Custom Managed Records

Contents

Overview

Records in Delphi can have fields of any data type. When a record has plain (non-managed) fields, like numeric or other enumerated values there is not much to do for the compiler. Creating and disposing the record consists of allocating memory or getting rid of the memory location.

If a record has a field of a type managed by the compiler (like a string or an interface), the compiler needs to inject extra code to manage the Initialization or Finalization. A string, for example, is reference counted so when the record goes out of scope the string inside the record needs to have its reference count decreased, which might lead to de-allocating the memory for the string. Therefore, when you are using such a managed record in a section of the code, the compiler automatically adds a try-finally block around that code, and makes sure the data is cleared even in case of an exception. This has been the case for a long time. In other words, Managed Records have been part of the Delphi language.

Records with Initialize and Finalize Operators

The Delphi record type supports custom Initialization and Finalization, beyond the default operations the compiler does for managed records. You can declare a record with custom Initialization and Finalization code regardless of the data type of its fields, and you can write such custom Initialization and Finalization code. This is achieved by adding specific, new operators to the record type (you can have one without the other if you want). Below is a simple code snippet:

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize (var Dest: TMyRecord);
  end;
```

Remember, you need to write the code for both class methods. For example, when logging their execution or initializing the record value, we are also logging a reference to memory location, to see which record is performing each individual operation:

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
  Dest.Value := 10;
  Log('created' + IntToHex (Integer(Pointer(@Dest))));
end;

class operator TMyRecord.Finalize (var Dest: TMyRecord);
begin
  Log('destroyed' + IntToHex (Integer(Pointer(@Dest))));
end;
```

The huge difference between this construction and what was previously available for records is the automatic invocation. If you write something like the code below, you can invoke both the Initializer and Finalizer, and end up with a try-finally block generated by the compiler for your managed record instance.

```
procedure LocalVarTest;
var
  my1: TMyRecord;
begin
  Log (my1.Value.ToString);
end;
```

with this code you will get a log like:

```
created 0019F2A8

10

destroyed 0019F2A8
```

Another scenario is the use of inline variables, like in:

```
begin
  var t: TMyRecord;
  Log(t.Value.ToString);
```

which gets you the same sequence in the log.

The Assign Operator

The := assignment copies all data of the record fields. While this is a reasonable default, when you have custom data fields and custom Initialization you might want to change this behavior. This is why for Custom Managed Records you can also define an assignment operator. The new operator is invoked with the := syntax, but defined as Assign:

```
type
  TMyRecord = record
    Value: Integer;
    class operator Assign (var Dest: TMyRecord;
                const [ref] Src: TMyRecord);
```

The operator definition must follow very precise rules, including having the first parameter as a reference parameter, and the second as a const passed by reference. If you fail to do so, the compiler issues error messages like the following:

```
[dcc32 Error] E2617 First parameter of Assign operator must be a var parameter of the container type
```

```
[dcc32 Hint] H2618 Second parameter of Assign operator must be a const[Ref] or var parameter of the container type
```

There is a sample case invoking the Assign operator:

```
var
  my1, my2: TMyRecord;
begin
  my1.Value := 22;
  my2 := my1;
```

produces this log (where a sequence number is included to the record):

```
created 5 0019F2A0

created 6 0019F298

5 copied to 6

destroyed 6 0019F298

destroyed 5 0019F2A0
```

Notice that the sequence of destruction is reversed from the sequence of construction.

The Assign Operator is used in conjunction with assignment operations like in the example above, and also if you use an assignment to Initialize an inline variable. Here there are two distinct cases:

```
var
  my1: TMyRecord;
begin
  var t := my1;
  Log(t.Value.ToString);

  var s: TMyRecord;
  Log(s.Value.ToString);
```

this logs:

```
created 6 0019F2A8

created 7 0019F2A0

6 copied to 7

10

created 8 0019F298

10

destroyed 8 0019F298

destroyed 7 0019F2A0

destroyed 6 0019F2A8
```

In the first case, the creation and assignment happen like in regular scenarios with non-local variables. In the second case, there is just a regular initialization.

Passing Managed Records as Parameters

Managed Records can work differently from regular records also when passed as parameters or returned by a function. Here are several routines showing the various scenarios:

```
procedure ParByValue (rec: TMyRecord);
procedure ParByConstValue (const rec: TMyRecord);
procedure ParByRef (var rec: TMyRecord);
procedure ParByConstRef (const [ref] rec: TMyRecord);
function ParReturned: TMyRecord;
```

Each log performs the following operations:

- *ParByValue* creates a new record and calls the assignment operator (if available) to copy the data, destroying the temporary copy when exiting the procedure.
- *ParByConstValue* makes no copy, no call at all.
- *ParByRef* makes no copy, no call.
- *ParByConstRef* makes no copy, no call.
- *ParReturned* creates a new record (via Initialize) and on return it calls the Assign operator, if the call is like the following, and deletes the temporary record:

```
my1 := ParReturned;
```

Exceptions and Managed Records

When an exception is raised, records in general are cleared even when no explicit try, finally block is present, unlike objects. This is a fundamental difference and key to the real usefulness of managed records.

```
procedure ExceptionTest;
begin
  var a: TMRE;
  var b: TMRE;

  raise Exception.Create('Error Message');
end;
```

Within this procedure, there are two constructor calls and two destructor calls. Again, this is a fundamental difference and a key feature of managed records. See the later section on a simple smart pointer based on managed records.

On the other hand, if an exception is raised in the Initializer of a managed record, the matching destructor is not invoked, differently from what happens for regular objects.

Arrays of Managed Records

If you define a static array of managed records, there are initialized calling the Initialize operator at the point declaration:

```
var
  a1: array [1..5] of TMyRecord; // call here
begin
  Log ('ArrOfRec');
```

They are all destroyed when they get out of scope. If you define dynamic array of managed records, the Initialization code is called with the array sized (with SetLength):

```
var
  a2: array of TMyRecord;
begin
  Log ('ArrOfDyn');
  SetLength(a2, 5); // call here
```

Delphi Managed Records in C++

The Initializing and Finalizing managed records automatically are not supported from C++ Builder. The Initialize and Finalize operators are declared in the generated HPP, but are not invoked. The Assignment and the copy constructor will not be correct either.