
{
  TComponent* ComponentOwner = Component->Owner;
  int DataSets = 1; // first one for About...
  if (ComponentOwner)
  {
    for (int i=0;
      i < ComponentOwner->ComponentCount; i++)
      if (dynamic_cast<TDataSet*>
        (ComponentOwner->Components[i]))
        DataSets++;
  }
  return DataSets;
}

AnsiString __fastcall
  TDataSourceComponentEditor::GetVerb(int index)
{
  if (index == 0) return
    "&About TDataSourceComponentEditor...";
  else
  {
    TComponent* ComponentOwner = Component->Owner;
    int DataSets = 0;
    if (ComponentOwner)
    {
      for (int i=0;
        i < ComponentOwner->ComponentCount; i++)
      if (dynamic_cast<TDataSet*>
        (ComponentOwner->Components[i]))
      {
        DataSets++;
        if (DataSets == index)
          return "&Connect to " +
            (ComponentOwner->Components[i])->Name;
      }
    }
  }
}
```

(Continued on next page)

Although it doesn't save a whole lot of time compared to moving to the Object Inspector, this component editor enables you to extend the current functionality. Possible ideas that I've consid-

ered with but never implemented are to append the name of the dataset with additional information, like the number of records, whether the dataset is currently open ("live") or closed, and—in case of the `TClientDataSet`—if the data are from a local or remote source. This would lead to functionality that the Object Inspector can not easily offer, and is left as an exercise for the reader (perhaps a topic for a follow-up article if you've enjoyed this one—feel free to let me know).

Conclusion

In this article, I've demonstrated how we can build component editors, by building a `TDataSource` component editor that can connect to any `TDataSet` component on the same form or data module. I hope to have shown that Property and Component Editors make powerful additions to the C++Builder IDE, and they can increase the RAD experience by automating or supporting tasks at design-time.

The full source for this article's example program can be downloaded from http://www.bcbjournal.org.

Contact Bob at
http://www.drbob42.com.

Listing B (Continued): DSCompEdit.cpp

```
void __fastcall TDataSourceComponentEditor::
  ExecuteVerb(int index)
{
  if (index == 0) Edit(); // about box
  else
  {
    TComponent* ComponentOwner =
      Component->Owner;
    int DataSets = 0;
    if (ComponentOwner)
    {
      for (int i=0; i < ComponentOwner->
        ComponentCount; i++)
      if (dynamic_cast<TDataSet*>
        (ComponentOwner->Components[i]))
      {
        DataSets++;
        if (DataSets == index)
        {
          (dynamic_cast<TDataSource*>(Component))
           ->DataSet = dynamic_cast<TDataSet*>
           (ComponentOwner->Components[i]);
          Designer->Modified();
        }
      }
    }
  }
}

namespace Dscompedit
{
   void __fastcall PACKAGE Register()
   {
      RegisterComponentEditor(
        __classid(TDataSource),
        __classid(TDataSourceComponentEditor)
        );
   }
}
```

Using GDI+, Part II: Bitmaps

By Damon Chandler

Last month, I demonstrated how to set up GDI+ for use in a VCL application. This month, I'll show you how to work with bitmaps in GDI+ and how to use GDI+ drawing routines to effect output to a `TBitmap` object.

Raster- vs. vector-based rendering

Drawing anything to a digital device ultimately boils down to highlighting certain pixels and turning off others. A digital image is created by defining a particular pattern of pixels, one which our visual system interprets as a representation of a scene [1].

In Windows, there are two main ways to render a digital image to an output device (e.g., a screen, printer). One approach is to load/create a bitmap and then transfer this bitmap to the device via, e.g., the `BitBlt()` GDI function or the `Graphics::DrawImage()` GDI+ method; this technique is typically called *bitmapped* or *raster-based rendering*. The other approach is to use a higher-level "language" to communicate to the output device the lines, shapes, and other graphics primitives that you want drawn; this latter technique is called *vector-based rendering*.

This month, I'll cover raster-based rendering with GDI+ bitmaps; and, next time, we'll tackle vector-based rendering (with metafiles). Raster-based rendering is typically faster than vector-based rendering, but this speed comes at a cost. Namely, bitmaps lack scalability—if you want to draw a larger version of a bitmapped image, you'll need to perform some form of interpolation.

GDI+ bitmaps

Recall that there are three types of bitmaps in the standard GDI: *device-dependent bitmaps* (DDBs); *device-independent bitmaps* (DIBs); and *DIB section bitmaps*.; see [2]. Also recall that whereas the pixel format—i.e., the number of bits with which each pixel is represented—of a DDB depends on the current screen settings, and whereas a DIB is not a true graphics object (meaning that you can't select it into a device context), a DIB section bitmap suffers from neither of these limitations. Accordingly, when using the standard GDI, DIB section bitmaps are the preferred variety for caching and drawing raster-based graphics. Because GDI+ relies on the standard GDI as its backend, all bitmaps in GDI+ are effectively DIB section bitmaps.

You create a (DIB section) bitmap in GDI+ by using the `Bitmap` class. The `Bitmap` class is a descendant of the GDI+ `Image` class that provides

Table 1: Commonly used methods of the Bitmap class

Method	Description
`UINT GetWidth()`	Returns the width of bitmap, in pixels.
`UINT GetHeight()`	Returns the height of the bitmap, in pixels.
`PixelFormat GetPixelFormat()`	Returns a `PixelFormat`-type constant [3] that identifies the number of bits used to represent each pixel.

(Continued on next page)

Table 1 (Continued): Commonly used methods of the Bitmap class

Method	Description
`Status LockBits( const Rect* rect, UINT flags, PixelFormat format, BitmapData* lockedBitmapData )`	Retrieves/specifies a pointer to a buffer to the bitmap's pixels; see "Accessing the pixels" later in this article.
`Status UnlockBits( BitmapData* lockedBitmapData )`	Effects to the bitmap any changes made to the buffer previously retrieved/specified via the `LockBits()` method; see "Accessing the pixels" later in this article.
`Status Save( const WCHAR* filename, const CLSID* clsidEncoder, const EncoderParameters* encoderParams = NULL )`	Saves the bitmap to a file specified by `filename` and in the file-format specified by `clsidEncoder`; the optional and file-format-specific `encoderParams` parameter can be used to specify attributes with which the file should be saved (e.g., compression quality for JPEGs).

bitmap-specific functionality such as creating a bitmap from a DDB or DIB, and accessing and modifying the bitmap's pixels. **Table 1** lists the `Bitmap` class's key methods (many of which are inherited from the `Image` class); see [4] for a full listing.

Notice that **Table 1** doesn't list a method for loading a bitmap from a file. As I'll discuss next, loading a bitmap is typically done by using the `Bitmap` constructor.

Bitmap constructors

The `Bitmap` class provides 10 constructors which are used to create GDI+ bitmaps from various sources and/or with various attributes. For applications in which a bitmap is used as an in-memory drawing surface, you typically need a bitmap of a specific pixel format and of a specific size. Here's the declaration of the `Bitmap` constructor that you'd use in that case:

```
void Bitmap(
  INT width,
  INT height,
  PixelFormat format
);
```

The `width` and `height` parameters specify the dimensions of bitmap, in pixels; and the `format` parameter specifies the bitmap's pixel format [3]. For example, the following code snippet demonstrates how to create a 512x512, 24-bits-per-pixel (bpp) bitmap:

```
// create a 24-bpp 512x512 bitmap
gdp::Bitmap bitmap(512, 512,
  PixelFormat24bppRGB);
GDPCheck(bitmap.GetLastStatus());
// other code here...
```

Note that all of the pixels of the newly created bitmap are initially set to zero. (Also, recall that the `GDPCheck()` function was defined in last month's article.)

Another common requirement is the ability to load a bitmap from a file; here's the `Bitmap` constructor to do that:

```
void Bitmap(
  const WCHAR* filename,
  BOOL useIcm = FALSE
);
```

The `filename` parameter specifies the name of .BMP file to load, and the optional `useIcm` parameter that specifies whether or not the method should acknowledge the bitmap's embedded color-profile information, if present. **Listing A** demonstrates how to load a bitmap from a file; the `Bitmap` object is created on the heap—so that it remains alive throughout the form's lifetime—and it is destroyed in the form's destructor *before* calling the `GdiPlusShutdown()` function.

The Bitmap class also provides a constructor to load a bitmap from a resource instead of from a file. To load a bitmap from a resource, you'd use the following Bitmap constructor:

```
void Bitmap(
  HINSTANCE hInstance,
  const WCHAR* bitmapName
 );
```

The hInstance parameter specifies the handle to the (application or library) instance that contains the resource identified by the bitmapName parameter. (This method is similar to the LoadFromResourceName() method of the TBitmap class.)

The Bitmap class also provides constructors that allow you to create a bitmap based on an existing DDB, DIB, or even a raw array of pixels:

```
// creation based on a DDB
// (or DIB section bitmap)
void Bitmap(
  HBITMAP hbm,
  HPALETTE hpal
 );

// creation based on a DIB
void Bitmap(
  const BITMAPINFO* gdiBitmap-
Info,
  VOID* gdiBitmapData
 );

// creation based on a raw array
void Bitmap(
  INT width,
  INT height,
  INT stride,
  PixelFormat format,
  BYTE* scan0
 );
```

For creation based on a DDB, the Bitmap constructor takes two parameters: a handle to the DDB (hbm; this can also be a handle to a DIB section bitmap) and a handle to the DDB's palette (hpal, which can be NULL if the DDB doesn't use a palette). The Bitmap object won't take ownership of the DDB; rather, it creates it's own bitmap representation (DIB section) based on the DDB.

For creation based on a DIB, the constructor also takes two parameters: a pointer to a DIB's header and color table (gdiBitmapInfo), and a pointer to the DIB's pixels (gdiBitmapData). Again, the Bitmap object doesn't assume ownership of the DIB.

For creation based on a raw array of pixels, the Bitmap constructor takes five parameters: integers that specify the width and height of the image, an integer that specifies how many bytes

Listing A: Loading a bitmap

```
//---------------------------------------------
// in header...
//---------------------------------------------
#include <memory>
class TForm1 : public TForm
{
// other stuff...

private:
  ULONG_PTR gdp_token_;
  std::auto_ptr<gdp::Bitmap> bitmap_;
  void __fastcall LoadBitmap(WideString fname);
};

//---------------------------------------------
// in source...
//---------------------------------------------
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  // initialize the GDI+ library
  GDPCheck(
    gdp::GdiplusStartup(&gdp_token_,
      &gdp::GdiplusStartupInput(), NULL
      ));
}

__fastcall TForm1::~TForm1()
{
  // delete the Bitmap
  delete bitmap_.release();

  // close the GDI+ library
  gdp::GdiplusShutdown(gdp_token_);
}
```

(Continued on next page)

of the array should be used for each row (`stride`), a `PixelFormat`-type variable [3] that specifies the pixel-format of the array (`format`), and a pointer to the beginning of the array (`scan0`).

The `Bitmap` class also provides four other constructors which are used to create a bitmap based on an `IStream`, based on a DirectDraw surface, based on the attributes of a GDI+ `Graphics` object, or based on an icon. I won't discuss these constructors here, but you can see [5] for more information.

Drawing GDI+ bitmaps to the screen

Now that you know how to create GDI+ bitmaps, let me show you draw them to the screen. As I mentioned last month, this is accomplished by using the `DrawImage()` method of the GDI+ `Graphics` class [6]. There are actually 16 variants of the `DrawImage()` method, which allow you to draw the bitmap at (integer and floating-point) locations, using (integer and floating-point) scaling factors, and using a callback function.

For example, to draw a bitmap in its original size at location (`x_pos`, `y_pos`) in your form's client area, you'd do the following:

```
// create a Graphics object associated
// with the form
gdp::Graphics graphics(Handle);
GDPCheck(graphics.GetLastStatus());

// draw the bitmap to the form
// at location (10, 20)
int const x_pos = 10;
int const y_pos = 20;
graphics.DrawImage(
  &bitmap, x_pos, y_pos
  );
```

Here, `bitmap` is assumed to be a pre-created `Bitmap` object. Similarly, here's the code to draw a scaled version of the bitmap:

```
// create a Graphics object associated
// with the form
gdp::Graphics graphics(Handle);
GDPCheck(graphics.GetLastStatus());
```

Listing A (Continued): Loading a bitmap

```
void __fastcall TForm1::
  LoadBitmap(WideString fname)
{
  // load the bitmap from a file
  bitmap_.reset(
    new gdp::Bitmap(fname.c_bstr())
    );

  // if the bitmap did not
  // load successfully...
  gdp::Status const res =
    bitmap_->GetLastStatus();
  if (res != gdp::Ok)
  {
    // delete the invalid bitmap
    delete bitmap_.release();
    // throw an exception
    throw EGDIPlusError(res);
  }

  // NOTE: You should check that
  // bitmap_.get() != NULL before
  // trying to access the bitmap
  // elsewhere in your code
}
```

```
// draw the bitmap to the form
// at location (10, 20) and in
// 20% of its original size
int const x_pos = 10;
int const y_pos = 20;
graphics.DrawImage(
  &bitmap, x_pos, y_pos,
  // new width
  bitmap.GetWidth() / 5,
  // new height
  bitmap.GetHeight() / 5
  );
```

And, to draw just a portion of the bitmap, you'd use the following approach:

```
// create a Graphics object associated
// with the form
gdp::Graphics graphics(Handle);
GDPCheck(graphics.GetLastStatus());

// draw the first 75 rows of the bitmap
// to the form at location (10, 20) and
// in 150% of its original size
gdp::Rect const RDest(
  // target (x, y)
  10, 20,
```

```
  // target width
  1.5f * bitmap.GetWidth(),
  // target height
  1.5f * bitmap.GetHeight()
  );
graphics.DrawImage(
  &bitmap,
  // target (destination) rectangle
  RDest,
  // source coordinates (chunk of
  // the bitmap to draw)
  0, 0, bitmap.GetWidth(), 75,
  // source chunk is in pixel coords
  gdp::UnitPixel
  );
```

This latter version of the `DrawImage()` method operates in a fashion similar to the `StretchBlt()` GDI function. Note, however, that although I've used integer-valued coordinates in these examples, the `DrawImage()` method also allows you to specify floating-point-valued coordinates.

Drawing to GDI+ bitmaps

As with DDBs and DIB section bitmaps (but not DIBs), you can render output to a GDI+ bitmap by using normal drawing routines. In the standard GDI, you'd effect output to a bitmap object by selecting the bitmap into a memory device context (DC); you'd then call your drawing function, specifying a handle to that memory DC as the "target" DC. In GDI+, you use a similar approach. Instead of creating a memory DC, you create a `Graphics` object that's associated with the bitmap.

The following code snippet demonstrates how to create a `Graphics` object that's associated with a bitmap and how to use that `Graphics` object to render output to the bitmap (and then to the screen); the result of this code is depicted in **Figure A**:

```
// create a 24-bpp 128x256 bitmap
int const bmp_cx = 128;
int const bmp_cy = 256;
gdp::Bitmap bitmap(bmp_cx, bmp_cy,
  PixelFormat24bppRGB);
GDPCheck(bitmap.GetLastStatus());

// create a Graphics object that's
// associated with the bitmap
gdp::Graphics bmp_graphics(&bitmap);
GDPCheck(bmp_graphics.GetLastStatus());

// create a brush with a horizontal
// white-to-black gradient that
// repeats after bmp_cx units
gdp::LinearGradientBrush brush(
  // gradient starting point
  gdp::Point(0, 0),
  // gradient ending/restarting point
  gdp::Point(bmp_cx, 0),
  // white to start
  gdp::Color(255, 255, 255),
  // black to end
  gdp::Color(0, 0, 0)
  );
GDPCheck(brush.GetLastStatus());

// fill the bitmap using the brush
bmp_graphics.FillRectangle(&brush,
  0, 0, bmp_cx, bmp_cy);

// create a Graphics object that's
// associated with the form and
// then draw the bitmap to the form
gdp::Graphics graphics(Handle);
GDPCheck(graphics.GetLastStatus());
graphics.DrawImage(&bitmap, 0, 0);
```

Figure A

By creating a GDI+ `Graphics` object that's associated with a GDI+ `Bitmap`, you can first draw to the bitmap, and then (using another `Graphics` object) draw the bitmap to the screen.

Note that this technique of first rendering output to a bitmap and then rendering the bitmap to the screen is known as *double buffering*; the bitmap serves as a so-called "back buffer" and the screen serves as the front or "primary buffer." Double buffering is useful for situations in which repetitive, complex rendering is required: Instead of placing the time-consuming drawing code within, for example, the form's `OnPaint` event handler, it makes more sense (assuming your drawing code doesn't change) to execute the expensive code only once, storing the output in a bitmap, and then drawing this bitmap to the screen.

As note that the `Graphics` constructor which takes a `Bitmap` pointer as it's single parameter (as used in this code) will fail—yielding a generic out-of-memory error message—if the `Bitmap`'s pixel format is `PixelFormatUndefined`, `PixelFormatDontCare`, `PixelFormat1bppIndexed`, `PixelFormat4bppIndexed`, `PixelFormat8bppIndexed`, `PixelFormat16bppGrayScale`, or `PixelFormat16bppARGB1555`. This is a documented limitation [6].

Accessing the bitmap's pixels

One of the main advantages of using a DIB section bitmap over a DDB is that the standard GDI allows direct access to the DIB section's pixels. Accordingly, GDI+ also permits direct access to a `Bitmap`'s pixel via the `Bitmap::LockBits()` method.

To use the `LockBits()` method, you specify: (1) the portion of the bitmap's pixels to which you want access, (2) the type of access desired (reading, writing, or both), (3) the format of the pixels (more on this shortly), and (4) a pointer to a `BitmapData` object, which the `LockBits()` method will fill with information about the bitmap's pixels. And, once you're done examining and/or modifying the pixels, you use the `Bitmap::UnlockBits()` method to effect the changes.

For example, to convert a 32-bpp bitmap to grayscale, you'd use the `Bitmap::LockBits()` and `UnlockBits()` methods as follows:

```
// load the bitmap from a file
gdp::Bitmap bitmap(L"c:/pict_32bpp.bmp");
GDPCheck(bitmap.GetLastStatus());

// define the area of the bitmap to lock
UINT const bmp_cx = bitmap.GetWidth();
UINT const bmp_cy = bitmap.GetHeight();
gdp::Rect bmp_rect(0, 0, bmp_cx, bmp_cy);

// grab a pointer to the pixels
gdp::BitmapData bmp_data;
GDPCheck(bitmap.LockBits(
  // portion of the bitmap to lock
  &bmp_rect,
  // read-/write-access flags
  gdp::ImageLockModeRead |
  gdp::ImageLockModeWrite,
  // desired format of output pixels
  bitmap.GetPixelFormat(),
  // filled with bitmap data (including
  // a pointer to the pixels)
  &bmp_data
  ));
unsigned char* p_pixels =
  static_cast<unsigned char*>
    (bmp_data.Scan0);

for (UINT y = 0; y < bmp_cy; ++y)
{
  // grab an RGBQUAD* to each row
  RGBQUAD* p_scanline =
    reinterpret_cast<RGBQUAD*>(
      p_pixels + (y * bmp_data.Stride)
      );
  for (UINT x = 0; x < bmp_cx; ++x)
  {
    // compute grayscale
    unsigned short const gray_val =
      0.299f * p_scanline[x].rgbRed +
      0.587f * p_scanline[x].rgbGreen +
      0.114f * p_scanline[x].rgbBlue;

    // change the pixel
    p_scanline[x].rgbRed =
      p_scanline[x].rgbGreen =
        p_scanline[x].rgbBlue =
          (gray_val > 255) ?
            255 : gray_val;
  }
}

// unlock the bitmap (commits any
// changes made to the pixels)
bitmap.UnlockBits(&bmp_data);
```

Note that the desired pixel format, which is specified via the third parameter to `LockBits()`, doesn't have to be the same as the pixel format of the bitmap. However, for 16-bpp, 24-bpp, and 32-bpp bitmaps, `LockBits()` will fail if you specify an output pixel-format that requires a color table (`PixelFormat1bppIndexed`, `PixelFormat4bppIndexed`, or `PixelFormat8bppIndexed`). Furthermore, if you request the data in a different format than that of the bitmap (e.g., if you request `PixelFormat24bppRGB` for a 32-bpp bitmap), GDI+ will have to create a temporary array to accommodate your new format. Although the documentation for the `LockBits()` method states that a temporary array is *always* created, I've found that this is not the case when you request the pixels in the same format as that of the bitmap. In short, it's generally best to avoid format conversions when using `LockBits()`—i.e., pass the return value of the `Bitmap::GetPixelFormat()` as the third parameter to `LockBits()`.

Saving bitmaps

After you've created and manipulated a GDI+ bitmap, you can save the output to a file by using the `Bitmap::Save()` method. The `Save()` method is actually inherited from the `Image` class, which allows you to save the bitmap to any file-format for which GDI+ provides an encoder (BMPs, GIFs, JPEGs, PNGs, and TIFFs). In order to save a bitmap to specific format, however, you need to specify the `CLSID` of the encoder; this is accomplished by using the `GetImageEncoders()` GDI+ function. **Listing B** provides utility functions for retrieving the `CLSID` of an encoder given its MIME-type string, and for saving a bitmap using that encoder.

Listing B: Saving a bitmap

```
void GetEncoderCLSID(
   CLSID& enc_clsid,
   WideString mime_type
  )
{
  UINT num_encs, size_encs;
  // get the number of installed encoders and
  // the size in bytes required to hold the
  // ImageCodecInfo data for these encoders
  GDPCheck(
    gdp::GetImageEncodersSize(&num_encs, &size_encs)
    );
  if (num_encs < 1)
  {
    throw EGDIPlusError(
      "No GDI+ encoders installed."
      );
  }

  // create a buffer to hold the encoders' data
  unsigned char* const p_buffer = new
    unsigned char[size_encs];
  try
  {
    // cast the buffer to ImageCodecInfo*
    gdp::ImageCodecInfo* const p_encs =
      reinterpret_cast<gdp::ImageCodecInfo*>
        (p_buffer);
    // get the ImageCodecInfo for the encoders
    GDPCheck(gdp::GetImageEncoders(
      num_encs, size_encs, p_encs
      ));

    // run through the array of ImageCodecInfo,
    // looking for the CLSID of the desired encoder
    UINT idx;
    for (idx = 0; idx < num_encs; ++idx)
    {
      if (WideString(p_encs[idx].MimeType) ==
          mime_type)
      {
        enc_clsid = p_encs[idx].Clsid;
        break;
      }
    }
```

Using GDI+ with TBitmap objects

Now that you know how to use the GDI+ `Bitmap` class, let's switch gears and discuss how to use GDI+ to draw to a `TBitmap` object. In many ways, the `TBitmap` class is easier to use than GDI+'s `Bitmap` class. On the other hand, because the VCL relies on the standard GDI, rendering fancy lines, shapes, and text and performing basic image-processing operations (namely, geometric transformations) is much easier in GDI+.

Fortunately, as C++Builder developers, we get the best of both worlds—you can easily instruct GDI+ to render its output to a `TBitmap` object.

Drawing to a TBitmap using GDI+

The use GDI+ to draw to a `TBitmap` object, you simply use the `Graphics` constructor that takes an `HDC` (handle to a device context) as one of its parameters. Typically, this constructor is used to create a `Graphics` object that sends its output to the DC of a window; however, the DC doesn't have to be a window's DC, it can also be a memory DC that's associated with a `TBitmap` object (i.e., `Bitmap->Canvas->Handle`).

The following code snippet demonstrates how to use GDI+ to render a gradient background and some text to the `TBitmap` object that's held in a `TImage` (`Image1`); the output of this code is depicted in **Figure B**:

```
// grab a reference to the TBitmap
// that's held within Image1 and
// then resize it
Graphics::TBitmap& Bitmap =
  *Image1->Picture->Bitmap;
Bitmap.PixelFormat = pf24bit;
Bitmap.Width = 256;
Bitmap.Height = 256;

// local scope
{
  // create a Graphics object that's
  // associated with the TBitmap
  gdp::Graphics bmp_graphics(
    Bitmap.Canvas->Handle
    );
  GDPCheck(
    bmp_graphics.GetLastStatus()
    );

  // create a brush with a horizontal
  // red-to-blue gradient that
  // repeats after Bitmap.Width units
  gdp::LinearGradientBrush brush(
    // gradient starting point
```

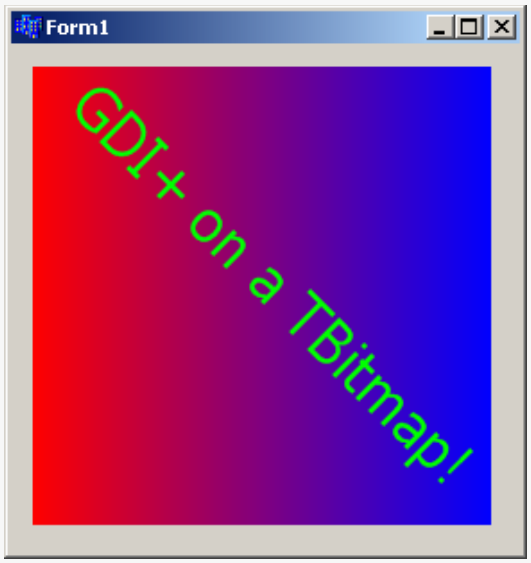
Listing B (Continued): Saving a bitmap

```
    // check that we found the encoder for
    // the specific MIME type
    if (idx == num_encs)
    {
      throw EGDIPlusError(
        "The specified encoder is not installed."
        );
    }
  }
  __finally
  {
    // free the buffer
    delete [] p_buffer;
  }
}

void SaveBitmap(
   gdp::Bitmap& bitmap,
   WideString fname,
   WideString mime_type,
   gdp::EncoderParameters* p_save_options = NULL
  )
{
  // get the CLSID of the encoder
  CLSID enc_clsid;
  GetEncoderCLSID(enc_clsid, mime_type);

  // save the bitmap using the encoder
  // (see [8] for info on using save_options)
  GDPCheck(
    bitmap.Save(fname.c_bstr(), &enc_clsid,
      p_save_options)
    );
}
```

Figure B



By creating a GDI+ `Graphics` object that's associated with the memory DC of a `TBitmap` object, you can use GDI+ to render output to a `TBitmap` object. Here, the `TBitmap` is simply displayed via a `TImage` control.

```
    gdp::Point(0, 0),
    // gradient ending/restarting point
    gdp::Point(Bitmap.Width, 0),
    // red to start
    gdp::Color(255, 0, 0),
    // blue to end
    gdp::Color(0, 0, 255)
    );
GDPCheck(brush.GetLastStatus());

// fill the TBitmap using the brush
bmp_graphics.FillRectangle(&brush,
  0, 0, Bitmap.Width, Bitmap.Height);

// create a 25pt Tahoma font
gdp::Font const font(
  L"Tahoma",            // font name
  25,                   // font size
  gdp::FontStyleRegular // attributes
  );
GDPCheck(font.GetLastStatus());

// draw some rotated green text
// to the TBitmap
bmp_graphics.RotateTransform(
  45.0f, gdp::MatrixOrderAppend
  );
bmp_graphics.DrawString(
  L"GDI+ on a TBitmap!", -1,
    &font, gdp::PointF(25, -30),
    &gdp::SolidBrush(
      gdp::Color(0, 255, 0))
    );
}

// redraw the TBitmap
Image1->Refresh();
```

Note that I declared `bmp_graphics` in a local-scope sub-block so that the `Graphics` object is destroyed before the `TBitmap` object is assigned to the `TImage`. This isn't strictly required for this example because the `TImage` class simply draws the bitmap to the screen; it doesn't draw *to* the bitmap. In general, however, you need to make sure that all GDI+ `Graphics` objects associated with the `TBitmap`'s `Canvas` are destroyed before calling a GDI routine that may alter the bitmap (see [7] for more information on mixing GDI and GDI+ code).

Converting a TBitmap to a GDI+ Bitmap

As I discussed earlier, a very useful feature in GDI+ is the ability to save a bitmap to various formats (see **Listing B**). Although the `TPicture` class is not too far behind in this regard (with the proper third-party libraries), it's often more convenient to use GDI+.

In order to save a `TBitmap` to a specific image-file format using GDI+, you need to create a `Bitmap` object based on the `TBitmap` object; here's the code to do that:

```
gdp::Bitmap* TBitmap2Bitmap(
   Graphics::TBitmap& VCLBitmap
  )
{
  // use the GDI+ Bitmap constructor
  // that takes a handle to a DDB or
  // DIB section bitmap and a handle to
  // the DDB or DIB section's palette
  gdp::Bitmap* p_bitmap =
    new gdp::Bitmap(
      // handle to the GDI bitmap
      VCLBitmap.Handle,
      // handle to the GDI palette
      VCLBitmap.Palette
    );
```

```
  // validate the GDI+ bitmap
  gdp::Status const res =
    p_bitmap->GetLastStatus();
  if (res != gdp::Ok)
  {
    delete p_bitmap;
    throw EGDIPlusError(res);
  }

  // return a pointer the GDI+ bitmap
  // (caller assumes ownership)
  return p_bitmap;
}
```

After the `Bitmap` object is created, you can simply use the technique presented in **Listing B** to save the `Bitmap` to the desired format.

Conclusion

In this article I've demonstrated how to work with GDI+ `Bitmaps` and how to use GDI+ to draw to a `TBitmap`. The source code for this article is available for download from **http://www.bcbjournal.org**. Next time, we'll switch gears and examine GDI+ metafiles and GDI+ vector-based drawing routines.

Contact Damon at **editor@bcbjournal.org**.

References

1. D. Marr and E. Hildreth, "Theory of edge detection," *Proc. R. Soc. Lond. B*, **207**, 1980.
2. D. Chandler, "Printing Bitmaps, Part I," *C++Builder Dev. Journal*, **5** (1), 2001. [link]
3. http://tinyurl.com/2v8zc
4. http://tinyurl.com/222mb
5. http://tinyurl.com/2zg9q
6. http://tinyurl.com/3bxqv
7. http://tinyurl.com/32bvo
8. http://tinyurl.com/27z75

This Month's Developer's Poll

This is a new feature of the Journal which allows you to compare your opinions to those of other members of the C++Builder Developer's Journal community. We will publish the results of this poll in next month's issue, and we will communicate the results of this poll to the C++ folks at Borland.

This month's poll question is…

What direction would you like to see Borland take with their next C++ offering?

a. A cross-platform IDE, compiler, and component framework such as a new, better version of C++BuilderX.

b. A Windows-only IDE, compiler, and an updated version of the VCL framework such as a new, better version of C++Builder 6.

c. A Windows-only IDE, compiler, and support for the .NET framework, such as a C++ variant of Delphi 8.

d. A VI-style editor, with no IDE, a command-line-only compiler, and a non-RAD framework.

Cast your vote online at **http://polls.bcbjournal.org**.

C++BUILDER Developer's Journal

EnCoded Communications Group

C++Builder Developer's Journal (ISSN 1093-2097) is published online monthly by Encoded Communications Group, 66 Lois Lane, Ithaca, NY 14850.

Customer Service: support@bcbjournal.org
Customer Relations (Voice) (607) 227-3757
Customer Relations (Fax) (707) 238-3031

Send all written correspondence to:
EnCoded Communications Group
66 Lois Lane
Ithaca, NY 14850

Editorial: editor@bcbjournal.org
Editor-in-Chief Damon Chandler
Managing Editor Jared Bish
Contributing Editors Bob Swart
Brent Knigge
Malcolm Smith

Price
Personal $49/year
Personal with email delivery of PDF file $59/year
Corporate/Library/Institutional $79/year

This Month's Contributors

Malcom Smith

Contributing Editor Malcolm Smith is owner of MJ Freelancing, which develops custom components and bespoke projects. Malcolm is also Chief Analyst Programmer for Comvision Pty Ltd. designing and implementing security management systems, concentrating on the integration of disparate CCTV and alarm systems as well as streaming digital video into security control rooms. Malcolm can be contacted at **msmith@bcbjournal.org.**

Bob Swart

Contributing Editor Bob Swart (aka "Dr.Bob," www.drbob42.com) is an author, trainer, and consultant who runs his own one-man company called "eBob42" in The Netherlands. Bob, who writes his own "Delphi Clinic" training material, has spoken at Delphi and Borland Developer Conferences since 1993.

Damon Chandler

Damon Chandler develops image processing and graphics-based applications in the Visual Communications Lab at Cornell University, where his research focuses on image compression algorithms. Damon is a co-author of the Windows 2000 Graphics API Black Book, a contributing author of the C++Builder 5 Developer's Guide, and a member of Team Borland (www.teamb.com). Damon can be contacted at **editor@bcbjournal.org**.

Interested in writing for the C++Builder Developer's Journal? Great! We're always on the lookout for new authors with fresh ideas. Your article can be a short as a quick tip or as long as a multipart series. If you have an idea, please don't hesitate to run it by our editors.

To submit a manuscript proposal (essentially, an extended abstract) either e-mail your proposal to editor@bcbjournal.org or use the Manuscript Proposals forum at http://forums.bcbjournal.org. We ask that you please limit your proposal to 250 words.

For more information, please visit http://bcbjournal.org/authors.htm.