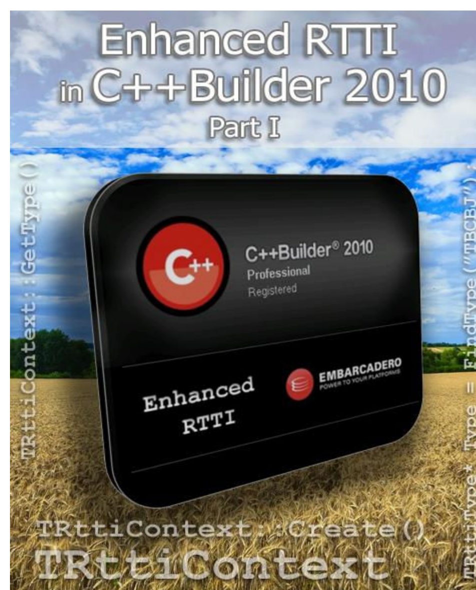
Enhanced RTTI in C++Builder 2010, Part I: Introduction

By Remy Lebeau

Versions: C++Builder 2010



Embarcadero's recently released RAD Studio 2010 contains many new features and productivity enhancements throughout the IDE, code editor, RTL/VCL, and compilers. In this article, I will describe a new feature that many long-time customers have been requesting for a very long time—more detailed VCL-based RTTI (Run-Type Type Information) emitted by the Delphi and C++ compilers (this does not cover C++-specific RTTI, which is a whole different type of RTTI altogether, and is dictated by the C++ language standards).

Foreword

Before I continue, I need to first point out a couple of very important issues that are present in the initial RTM release of C++Builder 2010. At the time of this writing, only the Delphi 2010 compiler can actually emit Enhanced RTTI data. There is no equivalent in the C++ compiler for generating Enhanced RTTI data, however C++ code can consume the Enhanced RTTI data that is generated by Delphi. Also, there are a number of bugs in the new Enhanced RTTI system when used in C++, several of which have been logged in Quality Central so far [1]-[4].

For the rest of this article, I am going to describe how the new Enhanced RTTI is supposed to be used, and describe what does and does not actually work. Embarcadero is planning to release an Update Pack for RAD Studio 2010 soon; several of the issues have been marked as needing fixes in that Update Pack.

What is Enhanced RTTI?

Delphi 2010 and C++Builder 2010 introduce a new Enhanced RTTI system that compliments the older RTTI system used in past versions. The older RTTI is accessed using functions and structure types that are implemented in the TypInfo unit (TYPINFO.PAS in Delphi, TYPINFO.HPP in C++). The new Enhanced RTTI system is implemented as an object-oriented class hierarchy in a new Rtti unit (RTTI.PAS in Delphi, RTTI.HPP in C++).

The new Rtti unit does not (yet?) replace the TypInfo unit, but rather builds upon it. Two of the main structures in the Rtti unit—`TRttiType` and `TValue`—provide access to `PTypeInfo` and `PTypeData` pointers from the TypInfo unit. The TypInfo RTTI is still used for things like DFM streaming and such.

In addition to the functionality already present in the TypInfo unit, the Rtii unit provides additional functionality for obtaining more detailed information about runtime packages and the types that are implemented within them, including unit-scoped enumerations, sets, and records, not just classes. It also provides some basic typecasting of values from one type to another compatible type.

Where the Rtti unit really shines though, is where the TypInfo unit has always lacked—Enhanced RTTI data can be generated for *non-published* fields, methods, and properties! You can even invoke methods on object instances via Enhanced RTTI, as it contains information about the calling conventions, parameter lists, and return types involved. A dispatch mechanism is provided that allows you to specify values for those fields when invoking methods.

Basically, if the compiler can compile something, there is a good chance you can access it via Enhanced RTTI now.

To emit Enhanced RTTI data, two new directives have been added to the Delphi compiler: `{$RTTI}` and

{$WEAKLINKRTTI}. In addition, a new Attributes syntax has been added to the Delphi Pascal language itself (well, more accurately, the .NET-style Attributes syntax found in earlier Delphi.NET versions appears to have now been implemented in Delphi for Win32) to allow developers to include their own custom data in a type's Enhanced RTTI.

Use of both compiler directives and the Attributes syntax are fully documented in RAD Studio 2010's Help system, so I will not cover them.

Establishing an RTTI context

To access the new Enhanced RTTI in your code, you start by instantiating an instance of the TRttiContext structure. This is the only publicly accessible gateway into the rest of the RTTI system. It manages all of the memory and reference counts on any objects that need to be internally allocated while accessing the RTTI data. Do not free any of these objects yourself. Internally, TRttiContext delegates to a private TRttiPool singleton object that owns everything.

Embarcadero's documentation says to use the static TRttiContext::Create() method to initialize a new TRttiContext instance, and then call the TRttiContext::Free() method when you're done using it:

```
#include <Rtti.hpp>
{
  TRttiContext ctx = TRttiContext::Create();
  // use ctx as needed ...
  ctx.Free();
}
```

TRttiContext contains a single data member—FContextToken—that is an IInterface pointer to a TPoolToken interfaced object, which ensures the TRttiPool singleton object remains available for the lifetime of the TRttiContext instance. TRttiContext::Create() returns a new TRttiContext instance whose FContextToken member is initialized to NULL. The TPoolToken object is created on an as-needed basis afterwards.

TRttiContext is a plain structure, not a TObject descendant (several other types in the Rtti unit are TObject descendants, however). As such, it does not need to be allocated on the heap, and its Free() method merely releases the FContextToken member— it does not actually free the memory for the TRttiContext instance itself. If no references to the TRttiPool singleton remain, the pool is freed as well, releasing all of its internally allocated data.

Bug #1

TRttiContext::Create() returns a new TRttiContext instance by value, not by reference or pointer. When a function returns a structure or class by value, the compiler creates a temporary instance in memory and passes it to the function via a hidden input parameter for the function to fill in as needed. However, a bug in the C++ compiler at this time [3][4] causes TRttiContext::Create() to always throw an EAccessViolation exception in the RTL's _IntfClear() function. The temporary TRttiContext instance that is used for the return value of TRttiContext::Create() is not initialized at all, causing the FContextToken member to contain whatever random value was previously stored in that memory space beforehand. If that value is not zero, _IntfClear() ends up trying to decrement a reference count on a non-existent interfaced object and crashes.

Fortunately, there is a simple workaround—declare the TRttiContext variable without calling Create() at all:

```
#include <Rtti.hpp>
{
  TRttiContext ctx;
  // use ctx as needed ...
  ctx.Free();
}
```

The reason this works is because TRttiContext's data member is a Delphi interface pointer. In C++, Delphi interface pointers are always wrapped inside a DelphiInterface-based smart wrapper class (in this case, by the _di_IInterface class). The various DelphiInterface constructors initialize the interface pointer to NULL, which makes calling TRttiContext::Create() redundant even if it functioned correctly.

Fortunately, TRttiContext::Free() is safe to call in C++. But, it is also redundant as it merely releases the FContextToken interface pointer, which is the same thing done by the DelphiInterface destructor.

The previous example can thus be simplified to the following, at least for the time being:

```
#include <Rtti.hpp>
{
  TRttiContext ctx;
  // use ctx as needed ...
}
```

Nobody seems to know why Embarcadero decided to introduce the Create() and Free() methods in TRttiContext. They accomplish the same thing that

the Delphi and C++ compilers are supposed to be doing natively, as Delphi interface pointers are a managed data type in both environments.

RTTI context methods

Once you have a valid `TRttiContext` instance to work with, you have access to the following five public methods:

```
TRttiType* __fastcall
  GetType(void *ATypeInfo);

TRttiType* __fastcall
  GetType(System::TClass AClass);

System::TArray__1<TRttiType*> __fastcall
  GetTypes(void);

TRttiType* __fastcall FindType(
  const System::UnicodeString
    AQualifiedName);

System::TArray__1<TRttiPackage*>
  __fastcall GetPackages(void);

// NOTE: System::TArray_1 is just an alias
// for the System::DynamicArray class.
```

The first overloaded version of `TRttiContext::GetType()` allows you to pass in a `PTypeInfo` pointer from the TypInfo unit. `TRttiContext::GetType()` will locate information about the package/module that owns the RTTI being pointed at by the `PTypeInfo`, wrap everything in a `TRttiType` object, and cache it in the `TRttiPool` singleton object.

The second overloaded version of `TRttiContext::GetType()` does the same thing, except for a class metadata reference instead, such as from the return value of the `TObject::ClassType()` method or the C++ compiler's `__classid()` keyword.

The `TRttiContext::GetTypes()` method retrieves an array of `TRttiType` objects of all top-level data types from all loaded packages/modules that provide RTTI information. Newly allocated objects are cached in the `TRttiPool` singleton object as needed.

The `TRttiContext::FindType()` method takes a fully qualified name of a public type and returns a pointer to its `TRttiType` object (the qualified name must be expressed in Delphi-style dot notation—a unit name followed by a "`.`" character followed by the type name. Do not specify qualified names using C++'s "`::`" notation). If the type is not found, a `NULL` pointer is returned.

The `TRttiContext::GetPackages()` method retrieves an array of `TRttiPackage` objects of all loaded packages that provide RTTI information. Newly allocated objects are cached as needed.

Bug #2

Both overloaded versions of the `TRttiContext::GetType()` method throw an `EAccessViolation` exception in the private `TRttiPool::GetType()` method if the "Build with Runtime Packages" setting is disabled in the Project Options [2]. Internally, `TRttiPool::GetType()` calls the private `TRttiPool::GetPackagesList()` method, which returns a `NULL` pointer that `TRttiPool::GetType()` does not handle correctly. The `TRttiPool::GetPackagesList()` method returns a `NULL` pointer because package type information is not emitted correctly by the C++ linker [1], which prevents all package RTTI information from being correctly located at runtime, even for Delphi-written packages.

The only current workaround for this bug is to enable the "Build with Runtime Packages" option; then everything works as expected. Because of this, you are best-off waiting for Embarcadero to release bug fixes before releasing any production-level code based on this new Enhanced RTTI system.

Conclusions

In this article, I have described the basics of accessing the top-level objects of the new Enhanced RTTI system. In the next article, I will delve further into the kinds of RTTI data and functionality that the `TRttiType` and `TRttiPackage` objects expose.

Contact Remy at **remy@lebeausoftware.org**.

References

1. QC #76875, "InitContext.PackageTypeInfo shouldn't be 0 in a C++ module," http://qc.embarcadero.com/wc/qcmain.aspx?d=76875
2. QC #76877, "AV in TRttiContext::GetType() when Runtime Packages are disabled," http://qc.embarcadero.com/wc/qcmain.aspx?d=76877
3. QC #77431, "AV in TRttiContext::Create()," http://qc.codegear.com/wc/qcmain.aspx?d=77431
4. QC #77436, "AV when a Delphi record containing managed data types is used as a function Result," http://qc.codegear.com/wc/qcmain.aspx?d=77436