Writing External Functions

A General Guide

by Claudio Valderrama C.

There seems to be confusion regarding the engine capabilities and restrictions when writing general purpose external functions, a.k.a. User Defined Functions (UDFs). This article tries to show the possibilities available. Rather than provide a complex example function that handles almost everything, I chose instead to take a pedantic approach with trivial examples. Combining possibilities for real life requirements is left to the application developer.

Introduction

A user defined function is a routine written in a dynamic library or shared library, depending on the platform. Of course, the routine must be visible to the process that loads the library, in this case, the Firebird server. In the server's parlance, the library is referred to as a *module* and the routine as an *entry point*.

1- Basic rules for writing UDFs

There are some basic rules for writing UDF's, that I will try to explain briefly. They should be obvious in most cases.

Calling convention

For those who wondered but never dared to ask, a calling convention specifies

- whether parameters are pushed left-to-right in the stack, or vice versa
- whether the caller or the callee is responsible to reset the stack after the callee returns
- the decoration rules (how the linker uses the function's signature to create the binary name).

The basic requirement for the entry point that exposes a function is that it should adhere to the so-called C calling convention that is supported by several compiled languages.

The C calling convention is just one set of rules; Pascal uses another convention and Windows's stdcall is another convention again.

For C itself, there is nothing to do. For C++, the extern "C" directive should be used. For Delphi, the function declaration should append *cdecl* and the function should be listed in the **exports** section.

Thread safety

For Delphi, the main unit should set the global variable System.IsMultiThread to True, in order to have a thread safe memory manager.

Globals

Do not use global variables. The engine may execute UDFs in parallel. You can use global constants, of course.

Dynamic memory management

You can use your preferred allocation and deallocation method for dynamic memory. However, this memory should be confined within the function, for its exclusive use (allocated and deallocated by the UDF). Never pass to the engine a chunk of memory created with a facility specific to the language—for example, C++'s *new* operator. The ib_util file defines the ib_util_malloc function for that purpose and you should use it. More about this when the FREE_IT keyword is explained.

SQL declaration syntax

When declaring external functions in your SQL DDL, there are two alternative syntaxes. The first declares all the input parameters with their respective attributes, then proceeds to declare the returned value with its respective attributes:

```
DECLARE EXTERNAL FUNCTION sql_name
[{ <datatype> | CSTRING (int) [CHARACTER SET charset_name]} [{BY DESCRIPTOR |
BY SCALAR_ARRAY | NULL}]] [, ...]
RETURNS { <datatype> | CSTRING (int) [CHARACTER SET charset_name]} [{BY VALUE
| BY DESCRIPTOR}] [FREE_IT]
ENTRY_POINT '<entryname>'
MODULE_NAME '<modulename>';
```

The second syntax declares all the input parameters with their respective attributes, then proceeds to designate one of those parameters as an output parameter (by its position, starting at one). It causes the engine to provide storage to the UDF, which only has to fill it:

```
DECLARE EXTERNAL FUNCTION sql_name[{ <datatype> | CSTRING (int) [CHARACTER SET
charset_name]} [{BY DESCRIPTOR |
BY SCALAR_ARRAY | NULL}]] [, ...]
RETURNS PARAMETER int
ENTRY_POINT '<entryname>'
MODULE_NAME '<modulename>';
```

In practice, this allows up to 10 input parameters in the first syntax plus the returned value and up to nine input parameters plus one output parameter in the second syntax.

Here, for simplicity, the [, ...] part means that more parameters can be declared, repeating the same previous syntax. Unfortunately, the syntax is harder to describe than to use. Most people do not realize all the possible combinations, and a few of them are invalid or nonsense, anyway. Typically, people are used to passing input parameters *by reference* and returning a result *by value.*

Character sets and collations

UDFs have no notion of collations. They do support character sets, although almost all functions found in the distributed libraries assume ASCII and work with no notion of character sets at all.

Data types

UDFs can receive and return data types native to the engine. For strings, they cab also use a C-compatible, null terminated string named CSTRING.

Parameter-passing protocol

There are different protocols for providing information to the UDF about parameters. The reserved word following the BY keyword in the declaration is the mechanism for defining the protocols and will be explained soon.

You must assume that the input parameters cannot be modified, even if they are passed BY VALUE, unless they are scalars like integers received BY VALUE. This is easily expressed in C++ or Delphi, that can declare constant parameters. Do not use BY VALUE in Delphi, however—see section 2.

2—Mechanisms

Now that we are set up with the most common rules for declaring UDFs and it is established that the C calling convention should be used, we can examine the detail of each part of the allowed syntaxes—the mechanism—the specification of how each parameter is provided to the UDF.

Pascal people are accustomed to passing parameters *by value* and *by reference*. C programmers are used to passing parameters by value and *by pointer*, since some types (built-in arrays, for example) are passed always by pointer. C++ programmers can pass parameters by value, by pointer and by reference. In C++, a reference can exist without being a parameter, but this is beyond our scope.

The concept of mechanism in the engine is not too different, since the same keywords are used, along with others that have proprietary origins. Let's examine them.

BY VALUE

As expected, this is a copy of the data inside the engine. If the UDF was going to change it (not recommended, see section 1), there would not be any effect on the engine's internal data. There would be small risk with scalar types, since their size is known beforehand.

However, if the UDF modifies a string parameter, it is writing in space allocated by the engine, exactly to the input parameter. If the usage leads the UDF to overwrite the original parameter with a longer string, typically disaster happens.

As a general rule, avoid modifying input params of types other than integers and floating point values. Ten years ago, it was observed that this mechanism was unreliable for input parameters, because it led to incompatible, non-portable code. For this to work always, the engine would have to mimic the layout of parameters exactly as a specific compiler would do and of course, the engine has no control on which compiler the application developer will use.

BY VALUE was popular with GDML, the original, proprietary InterBase language, a mixture of declarative and procedural language elements. Since compatibility problems had been observed already in 1994, BY VALUE for input parameters was not allowed in SQL DDL and is deprecated.

For example, in 16 bits, the Borland and the MS compilers offered different representations of the float data type and now it is known that BY VALUE does not work in 64 bits ports made recently. Reserve it for the return value (section 4).

Clearly, a value can not be null and null is not a value so, with parameters received or returned by value, there is no way to signal "unknown value" or SQL NULL.

BY REFERENCE

This is the default mechanism for both input and output parameters and for the UDF's result value in SQL DDL. As a curiosity, being the default, it cannot be declared explicitly. In practice, it means by pointer. C programmers will feel at home with it.

Even though Pascal can use pointer parameters, it was required to use a "type" to define the pointer based on the built-in or custom data type before using it for parameters, making pointer-based parameter declaration in functions cumbersome. But the reference is a hidden pointer, so declaring parameters by reference will do the trick for Delphi.

Do not use "const" in Delphi, since the compiler has the freedom to chose whether to pass the constant parameter by value or by reference. C++ programmers can receive parameters passed by reference from the engine exactly as C does, receiving pointers or using native C++ references. Using either pointers to constant data or constant references is recommended to avoid coding mistakes.

Although the passing of parameters by pointer may be seen as natural way to pair the null pointer with SQL NULL, it was not done for unknown reasons. Simply put, although this method could do so, it does not tell the developer when a parameter is SQL NULL: the parameter will be zero for numeric values and the empty string for CHAR, VARCHAR and CSTRING. Since there is no concept of zero DATE or zero TIMESTAMP, values received in those cases may be surprising. The TIME type has the concept of zero, of course.

BY DESCRIPTOR

This is a proprietary mechanism, known internally as *VMS descriptor*. To avoid name clashes, ibase.h defines the descriptor with the name *paramdsc* and accompanies it with some symbolic constants to explain the data types. In the parameter, the UDF will receive a pointer to an internal structure known as descriptor, carrying information about the data type, the subtype, the precision, the character set and collation, the scale, a pointer to the data itself and some flags that may include SQL NULL signaling. Some of this information overlaps, depending on the data type, since members of this structure are reused for different purposes.

This is not a copy of the parameter but a pointer to the internal structure being used by the engine. You must treat it as constant information always, unless it is the output parameter! Unless treated carefully, it may be a dangerous practice. We will see later that with this mechanism, the engine does not bother to check that the UDF's declared type for the param is the same as the value that will be passed. Use it only if you know what you're doing.

It was allowed since ancient times in GDML, but it never materialised in DDL in Borland builds, perhaps to avoid tech support nightmares. It was included in DDL. starting with Firebird 1.

The Firebird native UDF library named *fbudf* is an example of working with descriptors. More information can be found at http://www.cvalde.net/document/using_descriptors_with_udfs.htm and by reading the source of the library, located in the extlib/fbudf directory inside Firebird CVS.

BY BLOB

This is another proprietary mechanism, known also as *ISC descriptor* and *blob struct*. It cannot be declared explicitly since it is used automatically each time a parameter of type BLOB is declared, unless the declaration includes BY DESCRIPTOR.

If the parameter was received in the internal descriptor (previous case), then the user would have to invoke the API from inside the UDF to manipulate the blob. Then, a compromise was found, a new structure (a wrapper) was developed that is used only to communicate with the UDF. Until Firebird 1, when it was included in ibase.h with the name blobcallback, it was only described in the documentation.

The engine opens the blob—or creates a blob if the parameter is declared as the output parameter—and assigns some internal information to the wrapper's date members. It sets the internal blob descriptor as the "handle" member in the wrapper. To its procedural pointers it assigns the addresses of three call-safe routines, which will allow reentrancy in the engine's code by handling the details transparently. These functions allow the UDF to read a segment, to write a segment and, for stream blobs, to seek into the blob.

It is important that the UDF does not mess with the handle member or disaster may happen. It should be used only for passing to the call-safe functions that need a handle to operate.

For unknown reasons, Borland decided that if the input blob was null, a blob with no contents would be opened, but then it becomes impossible to distinguish between a blob with zero length and a null parameter. If the function returns a blob, again the engine provides the wrapper structure by creating a blob. If the UDF does not write in it, it will have zero length, but there is no way to signal SQL NULL to the engine.

Starting Firebird 1.5, if the UDF makes the blob handle null UDF when there are no more calls to the helper methods to get/set segments and seek, the engine will understand it should behave as if the UDF returned SQL NULL instead of a meaningful blob.

The blob is only opened or created. It is up to the developer to read the segments or to put information by writing segments. It is an efficient mechanism if only the basic information about the blob is needed, since the wrapper will indicate the number of segments, the size of the biggest segment and the total size of the blob for input blobs.

It should be noted that input blobs only can be read and the output blob only can be written. Failure to observe the rule will cause the engine to stop the UDF execution.

BY SCALAR_ARRAY

This is a thoroughly native mechanism that makes sense only if the input will be an array column. Unlike other types, you cannot return an array from the UDF. This is a mechanism for input parameters only.

Firebird arrays

Although many people do not use them, the engine allows both simple and multi-dimensional arrays to be defined, of almost any type allowed by the engine. It was allowed in GDML but never in DDL, perhaps because it was considered irrelevant by Borland. From Firebird 2, it is allowed in DDL.

Arrays are based on a special type of blob, known as streamed blob, as opposed to segmented blobs, that are the ones that DDL can declare. If you want to create a blob of type stream, you will have to use the API.

When an array is defined as part of a table's field set, the engine silently creates the streamed blob. To put values into it, the application developer uses again the API, since SQL and PSQL lack support to work with arrays, other than including individual elements of an array in a query. Through the API, the array is described using SDL (Slice Description Language) and the run-time internal structure is known as Internal Array Descriptor (previously known as Array Description Slice or ADS).

Passing arrays to a UDF

Since you can declare up to an array of 16 dimensions with a big range (number of elements) per dimension, you may cause the engine to use a lot of memory (since it loads all the array at once). Therefore, you should use this feature only when you really need to or, at least, use it knowing its effects.

To pass the values to the UDF, the engine retrieves the whole array in a single chunk in memory and then loads it into a simpler structure, known as Scalar Array Descriptor. If the type of the stored array does not match the type declared by the UDF, the engine performs conversion on each element, converting and copying one value after another into a new binary string. (If you are curious, peruse get_scalar_array() inside jrd/fun.epp).

For example, an array of integers would be converted on the fly to an array of strings if the UDF wants to receive a scalar array of varchar. In the event that the array is SQL NULL, the UDF will receive a *zeroed structure*—a term meaning that the structure is overwritten with ASCII(0) as if it was a string with length equal to the size of the structure. In practical terms, it implies that a scalar descriptor will have its member named sad_desc (of type paramdsc) with its member named dsc_address set to zero.

BY REFERENCE WITH NULL SIGNALING

This mechanism was introduced in Firebird 2 to reap one benefit of the descriptor mechanism without being involved with internal details of the engine and other problems that come from the excess of flexibility allowed by passing parameters by descriptor.

Since REFERENCE (sing.) is not a keyword and introducing new reserved words always comes with the risk of swamping some application developer, the decision was simply to append NULL to the data type to signal that it can be NULL. Therefore, if a UDF requests SQL NULL signaling for a parameter, it should be prepared to receive the null pointer if the engine is handling NULL.

This new mechanism allows the information to be enhanced without changing the behavior of the old BY REFERENCE mechanism, since it would crash almost any UDF written before the change. Do not use neither Delphi references nor C++ references here: they work on the assumption that a valid parameter has been provided, so they would dereference it behind the scenes to be treated like a value. With a null pointer, the code will cause disaster before the first line of code written by the developer inside the UDF: the so-called null pointer exception invariably causes an AV in Windows, segmentation fault in Unices, etc.

For example, when C++ declares a constant reference and the caller uses a constant of another type, the compiler creates a temporary of the correct data type and copies the original input in the temporary. Here, however, the UDF is code separated from the engine; it cannot adapt the engine's code magically to its own needs.

3—Input Parameters

We have seen that input parameters are by default passed as pointers and that they can be received as descriptors, too. Only arrays have to be passed by scalar array, otherwise the user will have to deal with the internal storage format, in a fashion similar to the function *get_scalar_array()* inside fun.epp in the FB source code. Now we will look at how different types are handled.

Scalar types

For C, these are float, double, short, long and int64 and their equivalent in other languages. We can add date and time (long integers). In 32 bit platforms, the engine maps float and double to their native representations, short to SMALLINT, int and long to INT and int64 to BIGINT or NUMERIC(18, 0).

These types are straightforward to deal with. If you are using a pointer, you should dereference it to get the value. If you are using a parameter by reference in Pascal or C++, you're done, unless you requested SQL NULL signaling, because you will need pointers to detect null.

String types

The native string types are CHAR and VARCHAR.

- CHAR is represented by a pointer to char, without a length indicator and without a null terminator. Therefore, handling it in UDFs is error-prone, as the developer has to rely on the UDF declaration to hard-code the maximum length in the code. If the DDL declaration falls out of sync with what the UDF code expects, disaster will happen.
- VARCHAR is represented by a pointer to char without null terminator, but prepended by a length indicator. The exact layout is depicted in ibase.h with the *paramvary* structure: when the UDF requests a VARCHAR parameter, a pointer to *paramvary* is received.

After that, getting the value is simple: take the length from *vary_length* and from the *vary_string* data member, you have "length" bytes to read. This length is expressed in bytes, so it may be confusing for character sets that do not use one byte per character.

> **Do not try to map this type to the Pascal string type, since *paramdsc* uses two bytes for the length, the original Pascal uses one byte and the new Delphi string specification is an implementation detail.**

Both CHAR and VARCHAR specify the maximum size of the field in the UDF declaration. If the size is not provided, one is assumed. The *vary_string* member is of type UCHAR, that maps to unsigned char in C/C++ and to byte in Pascal, where it may be cumbersome to deal with.

- CSTRING

The C string type (a simple array of bytes without length indicator and with a null terminator) has been used to pass string parameters by reference between different systems. It is no surprise it was chosen as an alternative. CSTRING cannot occur outside a UDF declaration: it is not a type accepted to declare domains, table columns or procedure parameters: it is allowed only to declare UDF parameters and/or the result value.

It seems more efficient because, unlike original Pascal strings, C strings are not copied from caller to callee. However, the engine does not use this type internally, so it has to convert a CHAR or VARCHAR to this representation in temporary storage.

CSTRING is thus simple to handle but not very efficient. For Pascal, a PChar declaration does the trick and bytes are read until the null ASCII value (ASCII(0)) is found. The UDF declaration has to specify a length for the parameter. This length, in turn, is the same as *strlen()* would return: it does not count the null terminator.

Other types

- The timestamp type (ISC_TIMESTAMP in ibase.h) is a structure comprising a date part and a time part. When receiving parameters by reference, a pointer to this structure is passed.
- For blobs and arrays, the contents are not provided. For blobs, it is the wrapper structure by default.

  Since arrays are blobs, if a blob field receives an array it will be able to do nothing useful with it, since it will be only an internal handle. For array parameters, the UDF receives the scalar array descriptor, followed by the data that belongs to the full array in all its dimensions.
- There are other declared internal types in Firebird, like *packed*, *byte* and *quad* that do not map to SQL types.

4—Return value

The return value is declared when the first syntax to declare a function is used:

```
RETURNS { <datatype> | CSTRING (int) [CHARACTER SET charset_name]} [{BY VALUE
| BY DESCRIPTOR}] [FREE_IT]
```

In this context, the engine will call the UDF like a regular function:

```
return_value = udf(<argument_list>);
```

- The default is to return the value BY REFERENCE, and recall that it cannot be specified explicitly. Returning by reference allows the UDF to signal SQL NULL by returning a null pointer to the engine.

  This explains why there is no explicit mechanism to return NULL, since it has worked always the same. Input parameters, on the other hand, even when passed by reference, did not signal NULL to the UDF.
- The second mechanism, BY VALUE, is easy to use for scalar types. It should be obvious that there is no way to signal SQL NULL and that it cannot be used for string data types.
- The third mechanism, introduced experimentally in Firebird 1, is the possibility to return a descriptor (*paramdsc*) to provide a small degree of flexibility when handling values of a type different but compatible with the type declared by the UDF.

  For example, a UDF that returns INT by value may actually return the equivalent to SMALLINT (short in C) with no harm, but a UDF that needs to signal NULL will want to return by reference. Since the received pointer will always be treated as a pointer to INT, dereferencing the pointer to another type as if it were a pointer to INT is likely to cause a trashed result in the best case. Returning by descriptor allows you to tell the engine that really a SMALLINT was returned, so it can retrieve the value properly.

If the UDF does not want to build and return a string of a type native to the engine, it can elect to return a CSTRING—the built-in string found in the C language—and the engine will convert it to a native type.

- There is no way to return an array from a UDF.

FREE_IT

To express exactly the combinations of the syntax that are allowed, the DDL syntax would have to be described in more cumbersome ways. Therefore, the previous syntax does not reflect that fact that FREE_IT can only be used when the parameter is returned by reference or by descriptor.

In Windows, at least, it is of paramount importance that both a library and the process that loaded it see the same heap, otherwise the deallocation will be invalid and will cause misfunction.

FREE_IT was invented by Borland as a way to allow the UDF to create the return value dynamically, but obviously it should be deallocated somewhere. Since, by the time deallocation is possible, it is no longer the UDF but the engine's code that is running, FREE_IT tells the engine to free the memory.

A small utility function was created, named *ib_util_malloc()* that maps to C's *malloc()* routine. When the UDF wants to create dynamically the memory that will be returned to the engine, it should invoke ib_util_malloc instead of C++'s *new* operator or Pascal's *GetMem* function and the returned parameter should be specified with the FREE_IT attribute.

> If the parameter is returned by reference, deallocating it with free() inside the engine is straightforward. If the parameter is returned by descriptor, however, we have two elements that were created dynamically: the descriptor and the data. Firebird 1 only freed the data, making it useless for most needs. Firebird 2 deallocates both the descriptor and the data.

FREE_IT has been the subject of much discussion. Different people have stated that it does not work and that the memory leak is inevitable. This is an urban legend. There is no memory leak if the basic principles are followed, coupling *ib_util_malloc* with FREE_IT and assuming the engine has been built properly.

Further, if you plan to use FREE_IT with descriptors in Firebird 2, remember you have to use the same *ib_util_malloc* to allocate the descriptor itself.

5—Output parameter

The output parameter is declared when the second syntax to declare a function is used:

```
RETURNS PARAMETER int
```

In this context, the engine will call the UDF like a function with no return parameter, known in C as function with void return type and in Pascal as a procedure:

```
udf(<argument_list>);
```

In this case, one parameter in the argument list has been designated the output parameter. The position ranges from one to ten. This is the original way to return the result from the UDF and it could have been used even with languages with no concept of dynamic allocation, since the parameter is created by the engine.

It is important to restate that the UDF does not need to do anything to create the output parameter. It gets created by the engine, with the rest of the arguments. If the UDF declared that parameter as a descriptor, the descriptor is passed and its *dsc_address* member will point to the allocated memory in accordance with the UDF declaration.

For example, if the declaration says that the UDF has 5 arguments and the third one is the output parameter by descriptor, being type varchar(20), the third argument will be a pointer to a *paramdsc* and the *dsc_address* member will point in turn to a *paramvary* struct, whose *vary_string* member points to memory addressing sufficient to contain 20 characters—more than 20 bytes if a MBCS charset was specified.

If the declaration says that the last argument is the output parameter and it is of type cstring(50), then the UDF will receive a pointer to a character (the starting address of the C string) with enough size for 50 characters plus the null terminator. After analyzing the output parameter, the engine may deallocate it when necessary, in the same way it treats the input parameters.

FREE_IT v/s PARAMETER n.

Since the original syntax seems to be easier, one has to wonder why Borland needed to invent a modifier to the existing mechanism, known as FREE_IT. The output parameter and the result value have somewhat different usages and they can be combined in the same library. Obviously, they cannot be combined in the same UDF, because you declare either an output parameter or a result value.

PARAMETER frees the application developer from having to remember to use the auxiliary function *ib_util_malloc()* when returning dynamically created memory to the engine.

Secondly, if the declaration of the UDF that allocates and returns dynamic memory does not include the FREE_IT keyword, a memory leak would be implicit with each call to the function.

Thirdly, PARAMETER avoids the risk of some incompatibility in *malloc()* usage because the engine was built with one compiler and the UDF with another.

Why PARAMETER is not ideal

At first sight it seems the ideal method. But there are good reasons why it is not.

For example, it suffers from a problem with the declaration. Suppose the UDF was built on the assumption that the output parameter is cstring(50). This means we have 51 bytes available (the UDF has to put the C null terminator at the end) or 51 characters for MBCS charsets. Since this is an output parameter, at most the data is initialized to zero. There is no way to know the length from inside the function.

The UDF works knowing it has 51 characters available. Later, the declaration is changed to be only CSTRING(30), then we have only 31 characters available to write out result. If the UDF is not amended, we'll have a classic buffer overrun with variable outcome: the engine may continue running trashed, other parameters may be altered or the engine may crash in an unrelated section of code.

Also, if the parameter was specified as VARCHAR(12000), the engine will always make space for 12001 characters in that parameter. This may be a waste of dynamic memory if, in most cases, the UDF will return three characters. (Another question is whether this concern is valid, now that memory modules are relatively cheap).

With FREE_IT, the application developer, the UDF creator, allocates exactly what the result needs and returns it to the engine. In the event the declaration specifies a shorter string than the returned one, when copying the value to its internal format, the engine will realize that there is not enough space and will stop the request with an error message. We have seen that returning a pointer to a smaller scalar value than the declared type by reference (for example, SMALLINT instead of INT64) causes wrong results.

You might say that the solution is to return result by value. This is valid for scalar types, but it does not detect overflow in case the UDF declared SMALLINT by value and the code returns a value that does not fit in a short. Also, you cannot return NULL to the engine.

6—Observations

Now that we have explored the syntax in detail, we need to make some general recommendations, based on what we now understand of the possibilities supported by the engine internally.

- The preferred mechanism for UDF's that return numeric results is by value. Scalar values can be copied directly, unlike strings, where different programming languages have different behavior. Of course, this precludes the possibility of signalling SQL NULL to the engine.
- The new mechanism "reference with null signaling" in Firebird 2, enabled by appending NULL to the declared parameter type, minimizes the need to resort to low level descriptors. People can revamp their UDF's to detect the null pointer and react acccordingly, without making more code changes, then change the UDF declaration through DDL.
- The support for *PARAMETER n* is incomplete. For example, only strings are proven to work with it. In Firebird 1, timestamp was included, but remained beset by an ancient bug (not fixed until Firebird 2) when the output parameter is not the latest.
- When you decide to receive descriptors, be prepared for the worst. The engine may send a null pointer, a paramdsc with the dsc_address set to null or a paramdsc with the null indicator in the dsc_flags to indicate SQL NULL and it will not respect your declared parameters types at all.

  When using references, the engine uses predefined logic to try to match the parameters with the UDF found by name. If there is not a match, an error is produced and the UDF is not called. With descriptors, you could specify int64 for a parameter and the engine will accept it, even if the invocation passes a string in that position. Therefore, you should always test the type of the parameter when it is not null. Only an output parameter is guaranteed to meet the declared type if specified by descriptor.
- Do not make assumptions about the engine's internals, unless you will state clearly that your library was created for a specific server version and architecture. Remember that a classic mistake is to return the address of a local variable when returning results by reference. Since the address becomes invalid once the function returns, disaster will happen. If you need this, allocate the variable dynamically with ib_util_malloc and declare the UDF with FREE_IT.
- Do not assume that ib_util_malloc maps to malloc. This was done only in an internal, built-in UDF but, unless you want to be checking the code of each new server version, stick to the published function, so if it is mapped to something else (for example, the engine uses memory pools), the UDF will work without modifications.
- Never free (deallocate) parameters provided by the engine. It is the engine that determines the lifetime of those parameters, not your UDF.
- Avoid API calls from inside a UDF. You may lock the engine or cause unpredictable results. For example, the *blobcallback* structure offers special methods to callback into the engine to read or write a blob. UDF's that attempt connections are walking on the edge of a skyscraper. Of course, utility routines that have nothing to do with database data may be called, like the API calls to convert between the internal representation

of date/time and the time_t structure in C. See fbudf for an example of those functions.

- If you need to fiddle with some internal data, experiment for yourself by writing internal UDF's that are made part of the engine code; see the related article*, How to Write an Internal UDF Function*.
- Allocating data that will be used only inside the UDF with ib_util_malloc is a futile exercise. This function is meant to share dynamic memory with the engine and leave the engine to deallocate it. Therefore, there is no public deallocation routine. If you get rid of it with C's free(), you are assuming that ib_util_malloc maps to malloc(), that may be no longer the case in some future engine release.

  Therefore, for dynamic memory that you allocate, use and dispose inside your UDF, prefer the native allocation routine, "new" in C++ and "new" or "GetMem" in Pascal, depending on your needs. Just be coherent: C++ new and delete should be paired (remember special delete[] for arrays), C malloc and free should be paired, Pascal GetMem and FreeMem should be paired and Pascal new and dispose should be paired. Both C++'s new/delete/delete[] and Pascal's new/dispose execute additional housekeeping beyond just retrieving from or returning memory to the heap.

7—Problems

Unfortunately, part of people's confusion seems to arise studying example UDF's, since the documentation has been ever lacking in this regard. Incredibly, the ibase.h file, meant to contain the declarations an application developer might need, never included the declaration for the blob wrapper structure (blobcallback since Firebird 1) and did not mention descriptors (paramdsc) even though they were available before Borland took ownership the InterBase server. The only mention of the blob wrapper was done in some official document, but missing the last safe-callback function to seek in blobs.

The rationale behind this oversight was probably because it was useful only for stream type blobs and SQL declarations could not generate this blob type, except indirectly by defining an array column. At the time of Firebird 2, the structure to handle input arrays (*scalar_array_desc*) still did not appear in a public header.

FreeUDFLib

For years, people have looked at FreeUDFLib as the working example to write external functions. However, its author, Greg Deatz, merged different approaches and made it possible to work the code more than one way, selectably, by means of conditional compilation.

FreeUDFLib can be considered a superb work, given the lack of decent documentation. However, Greg paid a lot of attention to functionality, at the cost of the main aspect that helps a newcomer: simplicity. It is almost impossible to separate the helper functions and complex logic from the bare-bones functionality an application developer needs to write a basic UDF.

Further, he decided to use optional thread-local storage (TLS), a technique discouraged in libraries by both Borland and Microsoft. All threads share the same data, but TLS attaches some data to a thread for later recovery (when the thread is scheduled to run, the data can be recovered). Your UDF does not need to keep data between invocations, unless it is a very strange UDF meant to aggregate data, using some obscure technique.

Greg's use of TLS was an attempt to minimize allocations and deallocations and to be able to free memory allocated in the UDF without resorting to FREE_IT. The assumption is that the engine unloads the library only when the server process is terminated, a time when no data processing is happening. Since your UDF will be invoked for each row in a recordset, each invocation may be considered independent and you do not need to keep information between invocations—your external function does not need TLS.

UCHAR

The *dsc_address* member in *paramdsc* is defined as a pointer UCHAR and the *vary_string* member in *paramvary* is defined as an array of one UCHAR element. This is only a stub, because the array can have many more elements, according to the length of the data being handled.

The type UCHAR (unsigned char) is alien to Pascal developers, since Pascal's byte is not considered a character type, but a numeric type. Pascal only knows about Char and its WideChar counterpart, whereas C handles char, unsigned char and signed char plus the wide char alternative.

For the same reason, the place where *paramdsc* demands a signed char has been mapped to Pascal's ShortInt, which provides the same range (-128..128). The conversion of the data structures to Pascal was done considering the ranges of the data types, but this forces Pascal code to do some convolutions. Further, Pascal is has a very strict separation between integers, characters and booleans that makes for longer code than in C++.

Overflow

When you designate one of the parameters as the output parameter, you should resign to the fact that you will have to hard-code the maximum length that you can use without overwriting memory, because the engine does not tell you about that. Even in a *paramvary*, the *vary_length* comes initialized to zero. Therefore, if you want to be independent of the DDL changes, you have two options:

- Request the output parameter by descriptor, then the *dsc_length* data member of *paramdsc* will tell you the length available. You have to reserve two bytes for the length in VARCHAR and one byte for the terminator in CSTRING. Also, to work with the length and data of VARCHAR in a portable way, you have to cast *dsc_address* to *paramvary*.

Remember, varchar by descriptor is the only case when you have two lengths, one in the descriptor and another in the *varying* structure. The length in the *varying* structure is two bytes less than the former. It only makes sense to adjust *dsc_length* for strings. It is internally predefined for the other types (like scalars).

- ❑ Alternatively, avoid output parameters altogether and, instead, allocate the returned value dynamically with ib_util_malloc and use the FREE_IT keyword in the declaration to inform the engine that it should deallocate the returned result once it has read it.

Exception handling

Your favorite resource protection block (C structured exception handling, C++ auto_ptr, Delphi try/finally, etc.) is unreliable if you have a call to an engine function inside the protected block.

1. Any exception that crosses the boundary between the owning process and the DLL is going to be a problem.
2. The engine code knows nothing about your UDF structure. Even though the current engine is writen in C++, your UDF may be written for another compiler, in another language, etc.

   For example, an attempt to call the *put_segment* method on an input blob is an error condition that will be handled by the engine, stopping the UDF execution. The execution never returns to the UDF to perform any kind of cleanup.

   Even though C++ has automatic destructors, experience shows that if an invalid operation is attempted on a blob, the destructor of an object created before the callback is not called. The same problem is likely to occur with the automatic cleanup of the Delphi native string. The best your technique can do is save you when the problem is in your UDF code. When the engine is involved and is the one that finds the exception, resources simply leak.

   Since the most common resource a UDF may need is memory from the heap, a solution may be found in future releases, but for now, there is no improvement here.

"Disaster will happen"

In several places, the expression "disaster will happen" has been used. The engine wraps the UDF call in a protection block. If the UDF produces an AV (Windows) or segmentation fault (UNIX), SIGBUS (Unix), a numeric overflow or a division by zero among other failures, the protection block will catch it and will cause the Firebird server to shut itself down. Others, like stack overflow, will cause the UDF to stop and the server will try to continue running.

At any event, since the UDF is in a library and will be executed in the context of the loader process (in this case, the server), any critical fault produced inside the UDF is seen by the operating system as a fault produced in the server process. This will kill one instance in Classic, but in Superserver, one thread executing a flawed UDF is enough to shut down the whole server. Therefore, you should test your external functions with a local server to avoid stopping everyone else !

8—The phoenix library

The phoenix library is not a set of essential enterprise functions. It is rather a bunch of basic external functions that explain how the different techniques for handling UDF parameters work. Some of those functions may be generally useful whereas others are mere logic demonstrations.

In C++, most of the work is done. We only had to declare the aforementioned *scalar_array_desc* to get the library working.

In contrast, in Pascal, some of the declarations are missing or outdated or inaccurate. In particular, old ports of ibase.h to Pascal may not have what we need. Most Pascal programmers will find the style used awkward or atypical. Having worked with Pascal (and later, Delphi) and C++ for near 15 years, we wanted to avoid bugs in the porting of the library to Pascal. The translation is almost literal and the mapping of data types is as accurate as possible. Smoothing the Pascal code is left as an excercise.

The file ibase_custom.pas

or how to put a phoenix in a henhouse

There should be a way to work with other languages than C/C++ in the engine. With some tricks, maybe we could can make UDFs in COBOL, but let's stick to Pascal for now.

Instead fixing one of the not up-to-date Pascal headers for the phoenix library, we made a new version that contains only what we need. We called it *ibase_custom.pas* but, if you like, you could name it *firebird_custom.pas*...

The interface section

The interface part contains

- The typical Firebird license
- Accurate mappings of Firebird data types to Pascal. They may be harder to use than other mappings, but they represent very well the range and usage of the given data

types. Since Pascal cannot handle things like ^type as a parameter declaration, we define the pointer types here and prepend a P to the type pointed to, so we can use Ptype in our parameter declaration (example: PUChar).

The API types have to remain abstract where possible, isolating the developer from platform differences (for example, SLONG has to be 32-bit signed integer everywhere to be mapped correctly to the INT data type in SQL, also known as INTEGER).

```
type
   SCHAR  = shortint;
   UCHAR  = byte;
   SSHORT = smallint;
   USHORT = Word;
   SLONG  = LongInt;
   ULONG  = LongWord;

   PUchar = ^UCHAR;
```

- ❑ The blobcallback record, together with its get/put/seek methods, again with data types based on the previous mapping. It is important to remember that those methods should use the C calling convention, too:

```
blobcallback = record
   blob_get_segment:     function(hnd: Pointer; buffer: PUchar;
                                  buf_size: USHORT;
                                  var result_len: USHORT): SSHORT; cdecl;
   blob_handle:          Pointer;
   blob_number_segments: SLONG;
   blob_max_segment:     SLONG;
   blob_total_length:    SLONG;
   blob_put_segment:     procedure(hnd: Pointer; buffer: PUchar;
                                   buf_size: USHORT); cdecl;
   blob_lseek:           function(hnd: Pointer; mode: USHORT;
                                  offset: SLONG): SLONG; cdecl;
end;
Pblobcallback = ^blobcallback;
```

- ❑ Some enumerations that may be not usable directly as parameters without a cast (due to restrictions in the mix between enumerated types and integers) but which serve as documentation for the constants they represent:

```
   // This enum applies to parameter "mode" in blob_lseek
   blob_lseek_mode = (blb_seek_relative = 1, blb_seek_from_tail = 2);
   // This enum applies to the value returned by blob_get_segment
   blob_get_result = (
         blb_got_fragment = -1, blb_got_eof = 0, blb_got_full_segment = 1);
```

- The VMS descriptor, known internally as "dsc" and named here *paramdsc*. It contains
  - the data type (documented with constants after the record)
  - the scale (typically negative for numeric and decimal, that are handled as integral types with the scale being the power of ten)
  - the length (with a maximum of 64K-1 that represents the maximum row size, although it is in practice near 100 bytes smaller)
  - the sub_type (represents the blob's subtype and for string types, it is the charset plus the collation, encoded)
  - the flags (described as constants, the most important one being DSC_null)
  - the address, that represents the pointer to the data. The address is of type pointer to unsigned character, so it needs to be cast to other types when needed. It could have been the generic Pointer (the Pascal equivalent to void*) but a good percentage of usage is as pointer to bytes, so the internal declaration was preserved.
  - Since the scale may go between -128 and 127, the only type that matches C's signed char is shortint, aliased SCHAR for the same reason.

```
paramdsc = record
   dsc_dtype:     UCHAR;
   dsc_scale:     SCHAR;
   dsc_length:    USHORT;
   dsc_sub_type:  SSHORT;
   dsc_flags:     USHORT;
   dsc_address:   PUchar;
end;
Pparamdsc = ^paramdsc;

const
// values for dsc_dtype
   dtype_unknown    = 0;
   dtype_text       = 1;
   dtype_cstring    = 2;
   dtype_varying    = 3;
   dtype_packed     = 6;
   dtype_byte       = 7;
   dtype_short      = 8;
   dtype_long       = 9;
   dtype_quad       = 10;
   dtype_real       = 11;
   dtype_double     = 12;
   dtype_d_float    = 13;
   dtype_sql_date   = 14;
   dtype_sql_time   = 15;
   dtype_timestamp  = 16;
   dtype_blob       = 17;
   dtype_array      = 18;
   dtype_int64      = 19;
   DTYPE_TYPE_MA    = 20;

// values for dsc_flags
   DSC_null      : USHORT = 1;
   DSC_no_subtype: USHORT = 2;   // dsc has no sub type specified
   DSC_nullable  : USHORT = 4;   // not stored. instead, is derived from
                                 // metadata primarily to flag SQLDA (in DSQL)
```

- The varying record, known internally as "vary" and named here *paramvary*. Use only for CHARACTER VARYING, also known as VARCHAR. It contains the length, followed by the data. The data is declared as an array instead of a pointer because the data immediately follows the length: they are contiguous.

  Do not activate range checking in Delphi if you plan to traverse the array with an integer variable used as an index: at run-time, there will be as many elements in the array as vary_length dictates and, if the length is zero, there will be an unused position. The array with one position is a trick to indicate that there is an array here.

  Records need to be declared with known sizes. With range checking activated, the only way to bypass the run-time checks is by taking a pointer to the first element and advancing the pointer to read more elements. Typically, Pascal programmers are more used to handling arrays with an index.

```
type
   paramvary = record
      vary_length: USHORT;
      vary_string: array[0..0] of UCHAR;
```

```
    end;
    PParamvary = ^paramvary;
```

- The *sad_repeat* record is defined inside *scalar_array_desc* in C but, in Pascal, this is not possible because nested records are taken as elements, not as definitions. We need to use *sad_repeat* variables, so they have to be defined beforehand. It contains simply the lower and upper range of a dimension in an array. Not surprisngly, the total number of elements in a dimension is the difference of the two values plus one.

```
// This structure is not defined in ibase.h.
// Cannot do nested typedefs in Delphi.
// Therefore, sad_repeat is defined ahead.
   sad_repeat = record
      sad_lower: SLONG;
      sad_upper: SLONG;
   end;
```

- The *scalar_array_desc* record, not present in ibase.h even in the native version for C. It starts with a descriptor. Only one descriptor is needed because all elements have the same type. See *paramdsc*, described previously, for more information. Follows the number of dimensions and the array of *sad_rpt* records, specified with one element at compile time.

```
   scalar_array_desc = record
      sad_desc: paramdsc;
      sad_dimensions: SLONG;
      sad_rpt: array[0..0] of sad_repeat;
   end;
   Pscalar_array_desc = ^scalar_array_desc;
```

- The mapping of the declaration of *ib_util_malloc*, found in ib_utils.h, that returns a generic pointer.

  With Pascal's linker, there will not necessarily be clash if another file declares it. Here we elected to have a second function return a pointer to char, since it is the most used case. We are importing the function twice with different names. Again, remember, it should follow the C calling convention.

```
function ib_util_malloc(size: Integer): Pointer; cdecl;
   external 'ib_util.dll';


function ib_util_malloc2(size: Integer): PChar; cdecl;
   external 'ib_util.dll' name 'ib_util_malloc';
```

You might wonder about the return type. Well, in a 16-bit world, the classic Borland C++ compiler, predecessor of BCB, allowed you to select different memory models with different pointer sizes. But in a flat 32-bit world, all native pointers have the same size, so changing from an untyped pointer to a typed pointer is only a convenience. No harm will happen. (Microsoft provides size optimizations for pointers to a member of a class in C++, but they are different from pointers to objects).

* * * * * * *

We do not call the external API, so we do not need any more declarations. Arrays are defined starting from zero to ease to port from C++. Our implementation is empty because we are only handling declarations. We did not get into objects, so there are no class methods to write here. Also, we avoid global variables as recommended, so there is nothing to put in the initialization section.

The project file

Our Delphi project file is straightforward: it includes the *library* keyword to mark the code as a DLL instead of an EXE, the usage clause (including our two units), the exports clause to make the entry points visible. In the executable part of the project file we simply set the IsMultiThread global variable to true, to tell Delphi we need a thread-safe allocator.

```
library phoenix;
uses
   main in 'main.pas',
   ibase_custom in 'ibase_custom.pas';

{$R *.res}

exports
   p_sumchar1,
   p_sumchar2,
   p_sumchar3,
   p_lastchar1,
   p_lastchar2,
   p_lastchar3,
   p_reverse1,
   p_reverse2,
   p_reverse3,
   p_defragment_blob,
   p_generate_blob,
   p_sample_blob,
   p_intersperse,
   p_array2text;
```

```
begin
   IsMultiThread := true
end.
```

The C++ and Pascal units

We are going to compare the code written in C++ and in Delphi. We'll examine first the declarations and then we will start looking at each function.

In the C++ header for the main file, we can find after the license:

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
#define PHOENIX_API __declspec(dllexport)
#else
#define PHOENIX_API
#endif
```

This is done because, in Windows, functions must be explicitly exported to be visible and usable from other libraries or programs. Since this code may be compiled in other platforms where all functions in a library are exported by default, the __declspec keyword is protected by a preprocessor macro, PHOENIX_API that needs to be defined only in Windows.

In contrast, we did not do that for the Delphi version, since it was attempted only on Windows. Whether Kylix may be able to compile and generate a useful library is territory for people with technical curiosity.

After that, we find the native declaration of the scalar array descriptor, the same one that is found inside the engine. Why did not we make a separate unit like in Delphi? Because this is the only declaration we are missing and all the rest is taken from the ibase.h header of Firebird 2. We can observe the nested declaration and usage of *sad_repeat* in one step:

```
// This structure is not defined in ibase.h.
struct scalar_array_desc
{
   paramdsc sad_desc;
   SLONG sad_dimensions;
   struct sad_repeat
   {
      SLONG sad_lower;
      SLONG sad_upper;
   } sad_rpt[1];
};
```

We'll finish with the declaration of the functions. The PHOENIX_API as described works as the exporting clause for Win32 and is mapped to nothing in other platforms.

C++ declarations

The C++ compiler automatically defines __cplusplus for a C++ project. Hence, if it is not defined, it may be that this header is being imported by a C program that does not understand the extern "C" directive, used by C++ as the equivalent of Pascal's cdecl, namely, to use C calling convention and linkage.

```
#ifdef __cplusplus
extern "C"
{
#endif


   PHOENIX_API int p_sumchar1(const char* s);
   PHOENIX_API int p_sumchar2(const paramvary* v);
   PHOENIX_API int p_sumchar3(const char* s);
   PHOENIX_API char* p_lastchar1(const char* s);
   PHOENIX_API char* p_lastchar2(const paramvary* v);
   PHOENIX_API char* p_lastchar3(const char* s);
   PHOENIX_API void p_reverse1(const char* const s, char* dest);
   PHOENIX_API void p_reverse2(const char* const s, paramvary* dest);
   PHOENIX_API void p_reverse3(const char* s, char* dest);
   PHOENIX_API void p_defragment_blob(const blobcallback* input,
                                      blobcallback* output,
                                      const SLONG& input_seg_size);
   PHOENIX_API void p_generate_blob(blobcallback* output,
                                    const SLONG& input_start_size,
                                    const SLONG& input_num_seg);
   PHOENIX_API paramvary* p_sample_blob(const blobcallback* input,
                                        const SLONG& input_len);

   PHOENIX_API paramdsc* p_intersperse(const paramdsc* p1,
                                       const paramdsc* p2);
   PHOENIX_API void p_array2text(const scalar_array_desc* const input,
                                 paramdsc* const pout);
#ifdef __cplusplus
}
#endif
```

You can see that the structures are received by pointer because it is straightforward in C++. We used const to mark input parameters as immutable from our perspective. This will prevent us from making a mistake.

However, the SLONG parameters were received as references. This is the equivalent of "var" in Pascal in a parameter, but const can be applied to it to indicate this is a variable received by reference, that we do not want to be able to change. In Pascal, this is not possible because const is equivalent to a C++ reference. However, we avoided using const in Pascal, since the behaviour of const may be to pass by value or by reference, depending on the compiler's rules.

Explicit passing behaviour

Here, we want to determine explicitly whether we are receiving pointers or references: one mismatch in the engine's interpretation of the DDL declarations for each function's parameter and we'll get garbage. Our rules are:

- strings are always passed by pointer
- scalars may be received by reference if we did not ask for NULL signaling
- scalars must be received by pointer if we did ask for NULL signaling
- blobs and arrays are received by pointer (although they could be received by reference because the engine never sends the null pointer for them)
- output parameters can be received by reference because the engine always provides storage for them
- descriptors are received by pointer just in case the engine sends a null pointer because it found the literal SQL NULL.

Receiving structures/records by pointer is not a problem. Both C++ and Pascal handle them well.

The problem is handling scalars by pointers. InC/C++, at least, a pointer is inter-operable with other integral data types. Hence, if the programmer fails to dereference it, the value of the pointer itself—a memory location—would be used instead of the pointer's contents—the memory pointed to by the pointer—and the compiler will not chime in.

We found that Delphi does not complain either, when the clause ^variable is used instead of variable^ to dereference a pointer. Apparently, the former way is a dumb clause without effect and we only got a warning because the compiler said the variable was not being used. Therefore, for scalars we prefer references, unless we have to detect NULL, which forces us to use pointers.

Pascal declarations

Since in Pascal, both the header and the body (the cpp) reside in the same file, we post here the first part of main.pas for comparison. This is what you will find after the license. We are including our custom version of ibase.h here:

```
unit main;


interface


uses ibase_custom;


function p_sumchar1(s: PChar): Integer; cdecl;
function p_sumchar2(var v: paramvary): Integer; cdecl;
function p_sumchar3(s: PChar): Integer; cdecl;
function p_lastchar1(s: PChar): PChar; cdecl;
function p_lastchar2(v: Pparamvary): PChar; cdecl;
function p_lastchar3(s: PChar): PChar; cdecl;
procedure p_reverse1(s, dest: PChar); cdecl;
procedure p_reverse2(s: PChar; dest: Pparamvary); cdecl;
procedure p_reverse3(s, dest: PChar); cdecl;
procedure p_defragment_blob(input, output: Pblobcallback;
                    var input_seg_size: SLONG); cdecl;
procedure p_generate_blob(output: Pblobcallback; var input_start_size: SLONG;
                  var input_num_seg: SLONG); cdecl;
function p_sample_blob(input: Pblobcallback; var input_len: SLONG):
Pparamvary; cdecl;
function p_intersperse(p1: Pparamdsc; p2: Pparamdsc): Pparamdsc; cdecl;
procedure p_array2text(input: Pscalar_array_desc; pout: Pparamdsc); cdecl;
```

Given the preceding comments, it is not surprising that our first observation is that C++ functions that return something are mapped to Pascal functions and C++ functions whose return type is void (nothing returned) are mapped to Pascal procedures.

Secondly, we have to use custom typedefs to overcome an ancient Pascal limitation. We cannot write

```
function p_sumchar1(s: ^Char): Integer; cdecl;
```

because the compiler does not accept a declaration of a pointer to type in an argument. This is the reason why ibase_custom.pas did declare those needed types and we use them now:

```
function p_sumchar1(s: PChar): Integer; cdecl;
```

Third, no function should forget to include the cdecl clause after its declaration or Delphi will use its own calling convention, generally known as fastcall, that tries to maximize CPU register usage to pass parameters. We have to tell Delphi to use the C calling convention. Doing otherwise will result in garbage being received from the engine or returned to it. There is the dangerous possibility of trashing the stack, too.

C++

```
#include "ibase.h"
#include "ib_util.h"
#include "main.h"

namespace
{
   const int CHAR_LEN = 30; // This constant must match
                            // the DDL declaration CHAR(30).
   const int MAXSEG = 65535; // 64K - 1, the limit for a blob segment
   const int MAXROW = MAXSEG - 100; // Approx the max row size
   const int VARCHAR_PREFIX = sizeof(paramvary().vary_length);
   char* ib_util_malloc2(const long size)
   {
       return static_cast<char*>(ib_util_malloc(size));
   }
   const int SLONG2TEXT = 11; // -2147483648
}
```

Pascal

```
implementation

{$IFDEF GUI_MSG}
uses Dialogs, SysUtils;
{$ENDIF}

const
   CHAR_LEN: Integer = 30; // This constant must match the DDL declaration
                           // CHAR(30).
   MAXSEG = 65535; // 64K - 1, the limit for a blob segment
```

```
    MAXROW: Integer = MAXSEG - 100; // Approx the max row size
    pvary_size: Pparamvary = nil;
    VARCHAR_PREFIX: Integer = sizeof(pvary_size^.vary_length);
    SLONG2TEXT = 11; // -2147483648
```

In C++, we wrapped the private declarations in the unnamed namespace. By definition, the unnamed (or anonymous) namespace is only visible to the translation unit where it appears.

The same effect is achieved inDelphi doing those declarations in the implementation section. As stated previously, ib_util_malloc2 is only a convenience to avoid converting the result from malloc into a pointer to character. In C++ all constants were defined as typed constants. In Pascal, we cannot do that: only untyped constants are true constants, that can be used in array definitions, for example.

The only other notable difference is C++'s

```
  const int VARCHAR_PREFIX = sizeof(paramvary().vary_length);
```

versus Pascal's

```
  const
    pvary_size: Pparamvary = nil;
    VARCHAR_PREFIX: Integer = sizeof(pvary_size^.vary_length);
```

In C++, a class can be "instantiated" on the fly; that is, you can invoke the constructor without assigning the result to a variable, creating an unnamed object whose lifetime spans only the statement where it appears. You can see that *paramvary* does not have a constructor, but the compiler provides one automatically.

We could not find an equivalent technique in Delphi, so we created a pointer of the desired type and got the size of vary_length. It can be argued that dereferencing a null pointer is an error, but here we only request the size. The same trick was used in plain C for different tasks.

There is an $IFDEF for some extra units in Pascal. The only reason is that Delphi was unable to debug the DLL it created using fbserver.exe as the host program. Probably it has to do with the fact that we have the system-wide debugger registered to be MSVC and when Delphi installs, the registry entry is overwritten. Delphi hangs while trying to debug sopmthing it does not understand and, since Windows calls the system-wide debugger when any program crashes, it was a big problem. To avoid it, we changed back to MSVC.

Since we could not debug the Pascal DLL in an interactive way, we had to put messages in a dialog box to see what was happening, but only if the symbol GUI_MSG is defined.

The value CHAR_LEN is important: in several cases, the function cannot determine the amount of space available to write a result in an output parameter. The burden is on the programmer: the declaration of the output parameter must match CHAR_LEN or strange effects will happen. If the declared length is bigger, garbage may appear at the tail; if the declared length is smaller, we would overwrite memory we must not touch and we have a classical buffer overrun problem. The effects are not obvious in most cases.

Other points to note

1. To make a clear difference between identifiers and literal strings in the declarations, our script starts with the declaration

```
set sql dialect 3;
```

although it is not very critical in our case.

2. Our library was designed to take ASCII. External functions can take other character sets, as explained in the syntax for declaring UDFs, but the code would have to rely on the operating system or third party libraries to work with them.

Functions that return scalars by value

Those functions do not have an interest in detecting NULL. They will get an empty string instead. They do not return NULL, so they return their integer result by value (if they had to return strings, they are not scalars but arrays, thus they have to be returned by reference or descriptor).

```
declare external function p_sumchar1
char(30)
returns int by value
entry_point 'p_sumchar1' module_name 'phoenix';
```

Here the 30 is important: it matches CHAR_LEN as explained previously. With a CHAR type, it is impossible to know its length dynamically. We only receive a pointer to the first character. Since we return by value, there is not much harm if the native integer type in the library matches exactly the integer type in the engine (SLONG), so we did not bother to use those special, abstract types defined in ibase.h.

```
// This is for input = CHAR type
int p_sumchar1(const char* s)
{
   int loop = CHAR_LEN - 1;
   while (loop >= 0) // let's ignore trailing blanks
   {
       if (s[loop] == ' ')
          --loop;
       else
          break;
   }
```

```
    int rc = 0;
    while (loop >= 0)
        rc += s[loop--];

    return rc;
}
```

First, we go to the last position. We start counting at zero, so the last position is 29. We go back discarding ASCII blanks until we find something "interesting". At this point, we continue going back, adding to "rc" the numeric value of the character at position "loop". Typical of C/C++ code, we handle the post decrement of the variable in the same statement. The logic guarantees that if the string is purely blanks, we get zero as the result instead of random garbage. However, embedded blanks, those that are before non-blank characters, are counted. For example, in

```
p_sumchar1('Ann Harrison   ')
```

we count only one blank (value 32), but in

```
p_sumchar1('Firebird    ')
```

no blank is included in the count. For the same reason, the result of

```
p_sumchar1('        ')
```

is zero. All the three variations of sumchar behave the same with respect to blanks.

Note to the bored reader

My boss asked me to explain the differences between the C++ version and the Pascal version, so I need to explain once and for all the effect of pre and post increments and decrements to Pascal programmers: ++var means increment the variable, then use it whereas var++ means use the variable, then increment the variable. Same goes for the decrement. The line above

```
--loop;
```

is not much different to

```
loop--;
```

since it is the only instruction. It cannot have side effects and for the same reason we consider the former clearer. But in the second case,

```
rc += s[loop--];
```

the order is important. We need to use the place s[loop] before the variable is decremented or we will skip one character and will read one before the beginning of the string. The statement can be written as

```
rc += s[loop];
```

```
--loop;
```

but almost nobody does that in C++. If you do, remember to put curly braces around the body of the "while" or you will have only one statement in the cycle (a typical oversight anybody, no matter how experienced, can make).

Unlike some experimental languages that are developed by open source enthusiasts, neither Pascal nor C++ take indentation into account to decide which statements belong to a clause (if, while, for, etc.). Indentation is just a visual aid to the programmer and the compiler cannot warn him/her about mismatches between indentation and curly braces.

Now the Pascal counterpart:

```
// This is for input = CHAR type
function p_sumchar1(s: PChar): Integer;
var
   loop, rc: Integer;
begin
   loop := CHAR_LEN - 1;
   while loop >= 0 do // let's ignore trailing blanks
   begin
      if s[loop] = ' ' then
         Dec(loop)
      else
         break;
   end;
   rc := 0;
   while loop >= 0 do
   begin
      Inc(rc, Ord(s[loop]));
      Dec(loop)
   end;
   Result := rc
end;
```

It's almost a copy of the previous function, with some differences. Instead of the native string, we have to receive the equivalent PChar; and we have to increment the counter *rc* by taking the ordinal position of the character to get the ASCII value, then decrement *loop* separately afterwards, to match the C++ version. We use Inc to mimic the += operator. We assign the final value of the counter *rc* to *Result* to return a value.

```
declare external function p_sumchar2
```

```
varchar(50)
returns int by value
entry_point 'p_sumchar2' module_name 'phoenix';
```

Here we do not care about our agreed length (30) because varchars contain their own length indicator. The length can be effectively discovered at run time. Also, we do not waste time ignoring trailing blanks because the VARCHAR type trims them by default.

```
// This is for input = VARCHAR type
int p_sumchar2(const paramvary* v)
{
   int rc = 0;
   for (int loop = (int) v->vary_length - 1; loop >= 0; --loop)
       rc += v->vary_string[loop];

   return rc;
}
```

Attention is called to a detail that may cause much grief. The member *vary_length* is defined as an unsigned 16-bit quantity. If we have a loop that needs to count from N back to 1, this would not be a problem. We would stop when the value reaches zero. However, we have to go from N-1 back to 0, so we stop when the value is -1.

The effect of the old C cast

```
(int) v->vary_length - 1
```

is the same as the original C++ cast

```
int(v->vary_length) - 1
```

or the ultra explicit new C++ cast

```
static_cast<int>(v->vary_length) - 1
```

in the sense that it affects only the first operand, not the whole subtraction. If we do not do the cast and we receive a string of length zero, since vary_length is unsigned, subtracting one from it would produce another unsigned that wraps, probably to the biggest number in the range (64K), causing us to walk totally illegal memory places, counting garbage.

```
declare external function p_sumchar3
cstring(50)
returns int by value
entry_point 'p_sumchar3' module_name 'phoenix';
```

Now the Pascal version:

```
// This is for input = VARCHAR type
function p_sumchar2(var v: paramvary): Integer;
var
   loop, rc: Integer;
begin
   rc := 0;
   for loop := Integer(v.vary_length) - 1 downto 0 do
      Inc(rc, v.vary_string[loop]);
   Result := rc
end;
```

We should remember the engine always passes parameters by pointer and this means by reference in Pascal. For the parameter, instead, we could have done

```
var p: PParamvary
```

using the declaration in *ibase_custom.pas*. Our reverse loop seems simpler in Pascal than in C++ thanks to the *downto* clause. We again take the ordinal position of the final character to get the numerical value that is assigned to the result of the function.

Again, we do not worry about the length, since this is a C-style string, with the ASCII NULL (binary zero) terminating the string. However, as explained earlier, the engine must waste time and space converting from the internal representation to a temporary of type CSTRING.

Since this is a type that does not exist in storage in the engine, we did not bother to trim trailing blanks. Implicitly, we'll count them if they come from a CHAR and we'll ignore them if they come from a VARCHAR.

For example, since literals are of CHAR type,

```
p_sumchar3(' ')
```

is 32. We could have done better by going to the end of the string, then start backwards discarding trailing blanks until we find something "interesting" as p_sumchar1 did, but it is left as an exercise.

```
// This is for input = CSTRING type
int p_sumchar3(const char* s)
{
   int rc = 0;
   for (; *s; ++s)
       rc += *s;

   return rc;
}
```

We need to iterate until the character we test is the binary zero. This is the end of the string. We could have used the standard strlen() but we wanted to use only custom code as much as possible to be able to debug everything.

The Pascal version again uses PChar:

```
// This is for input = CSTRING type
function p_sumchar3(s: PChar): Integer;
var
   rc: Integer;
begin
   rc := 0;
   while s^ <> #0 do
   begin
      Inc(rc, Ord(s^));
      Inc(s);
   end;
   Result := rc;
end;
```

It uses an atypical construction to be able to work with ASCII-null terminated strings: it compares each position against the literal character ASCII-zero (unprintable, first character in the ASCII table, denoted with #0) and enters the loop as long as this character is not encountered.

NOTE

In all these three functions, the variable *rc* can be eliminated in favour of manipulating Object Pascal's special *Result* variable directly. To preserve a visual equivalence between the C++ and Pascal versions of the examples, it was not done here.

Functions that want to take and give NULLs

Those functions take advantage of the new mechanism in Firebird 2 to receive parameters by reference, but getting the null pointer when the input parameter is SQL NULL. Therefore, given the result they return (the last character in the input string) they need to return a string.

Unlike programming languages where there is a distinction between a single character (scalar) and a string, here anything is a string. Even CHAR(1) or VARCHAR(1) have to be returned by reference. Since we'll be allocating the result dynamically, we need to use the keyword FREE_IT to inform the engine it should dispose of the result when it is done with it.

Since we always return one character, it can be argued that we can use an output parameter of known size, but then we lose the ability to signal null to the engine unless we resort to descriptors. All three versions thus share the common behaviour of not allocating anything if they detect a null input pointer, and will return the null pointer immediately to the engine.

Notice we allocate a single character for the result: we stated the result is plain CHAR(1), so we cannot include terminator as we would with CSTRING.

```
declare external function p_lastchar1
char(30) null
returns char free_it
entry_point 'p_lastchar1' module_name 'phoenix';
```

Again, this is the CHAR type, so we need to rely on our hard-coded 30 to find the last character.

Because of the way the CHAR type works, in most cases we will return the space character, unless there are no trailing blanks in the input string. If we had trimmed them, we would not return the actual last character.

```
// This is for input = CHAR NULL type and output CHAR with FREE_IT
char* p_lastchar1(const char* s)
{
   if (!s)
       return 0;

   char* const dest = ib_util_malloc2(1);
   *dest = s[CHAR_LEN - 1];
   return dest;
}
```

Our Pascal version makes the distinction between pointers and numeric types more clearly:

```
// This is for input = CHAR NULL type and output CHAR with FREE_IT
function p_lastchar1(s: PChar): PChar;
begin
   if s = nil then
   begin
      Result := nil;
      Exit;
```

```
    end;

    Result := ib_util_malloc2(1);
    Result^ := s[CHAR_LEN - 1];
end;
```

Even though we declare that we will return one character, we need to return a C string. Since we receive and return pointers, we can test against our familiar "nil" and return it, too. Of course, to achieve the same effect as a *return* with a value in C++, in Pascal we need to set the Result variable and call Exit to return immediately to the caller.

If we want to allocate memory that has to be freed by the engine, it has to be allocated with the utility function ib_util_malloc, regardless of whether we use C++ or Pascal, or some other language. Recall that we defined ib_util_malloc2 as a useful synonym that returns a pointer to char, without putting a cast here.

```
declare external function p_lastchar2
varchar(50)
returns char free_it
entry_point 'p_lastchar2' module_name 'phoenix';
```

In the second version we have an anomaly: we return immediately not only if the input is the null pointer, but also when the length is zero. Unlike CHAR, a VARCHAR can have zero length. In this case, the last character does not exist, so we return NULL. A hard-coded limit is unnecessary, since we get the length from the input structure *paramvary*.

```
// This is for input = VARCHAR NULL type and output CHAR with FREE_IT
char* p_lastchar2(const paramvary* v)
{
    if (!v || !v->vary_length)
        return 0;

    char* const dest = ib_util_malloc2(1);
    *dest = v->vary_string[v->vary_length - 1];
    return dest;
}
```

Our Pascal version is:

```
// This is for input = VARCHAR NULL type and output CHAR with FREE_IT
function p_lastchar2(v: Pparamvary): PChar;
begin
```

```
    if (v = nil) or (v^.vary_length = 0) then
    begin
      Result := nil;
      Exit;
    end;

    Result := ib_util_malloc2(1);
    Result^ := Chr(v^.vary_string[v^.vary_length - 1]);
  end;
```

Here, *Pparamvary* is used, a choice suggested earlier. We test for the null pointer or the VARCHAR having length zero to return immediately. Otherwise, a more complex sentence than its C++ counterpart is necessary. Since *vary_string* is of type unsigned character, the compiler would take it as a numeric. Hence, we use Chr() to convert it to the Char type. That conversion in C++ happens automatically (but only for one element, not for pointers of UCHAR v/s pointer to char).

Of course, our if() condition is slightly different in Pascal. Whereas C++ demands parentheses around the whole if(), Pascal needs them only to distinguish logical conditions and avoid having the compiler take the OR as binary OR (bitwise), rather than logical OR.

```
declare external function p_lastchar3
cstring(50)
returns char free_it
entry_point 'p_lastchar3' module_name 'phoenix';
```

Again, we test for the null string and, immediately afterwards, test whether the first character is the C null terminator. If so, the string is empty and we return NULL. We advance through the string until we get to the null terminator. By then, we are past the end of the string, so we need to go back one to the last meaningful character. The null terminator is an artifact that does not have a length indicator.

```
// This is for input = CSTRING NULL type and output CHAR with FREE_IT
char* p_lastchar3(const char* s)
{
   if (!s || !*s)
       return 0;

   while (*s)
       ++s;
```

```
    char* const dest = ib_util_malloc2(1);
    *dest = *--s;
    return dest;
 }
```

For Pascal programmers, we need to explain the line

```
 *dest = *--s;
```

This is a prefix decrement, meaning

```
 --s; // go back one position with the pointer
 *dest = *s; // assign the last character to the result.
```

Finally, it is worth explaining why we used the strange form

```
 char* const dest = ib_util_malloc2(1);
```

Since C++ allows that expression, we use it. We want to avoid, especially in more complex cases, accidentally changing the value of the pointer "dest", the address of the memory that was allocated dynamically. We need to pass it to the engine, or it will read garbage. Worse, a run-time failure may happen when trying to deallocate it, if it is not the same pointer we got from ib_util_malloc. The statement does not prevent changing the memory pointed to by the pointer; it only prevents the pointer from being directed to another memory location by mistake.

Now, the Pascal version:

```
 // This is for input = CSTRING NULL type and output CHAR with FREE_IT
 function p_lastchar3(s: PChar): PChar;
 begin
    if (s = nil) or (s^ = #0) then
    begin
       Result := nil;
       Exit;
    end;

    while s^ <> #0 do
       Inc(s);

    Result := ib_util_malloc2(1);
    Dec(s);
    Result^ := s^;
 end;
```

We return nil immediately if the input is nil. Otherwise, we advance the pointer, looking for the null terminator denoted by #0. We allocate one byte with the engine's utility function, then decrement the source pointer (since we are interested in the last useful character, not the unprintable termintor) and assign the result.

Notice lastchar2 and lastchar3, that can test the length or the terminator, returning NULL if the length is zero. This may seem strange, because zero length is not NULL. It is done because, with a zero length, the last printable character is non-existent. Returning #0 would produce failure in printing routines that use it as terminator, truncating output if that output were concatenated with another string. Since there is no valid character with length zero, returning unknown in the form of NULL seems better.

Notice also that the three functions of the lastchar family assign the dynamic allocation to Result, then assign the wanted character to Result^ instead of Result. We are assigning inside the one-byte string that was allocated. Result[0] could have been used if the compiler accepted it. Remember, we are dealing with null-terminated strings that start at position zero.

Functions that work with CSTRING as the input parameter and use native types as the output parameter

They do not receive NULL and cannot signal NULL.

For those functions, we wanted to demonstrate how to use output parameters, so we left the complexity of null input out of the examples. We'll get SQL NULL as an empty string. Even if we got the null pointer, we cannot signal NULL to the engine as a result since output parameters by reference cannot do that.

It is worth remembering that we cannot know from the engine the size of the output parameter and thus have to make sure our DDL declaration for the external function matches the size we use in the function's logic. For simplicity, we chose to receive the input argument as cstring.

Also, the first parameter is a fixed pointer. Because it will be used for comparisons and length calculation, it cannot be moved to avoid programming errors.

First version

In this version, the input is a CSTRING of up to 30 bytes and the output is a varchar(30). We declare it thus:

```
declare external function p_reverse1
cstring(30),
char(30)
returns parameter 2
entry_point 'p_reverse1' module_name 'phoenix';
```

First, we scan the input until we get to the null terminator. We go back one position to the last character. Then we copy it in reverse order. Since we are going backwards in the input and forward in the output, the reversing happens naturally. We initialize "n" to our maximum length to ensure we do not write outside our allowed output space.

There is a catch with CHAR types: they have to padded with blanks. Therefore, if the input was shorter than the output, we do a final loop to fill the output with trailing blanks.

```
// This is for output = CHAR type
void p_reverse1(const char* const s, char* dest)
{
   const char* p = s;
   while (*p)
       ++p;

   --p;
   int n = CHAR_LEN;
   while (p >= s && n)
   {
       *dest++ = *p--;
       --n;
   }
   while (n--)
       *dest++ = ' '; // fill with blanks for CHAR(CHAR_LEN)
}
```

The line

```
*dest++ = *p--;
```

presents two postfix operators, so it has to be decomposed intro three sentences. First, do the assignment of the contents pointed by the pointers (not the addresses of the pointers) and then (in any order) increment the destination and decrement the source. The final loop

```
   while (n--)
       *dest++ = ' ';
```

can be decomposed into

```
   while (n)
   {
       --n;
       *dest = ' ';
```

```
        ++dest;
    }
```

but do not forget the braces to enclose multiple statements. In precise terms, the equivalence is not exact, because the test for n-- involves a test and a decrement always (even if the condition was evaluated as false). The lone --n after the loop is needed to have exactly the same effect, although the compiler emits a warning given that this last decrement has no effect.

The Pascal version:

```
// This is for output = CHAR type
procedure p_reverse1(s, dest: PChar);
var
   p: PChar;
   n: Integer;
begin
   p := s;
   while p^ <> #0 do
      Inc(p);

   Dec(p);
   n := CHAR_LEN;
   while ((p >= s) and (n > 0)) do
   begin
      dest^ := p^;
      Inc(dest);
      Dec(p);
      Dec(n);
   end;
   while n > 0 do
   begin
      Dec(n);
      dest^ := ' '; // fill with blanks for CHAR(CHAR_LEN)
      Inc(dest);
   end;
end;
```

It is noticeably longer than the C++ version because it has to do the pointer increment and decrement separately and preserve the order, as explained previously (a postfix decrement should be done after the expression where it appeared in the C++ version, and so on).

The logic is the same, however. We find the null terminator in the source, go back one position to skip it and, while our pointer is inside the valid range *(p>=s)*, we do the copy, incrementing the destination and descrementing the source.

But we should also check that the output, fixed at CHAR_LEN, does not overflow. Remember, this number must match the DDL declaration for the functions.

Finally, if there is spare room in the target, we fill it with blanks according to the SQL CHAR type specification. We do not bother to append a terminator because it is not used by CHAR or VARCHAR.

Second version

```
declare external function p_reverse2
cstring(30),
varchar(30)
returns parameter 2
entry_point 'p_reverse2' module_name 'phoenix';
```

Our second version with the VARCHAR output starts the same. We have to find the C terminator.

Here we take a difference between the terminator position and the beginning to determine the length, then decrement the pointer to be at the last character's position. If the length is bigger than our predefined output length, we truncate the output. Since this is a VARCHAR, we need to fill *vary_length* with our output length and then copy the reversed input into *vary_string*. We'll use "n" as a sentry to avoid writing more than the allowed number of bytes in the output parameter.

```
// This is for output = VARCHAR type
void p_reverse2(const char* const s, paramvary* dest)
{
   const char* p = s;
   while (*p)
       ++p;

   int n = p - s; // ptrdiff_t
   --p;
   if (n > CHAR_LEN)
       n = CHAR_LEN;
```

```
    dest->vary_length = (USHORT) n; // write the length for VARCHAR
    for (UCHAR* to = dest->vary_string; p >= s && n; --n)
        *to++ = *p--;
}
```

In Pascal:

```
// This is for output = VARCHAR type
procedure p_reverse2(s: PChar; dest: Pparamvary);
var
   p: PChar;
   n: Integer;
   tostr: PUChar;
begin
   p := s;
   while p^ <> #0 do
      Inc(p);

   n := p - s; // ptrdiff_t
   Dec(p);
   if n > CHAR_LEN then
      n := CHAR_LEN;

   dest^.vary_length := USHORT(n); // write the length for VARCHAR
   tostr := @dest^.vary_string[0];
   while (p >= s) and (n > 0) do
   begin
      tostr^ := Ord(p^);
      Inc(tostr);
      Dec(p);
      Dec(n);
   end;
end;
```

We first find the end of the C string and calculate the length as the subtraction of pointers. The comment *ptrdiff_t* is just a leftover from the C++ version to indicate that, in C++, that special type is the result of pointer subtraction, being typically a native integer in the platform. We go back one character to avoid copying the #0 terminator itself, adjust the length so it doesn't overflow our output buffer, then we assign the length.

To avoid warnings, we cast to USHORT, the type of *vary_length*. There's no reasonable way *n* can be bigger than 2^31-1, other than in some future, enhanced version of Firebird.

We need an auxiliary, *tostr*. It has a different name than in the C++ version just because **to** is reserved in Pascal. Also, we need to initialize it with the address of *vary_string* to start copying bytes.

In the loop, we check that we are inside the range of the source string (p>=s) and that we still have bytes to copy (n>0). We apply *Ord()* to the source because, being of type PChar, its elements are Char. The target is UCHAR: hence, the pointer was defined PUChar, using the helper definitions from our ibase_custom.pas.

This loop is the same as the C++ version but is longer because the increments and decrements are done separately, respecting the order. There are other ways to do the loop, assuming the compiler allows them:

```
dest^.vary_length := USHORT(n); // write the length for VARCHAR
tostr := 0;
while (p >= s) and (n > 0) do
begin
   dest^.vary_string[tostr] := Ord(p^);
   Inc(tostr);
   Dec(p);
   Dec(n);
end;
```

Here, *tostr* is an integer, used as an index while treating *vary_string* as an array, a technique familiar to Pascal programmers. After that, we can change the condition *(n>0)* by *(tostr<n)* and eliminate the *Dec(n)* expression from the body of the loop.

Third version

For our third version, we have CSTRING for both input and output.

```
declare external function p_reverse3
cstring(30),
cstring(30)
returns parameter 2
entry_point 'p_reverse3' module_name 'phoenix';
```

As before, we find the terminator, then go back one position to the last character. Then we simply copy the reversed input into the output, using "n" as a sentry marking the maximum output length. Finally, do not forget or you will cause garbage to be read in the engine: we told the engine our output is cstring(30) and hence it allocates 31 characters to allow us to append the null terminator. This is our last sentence after the loop finishes.

```
// This is for output = CSTRING type
void p_reverse3(const char* s, char* dest)
{
   const char* p = s;
   while (*p)
       ++p;

   --p;
   for (int n = CHAR_LEN; p >= s && n; --n)
       *dest++ = *p--;

   *dest = 0; // write the NULL terminator for CSTRING.
}
```

Pascal version:

```
// This is for output = CSTRING type
procedure p_reverse3(s, dest: PChar);
var
   p: PChar;
   n: Integer;
begin
   p := s;
   while p^ <> #0 do
    Inc(p);

   Dec(p);
   n := CHAR_LEN;
   while (p >= s) and (n > 0) do
   begin
      dest^ := p^;
      Inc(dest);
      Dec(p);
      Dec(n);
   end;

   dest^ := #0; // write the NULL terminator for CSTRING.
end;
```

We go to the end of the source, looking for the C terminator #0, then go back one position to skip it. The maximum number of characters we can copy is *CHAR_LEN*, that should be the same as the UDF declaration, so we make it the second condition in the loop.

After finishing, we still need to terminate our output again with the C terminator.

Notice that none of the reverse functions returns any argument. Because they use an output parameter, in C++ they are of type **void** and, in Pascal, they are simple procedures, not functions.

Functions that work getting and/or setting blobs

Here we want to demonstrate the blob handling facilities that have long been available, although the *blobcallback* structure used was not declared in the public header. It was documented in the technical documentation, with one omission: the existence of a function to seek into blobs. The omission probably had to do with the fact that it works only for stream type blobs, which are not available through SQL DDL statements.

It is important to treat the blob handle as an opaque value, without changing it.

Blobs are handled as input or output parameters, never as return value. If a UDF declaration uses a blob as the return value, it will be converted to the mechanism with an output parameter. For example:

```
SQL> declare external function try int returns blob entry_point 'K'
CON>       module_name 'K';
SQL> declare external function trz int, blob returns parameter 2
CON>       entry_point 'K' module_name 'K';
SQL> ^Z
```

When asked to extract information, isql says:

```
/*  External Function declarations */
DECLARE EXTERNAL FUNCTION TRY
INTEGER, BLOB
RETURNS PARAMETER 2
ENTRY_POINT 'K' MODULE_NAME 'K';
```

```
DECLARE EXTERNAL FUNCTION TRZ
INTEGER, BLOB
RETURNS PARAMETER 2
ENTRY_POINT 'K' MODULE_NAME 'K';
```

Since blobs can be returned only as output parameters, there is no way to signal a NULL output blob. Starting in Firebird v1.5, the blob handle can be set to NULL just before returning, to tell the engine it should assume a null blob is the result from the function. However, the old functionality with input blobs has been retained: so as not to produce an incompatible change there is no way to distinguish an empty blob from a null blob.

Indeed, when the engine has a null representing a blob that it has to pass to a UDF, it creates an empty blob and fills the *blobcallback* structure to make it point to that empty blob.

```
declare external function p_defragment_blob
blob,
blob,
int
returns parameter 2
entry_point 'p_defragment_blob' module_name 'phoenix';
```

Since our function will use an output parameter, we declare it to return nothing.

Also, since C++ translates a reference to a pointer behind scenes, we used that idea to declare the segment size (the integer parameter) as a reference instead of as a pointer. This avoids having to dereference a pointer when its value is going to be used. We decided that we would equate an input blob with zero segments, or one whose total length was zero (an empty blob), to a null input. Thus, to inform the engine that the output parameter is to be treated as null, we set the handle of the output blob to zero and return.

defragment_blob ?

While the name of the function may suggest that it improves the storage of the blob, it should be taken only as an exercise in logic. Defragmenting a blob does not make much sense, since internally very small blobs reside in data pages and bigger blobs reside in special blob pages.

What this function does is to retrieve a blob and then generate an equivalent blob with the same information, but with a given segment size, the same for all segments except possibly the last. That, typically, will have the remainder unless the blob's size is a striaght multiple of the chosen segment size.

Given a segment size of zero, or one that goes above the limit, the maximum segment size (64K-1) will be used. The user should see the same data in the original and the new blob.

The differences are:

- A blob can have segments of different sizes, according to how it was written. Each call to put_segment generates naturally a segment that can vary from size 1 to 65535, depending on how much data was written. The declared segment size in DDL statements is only an indication. The new blob will have all segments the same size, except the last, probably..
- A blob can have a lot of small segments. If a big value (inside the allowed range, of course) is provided, chances are the blob will fit on one segment or will be composed of a few big segments.
- Fewer calls to *get_segment* will be needed on the new blob, and chunks retrieved will be of predictable size, but a performance gain is hard to prove.

To repeat, defragmenting here means that, provided long segments, the new blob will probably end up with fewer segments. It has little effect on how the engine stores the blob and has *no effect* on physical defragmentation at the hard disk level.

```
// This is for input blob struct and output blob struct.
// Null cannot be distinguished from empty blob.
// will return NULL if the input blob is empty.

void p_defragment_blob(const blobcallback* input, blobcallback* output,
                       const SLONG& input_seg_size)
{
   if (!input->blob_number_segments || !input->blob_total_length ||
       input_seg_size < 0)
   {
       output->blob_handle = 0; // hint to the engine -> make output blob NULL
       return;
   }

   int seg_size = input_seg_size;
   if (!seg_size || seg_size > MAXSEG)
       seg_size = MAXSEG;

   UCHAR* const buf = new UCHAR[seg_size];
   const UCHAR* const endbuf = buf + seg_size;
```

```
    UCHAR* p = buf;
    int num_seg = input->blob_number_segments + 1;
    while (num_seg)
    {
        USHORT result_len = 0;
        const int rc = input->blob_get_segment(input->blob_handle,
                                               p, endbuf - p, &result_len);

        p += result_len;
        if (!rc || p >= endbuf) // blob eof, buffer full
        {
           output->blob_put_segment(output->blob_handle, buf, p - buf);
           p = buf;
           if (!rc)
              break;
        }
        if (rc != -1) // -1 happens if we got a a fragment of a segment
           --num_seg;
    }
    delete[] buf;
 }
```

First, we allocate a buffer of our desired output segment size. Then we go into a loop for one more than the number of segments the input blob has. This is because we want to handle two conditions at the same time instead of repeating logic after the loop. When the buffer is full, we write it to the output blob, but also, when the input blob is exhausted, we write the remainder to the output blob.

For safety, we initialize the length of the retrieved segment to zero before calling *get_segment*. The result of *get_segment* has three values:

- one means we got a full segment
- zero implies there was not data retrieved (typically, end of blob)
- -1 is to signal that we got all our buffer with data, but it was not enough to retrieve a full segment (since the segment was bigger than the buffer, the next call to get_segment will retrieve the remaining data in the segment).

Fragment of a segment

The third case is known as getting a fragment of a segment. Take note we use that information to decide whether or not to decrement the number of segments that we have still to read: if we got a fragment, then we still did not read a whole segment.

Since the logic to write the output is done for both the *buffer full* and *end of blob* conditions, we only break the loop if there is no more data (rc is zero). At the same time, after we write the buffer, we reset the pointer that indicates the position in the buffer that will be passed to *get_segment*.

Handling the available space in the buffer is the only logic that needs careful attention here. Our main loop tests *num_seg* only for security, since the logic that writes the output blob will terminate the loop is there is no more data.

Finally, after the loop, we deallocate our buffer.

Since we will never move *buf* and *endbuf*, we can mark them as const after the asterisk, meaning the pointer itself can not be changed. Although the data pointed by *buf* can still be changed, *endbuf* can neither move itself nor modify the data.

Now our Pascal version. Remember, we are not doing a real defragmentation in the internal structures of the database. We are simply writing the blob with less format. This way, a single *get_segment*, with buffer sufficient to accommodate the biggest segment, will be called fewer times than for a lot of small segments, even if the buffer to read the blob is big.

However, since we can control the size of the segments, we could eventually write a new blob that has more but smaller segments than the original blob:

```
// This is for input blob struct and output blob struct.
// Null cannot be distinguished from empty blob.
// will return NULL if the input blob is empty.
procedure p_defragment_blob(input, output: Pblobcallback;
                             var input_seg_size: SLONG);
var
   seg_size, num_seg, rc: Integer;
   buf, endbuf, p: PUchar;
   result_len: USHORT;
begin
   if ((input^.blob_number_segments = 0) or (input^.blob_total_length = 0) or
      (input_seg_size < 0)) then
   begin
      output^.blob_handle := nil; // hint to the engine -> make
```

```
                                //output blob NULL
      Exit;
   end;

   seg_size := input_seg_size;
   if (seg_size = 0) or (seg_size > MAXSEG) then
      seg_size := MAXSEG;

   GetMem(buf, seg_size);
   endbuf := buf;
   Inc(endbuf, seg_size);
   p := buf;
   num_seg := input^.blob_number_segments + 1;
   while num_seg > 0 do
   begin
      result_len := 0;
      rc := input^.blob_get_segment(input^.blob_handle, p,
                                PChar(endbuf) - PChar(p), result_len);

      Inc(p, result_len);
      if (rc = 0) or (PChar(p) >= PChar(endbuf)) then // blob eof, buffer full
      begin
         output^.blob_put_segment(output^.blob_handle, buf,
                                PChar(p) - PChar(buf));
         p := buf;
         if rc = 0 then
            break;
      end;
      if rc <> -1 then // -1 happens if we got a fragment of a segment
         Dec(num_seg);
   end;
   FreeMem(buf, seg_size);
end;
```

The variable *input_seg_size* was declared as **var** as the means of declaring a variable declared by pointer without having to use pointer dereferencing notation to use the variable. Recall that **const** in Pascal is compiler dependent). To avoid any problem of changing it accidentally, it is copied into a variable named *seg_size*, that is adjusted to the maximum allowed segment size, in case the requested size is out of range.

Before that, we made the same assumption as the C++ code, that a blob with no segments, a blob with total length of zero, or a requested segment size less than zero meant returning a NULL blob. We do that with the aforementioned hint to the engine: putting the handle in the blob structure to nil.

We are going to play with the direct memory allocation and deallocation routines here. We request dynamic memory for the requested segment size and set our *endbuf* indicator to the same address as the new memory chunk. Then, we increment it by *seg_size*, making it point one byte past the allocated buffer.

Unlike C++, where the normal operators can be used, we are forced to use the Inc procedure to increment a pointer. Also, unlike C++, we cannot tell the compiler we do not want to move the address of *buf* and *endbuf* (to be able to deallocate and to mark the end, respectively), so we should check that we did not alter them in the rest of the function.

We use *p* as an indicator of the position available in the buffer to read a segment. For example, with a buffer size of 10 and two segments of length 3 and 4, *p* and *buf* are the same in the first get_segment call. In the next cycle, *p* is 3 bytes ahead (remaining space is 7) and in the third iteration, *p* is 7 bytes ahead (remaining space is 3).

As in the C++ code, our *num_seg* is set to one more than the number of segments to guarantee that the loop will be able to write any remnants from the source blob into the target blob. In practice, the condition of the **while** loop will never be false because the loop will be broken from inside it.

One annoying difference with C++ is the cast to PChar to be able to take the difference in bytes between two pointers. The reason is that such a difference is managed natively in Pascal only for pointers to normal Char. Here, because the blob may contains any kind of information, we are using unsigned Char to have each position treated as an 8-bit byte.

Now, inside our loop, we set the *result_len* to zero for safety, then read a segment according to our remaining space, our objective being to fill the segment buffer and then write it to the destination. Again we have to use Inc to position *p*, the beginning of the available space in the buffer. If we got *eof* (result of *get_segment* being zero) or the available space is zero (*p* reached the position of *endbuf*), then we first write the buffer in the destination blob. We position *p* at the beginning of buf again for a possible new loop, once again making the full buffer available to *get_segment*.

Finally, if the cause of the code block being executed was *eof*, we break the loop.

Otherwise, if the loop is not broken at that point, we go to the place where the result from *get_segment* is checked against -1, to check whether a whole segment was available to be read, or just a fragment of a segment. We decrement the *num_seg* variable, as expected, if we get a whole read, since we are counting full segments.

> Imagine our source blob has a segment of length 20, our output segment size is 9 and our buffer is totally empty. In that scenario, there will be two get_segment calls that will get -1 as result, since they are still reading a fragment of the same segment. The third will read two bytes and will return zero, at which point we decrement our segment counter.

This counter is really just a safety net in case something unexpected happens. Under normal circumstances, the "break" inside the loop will finish it. After the loop, we return the dynamic memory that we allocated.

Finally, a clarification with the pointer game, where we simplified the names and got rid of the casts just to show the essential idea:

```
get_segment(blob_handle, p, endbuf - p, result_len);
```

means we are reading at *p*, the beginning of the available space, and the amount of available space is the difference between the end of the buffer and the current available space. However, when writing,

```
put_segment(blob_handle, buf, p - buf);
```

we are always writing from the beginning of the buffer (*buf*) and we can't pad the last segment with random characters or garbage from the previous loop, hence the number of bytes to be written is the available position minus the beginning of the buffer. In both cases, we forced to PChar to be able to compile and to make sure the difference is measured in bytes.

Blob generation

If we wanted to test our previous routine, we would have to keep a database with blobs of several combinations of segment sizes and number of segments to validate it. Instead, we have a built a function that can create and populate a blob given two parameters, the starting segment size and the number of segments.

```
declare external function p_generate_blob
blob,
int,
int
returns parameter 1
entry_point 'p_generate_blob' module_name 'phoenix';
```

For each new segment, we increment the segment size by one and we take care to not go beyond the maximum segment size. If we reach MAXSEG, we will keep writing the rest of the segments requested with that maximum size.

While it may seem strange that we are going to pass blobs between two functions without using a table, it is perfectly natural: a blob is temporary until its blob identifier is assigned to a blob field in a table and that operation is committed.

The engine automatically discards temporary blobs when they are no longer needed. This happens typically when the request that used them finishes or when the transaction under which they were created is committed or rolled back. A temporary blob that gets assigned to a blob field in a table is said to become a *materialized blob*.

In this case, we are playing exclusively with temporary blobs, but there is nothing that prevent us from assigning the result of this function to a table's field. Temporary blobs can be read the same way as materialized blobs.

Our function first checks that the desired starting segment size and the number of segments are one at least; otherwise it returns without doing anything. We then proceed to declare an array of the maximum allowed length.

Granted, we could have been more careful and, given the starting segment size and the number of segments, could have determined the maximum needed buffer size as

```
min(seg_size + num_seg - 1, MAXSEG)
```

and allocated it dynamically. However, we went for simplicity here.

We might also have created the buffer, as we did now, with its maximum size, but outside any function, as a static variable and have initializes it using platform-dependent facilities when the library is loaded. We wanted to avoid writing code that was dependent on the operating system, though.

We chose to initialize it simply with ASCII values representing the printable characters zero to nine and going back to zero, and so on, until all the buffer is filled. If you consider this technique a waste of time, then a global buffer is feasible, considering the buffer is only read by this function.

As an alternative to operating system hooks to initialize a library that has been loaded, a C++ class with a constructor can be used (where the only instance would be global) or the Delphi initialization section may be used in the same unit where the UDFs are defined.

```
// This is for output blob struct, to create a blob to test the previous
function.

void p_generate_blob(blobcallback* output, const SLONG& input_start_size,
                     const SLONG& input_num_seg)
{
   if (input_start_size < 1 || input_num_seg < 1)
       return;

   UCHAR buf[MAXSEG];
   for (int i = 0; i < MAXSEG; ++i)
       buf[i] = '0' + static_cast<UCHAR>(i % 10);
```

```
    int seg_size = input_start_size;
    if (seg_size > MAXSEG)
        seg_size = MAXSEG;

    for (int num_seg = input_num_seg; num_seg; --num_seg)
    {
        output->blob_put_segment(output->blob_handle, buf, seg_size);
        if (++seg_size > MAXSEG)
           seg_size = MAXSEG;
    }
}
```

The logic is really simple: we ensure that our original segment size does not surpass MAXSEG or use this value and go into a loop that calls *put_segment* as many times as the number of segments requested. In each iteration, we increase the segment size. If we hit MAXSEG, we continue using this value for the rest of the iterations.

Here is our Pascal version. Instead of messing with a C++ program (the server) running a Pascal DLL under the Delphi debugger, we chose to put compile-time conditional code to put a dialog on screen, since we are going to debug locally (otherwise, we will halt the function execution on the remote machine, of course). Apart from a Spanish message, the logic is the same as the C++ version:

```
// This is for output blob struct,
// to create a blob to test the previous function.
procedure p_generate_blob(output: Pblobcallback; var input_start_size: SLONG;
                           var input_num_seg: SLONG);
var
   i, seg_size, num_seg: Integer;
   buf: array[0..MAXSEG - 1] of UCHAR;
begin
{$IFDEF GUI_MSG}
   ShowMessage('Aquí voy, generate_blob.');
{$ENDIF}
   if (input_start_size < 1) or (input_num_seg < 1) then
      Exit;

   for i := 0 to MAXSEG - 1 do
      buf[i] := Ord('0') + UCHAR(i mod 10);
```

```
    seg_size := input_start_size;
    if seg_size > MAXSEG then
    seg_size := MAXSEG;

  {$IFDEF GUI_MSG}
    ShowMessage(Format('%p %p %.10s', [output, output^.blob_handle,
                       PChar(@Buf[0])]));
  {$ENDIF}
    for num_seg := input_num_seg downto 1 do
    begin
      output^.blob_put_segment(output^.blob_handle, @buf[0], seg_size);
      Inc(seg_size);
      if seg_size > MAXSEG then
        seg_size := MAXSEG;
    end;
  end;
```

If the starting size or the number of segments are not positive, we exit, returning an empty blob (not a null blob, because the engine already created an empty blob for us to fill).

We fill our buffer with the sequence '0' to '9', then '0' to '9' again and so on. The usage of *Ord()* is needed since we are dealing with unsigned characters. We adjust the segment size, produce a dialog box with debugging information and go in the loop of writing each segment with as much information as we can from the buffer with the sequence '0'..'9', taking care to not go beyond the maximum allowed segment size, that can be reached if the initial size incremented by the number of segments to be written is bigger than MAXSEG.

Samples for testing

Since we have a function to rewrite the blob in segments of fixed size and we have a function to generate blobs of monotonically increasing segment sizes with predicatable data, we need a function that can be applied to them, to verify that:

- The UDF generate_blob is working as expected.
- The UDF defragment_blob, after being applied to a database field or the output from generate_blob, still retains all the data that it began with and no data corruption has happened as a result of changing the segment size. For validation purposes, the user should see the same data without considering segment boundaries that are marked with a comma.

- ❑ Any function that outputs a blob works as expected without the need to first assign the blob to a table's field and then read from that table.

```
declare external function p_sample_blob
blob,
int
returns varchar(150) free_it
entry_point 'p_sample_blob' module_name 'phoenix';
```

The function is called sample_blob because it takes a sample of each blob's segment.

For readability, we want to test small blobs, so we have limited the output of the function to 150 characters. If more is desired, then the function has to be declared with a bigger varchar size as output. again. (It can be declared with another name, leaving the original declaration unchanged.)

If the data returned by the UDF is bigger than the declared output length, then the engine will not be able to assign it to the preallocated buffers and will complain about data truncation. Nothing harmful will happen, since it is the engine that prevents a buffer overrun. However, to avoid having more parameters in the function, we hard-coded some values.

The first part of the function is used in verification. If a blob has no segments or length zero, we'll assume the empty blob is a null blob and will return NULL accordingly. We also reset the sample to be not bigger than the biggest segment in the blob (this information is known from the *blobcallback* structure).

Next, we get into a formula to determine the maximum sample: given a sample, we need one more byte for a comma (we separate segments by a comma), then that value multiplied by the number of segments in the blob cannot surpass the practical limit of a row's size in the engine. For the sake of accuracy, we take into account the two bytes that the varchar fields need.

Finally, we determine that we do not want the sample of each segment to be bigger than an arbitrary length, 30. Imagine if you have 50 segments of maximum length 100 and you requested a sample of 15, this will produce an output of approximate length 15 * 50, or 750 bytes, not easily readable for a single field.

Let's consider the following limitations:

- ❑ The calculation of the output's length is the worst case, using the biggest blob's segment as recorded by the engine. Obviously there can be much smaller segments (even one byte in length) that may make the result smaller.
- ❑ The UDF has no way to know the declaration of the result, so it assumes it has the full row size allowed by the engine as the maximum. If you get truncation errors, either you will have to be more modest in your sample size (chunk of each segment that is

shown) or you will have to declare the result to be wider. Unfortunately, field declarations are limited to values smaller than 32K, so the UDF is over-optimistic in assuming it can return values a few smaller than 64K (MAXROW).

```
// This is for input blob struct and output VARCHAR(150)
// Null cannot be distinguished from empty blob, but the routine
// will return NULL if the blob is empty. We will use FREE_IT.
// Really, we will return any length. When we surpass 150, the engine
// will not be able to copy the result to another location and will
// generate the "string truncation" message.
paramvary* p_sample_blob(const blobcallback* input, const SLONG& input_len)
{
   if (!input->blob_number_segments || !input->blob_total_length
       || input_len < 1)
   {
       return 0;
   }

   int sample_len = input_len;
   // adjust sample_len to something reasonable

   // a) cannot be bigger than segment max size; we do not know min size
   if (sample_len > input->blob_max_segment)
       sample_len = input->blob_max_segment;

   // b) cannot be bigger than max row size near 64K div by num segments
   // (1 + sample_len) * num_seg + VARCHAR_PREFIX < MAXROW
   // where 1 is for each comma appended to separate segments.
      // This gives us:
   // sample_len < (MAXROW - VARCHAR_PREFIX) / num_seg - 1;
   int num_seg = input->blob_number_segments;
   const int max_sample = (MAXROW - VARCHAR_PREFIX) / num_seg - 1;
   if (sample_len > max_sample)
       sample_len = max_sample;

   //c) We set the arbitrary limit in 30
   if (sample_len > 30)
```

```
        sample_len = 30;

    const USHORT seg_lim = input->blob_max_segment > MAXSEG ?
        MAXSEG : (USHORT) input->blob_max_segment;
    UCHAR* const input_buf = new UCHAR[seg_lim];
    paramvary* const v = reinterpret_cast<paramvary*>(
                ib_util_malloc((sample_len + 1) * num_seg + VARCHAR_PREFIX));
    UCHAR* output_buf = v->vary_string;
    v->vary_length = 0; // let's play safe if we have to exit prematurely

    for (; num_seg; --num_seg)
    {
        USHORT result_len = 0;
        const int rc = input->blob_get_segment(input->blob_handle,
                                               input_buf, seg_lim,
                                               &result_len);
        // This cannot happen in our limited world,
                // so if it happens, we quit.
        if (rc == -1) // got only a fragment of a segment?
           break;

        if (result_len > (USHORT) sample_len)
           result_len = sample_len;

        for (const UCHAR* ip = input_buf; result_len; --result_len)
           *output_buf++ = *ip++;

        *output_buf++ = ',';
    }
    delete[] input_buf;
       // ptrdiff_t->USHORT
    v->vary_length = static_cast<USHORT>(output_buf - v->vary_string);
    return v;
}
```

For the rest of the function, we first verify that our buffer is enough to read one segment at a time. This is for simplicity, since with only one call to *get_segment* we will get even the largest segment and will take a few bytes from it (the sample). If we had to deal here with segment fragments, the logic would be more complex.

We made an additional check to ensure that the stated biggest segment is in the range 1 to MAXSEG. This happens because the supposed limit for a segment in the engine is 655535 but the field *blob_max_segment* in the *blobcallback* structure is a signed 32-bit quantity, which puts it well beyond the limits of an unsigned 16-bit quantity (the type of the segment size accepted by *get_blob* and *put_blob*).

Whether this condition can arise in a real scenario is unknown, so, inside the loop, we made the test "if get_segment says we only read a fragment of a segment, we quit the loop, since we do not have logic to cope with that condition".

We allocated our buffer for the worst case (biggest segment) using the native facilities in the language, i.e., the "new" operator for C++. We cannot call ib_util_malloc because it is only useful for allocating memory that will be passed from the UDF to the engine, and not for memory used inside the UDF.

In this case, our temporary buffer is deallocated before the function exists. However, the next call makes use of ib_util_malloc because our declaration indicates that the function will allocate and return varchar and that the engine is reponsible for getting rid of it (free_it).

We add one to the sample length, to make room for the comma that we will use to mark the separations between two consecutive segments and then multiply that quantity by the number of segments that the incoming blob has.

Finally, we add VARCHAR_PREFIX (two bytes) to account for the length information a VARCHAR needs at the beginning, pass this value to ib_util_malloc and cast the result to be of type *paramvary* (remember, we declared that the function would return a VARCHAR).

We declare a pointer to the beginning of the data part of the result (the *vary_string* member in *paramdsc*) for use when copying samples from each segment in the input blob. We cannot move the variable pointing to "v" (the *paramdsc*) because we need to return that pointer. As a precaution, we assign zero to the length of the result.

As many times as there are segments, we set the number of bytes read to zero for safety, then try to read a segment into our buffer. If we encounter the previously described condition, where the buffer is insufficient and we must read just a fragment, we quit the loop, adjust the bytes read so they are not bigger than our desired sample's size and use the same variable to copy bytes from the buffer to the output variable.

Finally, we append the comma to the output buffer and go for another cycle.

Once the loop is finished, we free our temporary buffer (not the result!) and adjust the length. Calculating the effective length (which may be smaller than the declared length) is easy since both *vary_string* (the starting value for *output_buf*) and *output_buf* itself are of type pointer to byte, so their difference is the number of bytes that were written.

We then return the varchar variable (a pointer). The engine will take it, read it and deallocate it when needed.

We could have been paranoid and, in the loop that copies bytes, ensured we did not write more than *seg_lim* bytes into the result, which would cause a buffer overrun in dynamic memory. If it happened, it would be due to either a logic bug in this function or an inconsistency in the information about the blob that was provided by the engine , i.e. possibly blob corruption in the database.

Just before ending the loop, we append the semicolon with the statement

```
            *output_buf++ = *ip++;
```

If the blob contains some unprintable or blank characters, that trailing semicolon will indicate visually the real end of the sample. For purists, this has the nasty effect that there will be a trailing semicolon, after the last segment sample. If you do not want to see the trailing semicolon, you can write instead:

```
         if (num_seg > 1)
            *output_buf++ = ',';
```

Now, our Pascal version, again with some Spanish messages and debugging information in dialog boxes for local usage.

Since we now have the ability to return a null string (using a descriptor) we do so if the parameters are invalid or out of range. Then we proceed to adjust the sampling lengthas in the C++ version. The limit of 30 bytes per segment is arbitrary but with a declared returned length of 150 bytes, at least we can show five segments if there are five or more.

We do the extra check of adjusting *seg_lim* for the sole reason that the maximum segment in the current engine is MAX_SEG but, for some reason, the blobcallback structure provides *blob_max_segment* with more capacity than it currently needs.

```
// This is for input blob struct and output VARCHAR(150)
// Null cannot be distinguished from empty blob, but the routine
// will return NULL if the blob is empty. We will use FREE_IT.
// Really, we will return any length. When we surpass 150, the engine
// won't be able to copy the result to another location and will
// generate the "string truncation" message.
function p_sample_blob(input: Pblobcallback;
                       var input_len: SLONG):  Pparamvary;
var
   sample_len, num_seg, max_sample, rc: Integer;
   seg_lim: USHORT;
   input_buf, output_buf, ip: PUChar;
   v: Pparamvary;
   result_len: USHORT;
begin
   if ((input^.blob_number_segments = 0) or (input^.blob_total_length = 0)
         or (input_len < 1)) then
   begin
{$IFDEF GUI_MSG}
      ShowMessage('Retorno null, sample_blob.');
{$ENDIF}
      Result := nil;
      Exit;
   end;

   sample_len := input_len;
   // adjust sample_len to something reasonable

   // a) cannot be bigger than segment max size; we don't know min size
   if sample_len > input^.blob_max_segment then
      sample_len := input^.blob_max_segment;

   // b) cannot be bigger than max row size near 64K div by num segments
   // (1 + sample_len) * num_seg + VARCHAR_PREFIX < MAXROW
   // where 1 is for each comma appended to separate segments. This gives us:
   // sample_len < (MAXROW - VARCHAR_PREFIX) / num_seg - 1;
   num_seg := input^.blob_number_segments;
   max_sample := (MAXROW - VARCHAR_PREFIX) div num_seg - 1;
   if sample_len > max_sample then
      sample_len := max_sample;

   //c) We set the arbitrary limit in 30
   if sample_len > 30 then
      sample_len := 30;

   if input^.blob_max_segment > MAXSEG then
      seg_lim := MAXSEG
   else
      seg_lim := USHORT(input^.blob_max_segment);
   GetMem(input_buf, seg_lim);
   v := ib_util_malloc((sample_len + 1) * num_seg + VARCHAR_PREFIX);
   output_buf := @v^.vary_string[0];
   v^.vary_length := 0; // let's play safe if we have to exit prematurely

   for num_seg := num_seg downto 1 do
   begin
      result_len := 0;
      rc := input^.blob_get_segment(input^.blob_handle, input_buf,
                                    seg_lim, result_len);
      // This cannot happen in our limited world, so if it happens, we quit.
      if rc = -1 then // got only a fragment of a segment?
         break;

      if result_len > USHORT(sample_len) then
         result_len := sample_len;

      ip := input_buf;
      while result_len > 0 do
      begin
         output_buf^ := ip^;
         Inc(output_buf);
         Inc(ip);
         Dec(result_len);
      end;

      output_buf^ := Ord(',');
      Inc(output_buf);
   end;
   FreeMem(input_buf, seg_lim);
   // ptrdiff_t to USHORT
   v^.vary_length := USHORT(PChar(output_buf) - PChar(@v^.vary_string[0]));
{$IFDEF GUI_MSG}
   ShowMessage(Format('%d %.30s', [v^.vary_length,
               PChar(@v^.vary_string[0])]));
{$ENDIF}
```

```
    Result := v;
  end;
```

We allocate memory directly from the basic facilities to contain our read blob but, to allocate the output, we use the Firebird routine since it will be passed to the engine and the engine should deallocate it. We read a whole segment at a time but write only the requested "sample" size of each segment (or less if the segment was smaller). We copy the bytes one by one.

Again, our **while** loop is the counterpart of the more compact C++ *for*, due to the absence of a Pascal syntax to mix the increment and decrement operators in the same statement.

Finally, we add the comma to separate the segment samplings visually, adjust the output pointer one byte ahead, again, and go into another cycle. When the cycle is finished, we return the memory for the input buffer, adjust the length in the *Paramvary* structure, call (if enabled) a dialog with debugging information and assign the allocated *Paramvary* to the result of the function.

Side note—blobcallback

It is intriguing that, in all those years, we did not receive any question from newcomers about the magic in the *blobcallback* structure. Either nobody noticed it or everybody knew about it or all programmers took it for granted.

Pascal translations have existed for years so you did not need to know our ibase_custom.pas (developed only recently) to question why it works. In C++, there is no big difference between a struct (record in Pascal) and a class (more or less a class in Pascal) because a struct is considered a class with all members being public. Therefore, if you define functions inside a struct, they will be considered methods and the compiler will pass the "this" pointer behind scenes. This would make *blobcallback* a different beast depending on whether C or C++ compiled the header.

Apart from the fact that plain old C will not like functions inside a structure, *blobcallback* in ibase.h only declares pointers to functions, because the addresses are filled by the engine. In Pascal, records are simple records, not objects, so even if you manage to declare functions inside them, they will not receive any "self" pointer. But since the engine needs to fill in these addresses, again the Pascal version declares pointers to functions.

Therefore, we avoid any difference in interpretation from different language compilers. The compilers do nothing to pointers to plain functions declared inside structures or classes. Of course, C++ has (rarely used) pointers to members and Delphi has (widely used) declarations of procedures and functions "of object" for delegation.

Functions that work with internal descriptors

We reached the point where adventurous programmers want to go: directly using information from the engine in the format that the engine uses. Granted, using VARCHAR variables directly is already making use of internal structures, but here we go a step further: we use the descriptor, the structure indicating several run-time attributes of the data.

At a low level where the engine does not care about field names, descriptors point to the each field's data and explain how to interpret the data they hold, in a generic variable of type pointer to byte.

A structure known as *format* describes the layout of a record, including an array of descriptors. The format is not available externally, but the descriptor is.

You should be extremely careful when dealing with descriptors, since the engine is not giving you a copy of its data or its metadata, but direct access to the in-memory structures holding data. A mistake here may cause side effects hard to track. As a general rule, NEVER modify an input descriptor in any way. If you use C++, declare the input descriptor(s) as pointers to constant data, thus enabling the compiler to catch programming errors.

We explained previously that descriptors pose an additional challenge: the lack of data type verification in against the UDF declaration. To explain this, let's analyze these facts:

- Input parameters by value are deprecated and cannot be declared through SQL DDL, so they are not considered here.
- With input parameters declared by reference or by reference with null signaling, the engine checks that each parameter type exactly matches the UDF declaration. Since this is too restrictive, the engine checks for compatible types and does the conversions if possible, for example to pass an integer to a function that wants a double precision value. But if no conversions are available, the engine will say that there is not a match between the UDF declaration and the intended usage and will not compile the statement.

  There seems to be evidence that UDFs can be overloaded (based on internal code that compares what it calls *homonyms*) but a unique index on the UDF name in the system table rdb$functions makes impossible to declare two UDF's with the same name and different parameters (probably the engine would have to check that the combination of entry_point and module_name is unique).
- With input parameters declared by blob,, the engine will pass blobs and arrays to the UDF and will wrap their internal structures in the externally visible *blobcallback* structure. Probably arrays are undesirable here, but the check is left to the user. On

the other side, the UDF might want to print or save arrays in raw format for debugging, for example.

- With input parameters declared by scalar_array, the engine will check at run-time that it is getting an array and will pass a higher level, simplified structure named scalar array descriptor.
- However, with input parameters by descriptor, the engine does not care which data it is passing to the UDF. The declaration of the UDF's parameters becomes only a formality to keep happy the places that would otherwise complain about unknown data types.

  For any input parameter declared by descriptor, the engine can pass a descriptor pointing to integers, doubles, timestamps, strings, blobs (pointing to the internal structure, not *blobcallback*) and arrays (again, pointing the internal array structure that is, after all, a blob.

  You have to deal with all this variability. Since you cannot raise exceptions from UDFs, your only indication that you do not accept the given parameters or that something went wrong would be to return NULL v/s whatever arbitrary value (a pointer to a global constant, for example) or to return an integer with values zero or one. (Boolean UDFs may have beeen considered, but they were not implemented.

  If you are just looking for a way to detect SQL NULL without all these details, prefer the new reference with null signaling mechanism, explained previously with the "lastchar" family of functions.

UDFs with descriptors

Our next example outputs a combination of two strings. To demonstrate descriptors, we receive both input parameters by descriptor.

Since we only want to deal with strings, we built a helper named *get_string* for use when the descriptor indicates it contains a CHAR, VARCHAR or CSTRING type. It will return a pointer to byte pointing to the data (the string. It will also set the length of the returned string in the second parameter, received by reference.

If the descriptor does not contain one of the three string types, get_string simply returns the null pointer and leaves the length parameter unchanged. We could try to make this function available to the engine for experimentation but we elected to keep it private.

Notice the function says clearly it will not change the input descriptor. The price one pays for the complexity of dealing with a descriptor is partly mitigated because descriptors are efficient: they are pointing directly to the internal data, so no translation or copy to a temporary buffer needs to be done by the engine.

```
// Do not publish this helper.
// Given a descriptor, it gets the string and the length if it is one of the
// three string types recognized by the engine.
// It does not put the terminator.

const UCHAR* get_string(const paramdsc* p, USHORT& length)
{
   if (!p || !p->dsc_address || (p->dsc_flags & DSC_null))
       return 0;

   const UCHAR* rc = 0;
   switch (p->dsc_dtype)
   {
   case dtype_text:
       rc = p->dsc_address;
       length = p->dsc_length;
       if (length > 0)
       {
          int loop = length - 1;
          while (loop >= 0) // let's ignore trailing blanks
          {
             if (rc[loop] == ' ')
                 --loop;
             else
                 break;
          }
          length = (USHORT) (loop + 1);
       }
       break;
   case dtype_varying:
       {
          const paramvary* v =
                                  reinterpret_cast<paramvary*>(p->dsc_address);
          rc = v->vary_string;
          length = v->vary_length;
       }
       break;
```

```
        case dtype_cstring:
            {
               rc = p->dsc_address;
               length = 0;
               for (const UCHAR* cp = rc; *cp; ++cp)
                  ++length;
            }
            break;
        default:
            break;
        }
        return rc;
    }
```

First, we'll check that the descriptor is useful. It is the equivalent of SQL NULL if we find any of the following conditions:

- The descriptor itself is the null pointer. It can happen if the UDF receives the literal NULL directly in one parameter. Remember, our next function (the visible one) will pass whatever descriptor it gets to this function.
- The descriptor's data pointer (address) is the null pointer. Although it is almost impossible for this to happen, it is a risk we do not want to take. No data would mean SQL NULL.
- The descriptor's flags include the activated null indicator.

When any of these conditions is met, we simply return the null pointer. Nothing more to do.

Now, if we got a valid descriptor, we initialize the return pointer to null and we'll have to branch, based on the data type (*dsc_dtype*) as follows:

- For CHAR, the address is the data and the length is taken from the descriptor's length. Since we want to trim trailing blanks, we have extra logic to determine the trimmed length.
- For VARCHAR, the address is really the address of the *paramvary* struct we used in a previous example, hence we cast the address to *paramvary* and get the data and the length. If we wanted to be more cautious, we would compute the minimum of *dsc_length* – VARCHAR_PREFIX and *vary_length*. Typically the engine does that for safety, since for VARCHAR, the descriptor's length includes the length indicator in *paramdsc*, a 16-bit unsigned quantity.
- For CSTRING, the address is the data, but we do not trust the descriptor's *dsc_length*. Instead, we calculated the length of the string as strlen() would do.

The disadvantage of that code is that, if we get a string in charset BINARY (OCTETS), it could have embedded ASCII nulls and therefore, we would miscalculate the length.

Again, if we want to be more accurate, the descriptor contains charset information (something that's not available otherwise) and we could skip the loop if we receive charsets we know are problematic (binary plus MBCS charsets including UNICODE_FSS).

- ❑ For other data types, it simply does nothing, since the return value is already initialized to be the null pointer.

As the function's comment says at the top, this function returns exactly what it gets. For people used to C strings, the function does not put the null terminator or it would need to copy the buffer. This is why the function, if it returns a not null pointer (meaning it got one of the three string types and nothing else) the length is provided in the variable passed as the second parameter. Of course, for CSTRING the value comes already with the null terminator.

Here is our Pascal version. If we detect SQL NULL as either nil input pointer, descriptor's address (data) pointer being nil or the descriptor's flags indicating NULL, we'll return nil:

```
// Do not publish this helper.
function get_string(p: Pparamdsc; var length: USHORT): PUchar;
var
  rc, cp: PUChar;
  loop: Integer;
  v: Pparamvary;
begin
  if ((p = nil) or (p^.dsc_address = nil)
      or ((p^.dsc_flags and DSC_null) > 0)) then
  begin
    Result := nil;
    Exit;
  end;

  rc := nil;
```

```
case p^.dsc_dtype of
   dtype_text:
   begin
      rc := p^.dsc_address;
      length := p^.dsc_length;
      if length > 0 then
      begin
         loop := length - 1;
         while loop >= 0 do // let's ignore trailing blanks
         begin
            if (PChar(rc) + loop)^ = ' ' then
               Dec(loop);
            else
               break;
         end;
         length := USHORT(loop + 1);
      end
   end;

   dtype_varying:
   begin
      v:= Pparamvary(p^.dsc_address);
      rc := @v^.vary_string[0];
      length := v^.vary_length
   end;

   dtype_cstring:
   begin
      rc := p^.dsc_address;
      length := 0;
      cp := rc;
      while cp^ <> 0 do
      begin
         Inc(length);
         Inc(cp);
      end;
   end;
```

```
        else; // nothing for now
    end;
    Result := rc;
end;
```

We initialize our intermediate variable *rc* to nil immediately to avoid more checks later. We try to report effective lengths. For CHAR, we skip trailing blanks; for VARCHAR, we get the length from the *Paramvary* structure and for CSTRING, we count bytes until we get the terminator.

There might be several ways to do the same operation. We could have compared the *vary_length* member of the *Paramvary* against that of the *Paramdsc* (always two bytes until FB2 at least) taking the minor value for VARCHAR and assuming that the *dsc_length* member of the *Paramdsc* has correct value for CSTRING. For CHAR and CSTRING, the descriptor's data member is the string, whereas for VARCHAR we have to convert it to *Paramvary* or skip two bytes, if we want to hardcode known facts (we can use the VARCHAR_PREFIX constant). Finally, if nothing is found that resembles a string, we simply return nil and the length parameter is undefined.

```
declare external function p_intersperse
varchar(30) by descriptor,
varchar(30) by descriptor
returns varchar(60) by descriptor free_it
entry_point 'p_intersperse' module_name 'phoenix';
```

Now we have the function that will intersperse two strings, taking one byte from the first argument, then one byte from the second argument, another from the first and so on.

First, we use our auxiliary previous function to get both strings. Since *get_string* will return NULL when the input is invalid or NULL, we do the same here.

Now, we calculate the total output length. Because we are using two unsigned 16-bit quantities, we do the sum in two steps in a signed 32-bit quantity to ensure we can get the correct sum (no wrapping will happen), then adjust the length so it won't be bigger than our maximum row size (we truncate silently) minus two bytes for the varchar indicator.

Our declaration says that we accept two inputs of length 30. Although the engine will not obey our data types since it's using descriptors, it will obey the length! This means if you pass a string bigger than 30 characters, you will get the classical string truncation error. Same on the returned value, if the function happens to create something wider than 60 characters, the engine won't be able to copy the result to the internal buffers.

If you want to experiment with longer strings, you can fix the declaration. Since the function is working with descriptors, it will retrieve the length dynamically.

```
// This is for arbitrary input, but only will accept VARCHAR or CSTRING.
// Finally we handle CHAR, too.
// If either or both input arguments are NULL, NULL is returned; same if we
// do not get the desired input data types.
// FREE_IT is used for the returned data and the descriptor itself.
paramdsc* p_intersperse(const paramdsc* p1, const paramdsc* p2)
{
   USHORT p1_len = 0;
   const UCHAR* p1_string = get_string(p1, p1_len);
   if (!p1_string)
       return 0;

   USHORT p2_len = 0;
   const UCHAR* p2_string = get_string(p2, p2_len);
   if (!p2_string)
       return 0;

   int total_len = (int) p1_len;
   total_len += (int) p2_len; // sum made in two steps
                                 // to avoid wrapping before assignment.

   if (total_len + VARCHAR_PREFIX > MAXROW)
       total_len = MAXROW - VARCHAR_PREFIX;

   paramvary* const v =
             reinterpret_cast<paramvary*>(ib_util_malloc(total_len
             + VARCHAR_PREFIX));
   UCHAR* output_buf = v->vary_string;
   v->vary_length = 0; // let's play safe if we have to exit prematurely
   paramdsc* const d =
             reinterpret_cast<paramdsc*>(ib_util_malloc(sizeof(paramdsc)));
   d->dsc_address = reinterpret_cast<UCHAR*>(v);
```

```
    while (total_len)
    {
        if (p1_len)
        {
           *output_buf++ = *p1_string++;
           --p1_len;
           --total_len;
        }
        if (p2_len && total_len)
        {
           *output_buf++ = *p2_string++;
           --p2_len;
           --total_len;
        }
    }
    v->vary_length = static_cast<USHORT>(output_buf - v->vary_string);
                               // ptrdiff_t to USHORT
    d->dsc_dtype = dtype_varying;
    d->dsc_length = v->vary_length + VARCHAR_PREFIX;
    d->dsc_flags = 0;
    return d;
 }
```

After we have our length calculated, we proceed to allocate dynamic memory for the result. Here we will return a descriptor, so we should allocate both the *paramvary* and the *paramdsc* with the facility *ib_util_malloc* so the engine can deallocate them.

Keep in mind that this function will work as desired only on FB v2, since in v1.5 it won't deallocate the descriptor (only the varchar structure), so a small memory leak will happen in each call.

We set the varchar's length to zero for safety, allocate our descriptor and assign the *paramvary* to the descriptor's address data member. We know we should iterate until *total_len* is exhausted (remember, *total_len* may be smaller than the sum of the lengths of the two input parameters). Since we don't make any assumption about the relative sizes of both inputs, if one is smaller, the loop simply keeps copying the rest of the bytes from the other.

We take care to ask for the remaining length for both arguments and also to check for *total_length* when handling the second argument, since it can be exhausted when copying one byte from the first argument.

After the loop, we calculate the length in the *paramdsc* as the difference between the variable used to fill in the result and the starting address of the result. This gives the difference counted in bytes, exactly what we want.

In C/C++, the difference between two pointers is a signed quantity known as ptrdiff_t, but here, we cast it to USHORT, because *vary_length* has this type and because we ensured that *total_len* would be always smaller than MAXROW (itself, in turn, being smaller than the upper limit of USHORT).

Finally, we have to fill our descriptor with meaningful data. The type is VARCHAR, the length is the same as *vary_length* plus the two bytes that *vary_length* itself uses (VARCHAR_PREFIX). The flags are reset. We return our result and the rest is the engine's task.

Now, our Pascal version. We initialize our result immediately to nil in case at least one of the input parameters is SQL NULL or is not of one of the three string types. Next, we call our previous function *get_string* to try to get a pointer to unsigned char that represents the beginning of the data in the input descriptors.

Our *total_len* is of a type enough to perform the sum of the two lengths and determine if we are above the maximum row size in the engine, in case we need to truncate the output. We take into account that we will be returning a VARCHAR inside a descriptor, so we need to reserve additional space with the VARCHAR_PREFIX constant (currently being two bytes).

```
// This is for arbitrary input, but only will accept VARCHAR or CSTRING.
// Finally we handle CHAR, too.
// If either or both input arguments are NULL, NULL is returned; same if we
// don't get the desired input data types.
// FREE_IT is used for the returned data and the descriptor itself.
function p_intersperse(p1: Pparamdsc; p2: Pparamdsc): Pparamdsc;
var
   p1_len, p2_len: USHORT;
   p1_string, p2_string, output_buf: PUchar;
   total_len: Integer;
   v: Pparamvary;
   d: Pparamdsc;
begin
   Result := nil;
```

```
p1_len := 0;
p1_string := get_string(p1, p1_len);
if p1_string = nil then
   Exit;

p2_len := 0;
p2_string := get_string(p2, p2_len);
if p2_string = nil then
   Exit;

total_len := p1_len;
Inc(total_len, p2_len); // sum made in two steps
                        // to avoid wrapping before assignment.
if total_len + VARCHAR_PREFIX > MAXROW then
   total_len := MAXROW - VARCHAR_PREFIX;

v := ib_util_malloc(total_len + VARCHAR_PREFIX);
output_buf := @v^.vary_string[0];
v^.vary_length := 0; // let's play safe if we have to exit prematurely
d := ib_util_malloc(sizeof(paramdsc));
d^.dsc_address := PUchar(v);

while total_len > 0 do
begin
   if p1_len > 0 then
   begin
      output_buf^ := p1_string^;
      Inc(output_buf);
      Inc(p1_string);
      Dec(p1_len);
      Dec(total_len)
   end;
   if (p2_len > 0) and (total_len > 0) then
```

```
      begin
         output_buf^ := p2_string^;
         Inc(output_buf);
         Inc(p2_string);
         Dec(p2_len);
         Dec(total_len);
      end;
   end;
   // ptrdiff_t->USHORT
   v^.vary_length := USHORT(PChar(output_buf) - PChar(@v^.vary_string[0]));
   d^.dsc_dtype := dtype_varying;
   d^.dsc_length := v^.vary_length + VARCHAR_PREFIX;
   d^.dsc_flags := 0;
   Result := d
end;
```

We allocate the space for the *ParamVary* with *ib_util_malloc* because our result has to be deallocated by the engine; then set the auxiliary variable *output_buf*, pointing to its *vary_string* member (the ultimate place for our data). We set the *vary_length* to zero in advance, in case we write nothing.

Now, we have to allocate the descriptor itself, that will be deallocated by the engine, too This trick will only work with FB2; previous versions will leak the memory used for the descriptor in each call.

Next, we work the reverse of the logic to get a VARCHAR from a *Paramdsc*: we build the VARCHAR, making the address of the descriptor the *Paramvary* itself, not its data member.

We start a loop, copying one byte from each input string into the output and so on. The loop is bigger than its C++ counterpart only due to the lack of Pascal syntax allowing increment and decrement operators in the same statement. We add the check for *total_len* being still bigger than zero before copying from *p2_string*, because our copy from *p1_string* may have exhausted the output capacity.

Finally, we adjust the length of the output, keeping things coherent: *output_buf* started pointing to v's *vary_string* and advanced in the loop, so the difference between the two is the length in bytes. We cast to PChar to allow the compilation to succeed in Pascal.

We tell the engine that the descriptor contains a VARCHAR and the descriptor's length is, in turn, the same as the the length of *Paramvary* plus the *vary_length* size, represented by VARCHAR_PREFIX. We clear the descriptor's flags and assign the descriptor to the function's result.

Functions that work with arrays

Arrays are a subject for polemic in a RDBMS. On one side, they allow misuse because they are not normalized data. On another. for small repetitive data of known fixed length (number of elements in known in advance), they may be easier than handle than several rows for programmers using a traditional programming language.

In Firebird, most of the work with arrays is done through the API. The only array support built-in for queries written in SQL is the ability to treat specific array elements as if they were fields. PSQL has no support to create or deal with arrays, above what a single query can do. Particularly annoying is the lack of support to update even single elements from SQL.

Firebird keeps arrays in an internal format and loads them in memory with a layout defined by a structure known as **internal array descriptor**.

UDF's are not different from SQL, in the sense of having special facilities to work with arrays. Granted, a UDF could read an array as a blob (arrays are defined on top of stream type arrays) and copy the logic to decode and load an array from the engine's internals, but that is a place most developers would not want to go.

However, the engine provides a facility, never advertised at SQL level but available through proprietary language, that allows a UDF to request that the engine convert the internal format to a simplified layout determined by a structure known as a **scalar array descriptor**. This structure could only be an input parameter to the UDF, since there is no support for UDF's to return arrays to the engine.

All that said, for some basic operations with matrices, it may be useful to know how to read arrays.

In Firebird v2, SQL DDL has been enhanced so a UDF can be declared to receive a parameter by scalar array. No enhancement was done to the internal code: it lay dormant for 10 years just because it was not exposed, unless you knew GDML. The only change is the ability to declare this mechanism without resorting to changing values in system tables directly.

The example UDF exposed here does not aspire to be a milestone, it provides information to foster more experimentation. We decided to show an example that works with the SQL *int*, a 32-bit signed integer. Since we are reading SLONGs and putting them in a string, we created a helper routine that will write the SLONG as a string, but reversed. The caller is responsible for providing a buffer of enough size (defined by the constant SLONG2TEXT) and for reversing the string for display. We did that because it's easier to decode the number in the string in reverse order and the logic of the caller fit naturally with that behavior.

```
// Do not publish this helper.
// Given an int32 and an output buffer of SLONG2TEXT size, converts the number
// to a string representation, reversed. It does not put the terminator.
// Probably you have found in many programming textbooks much better versions.
int reversed_string_from_int32(SLONG n, UCHAR* const out)
{
   // Test, only done to check negative and bigger numbers.
   //if (n % 2)
   //   n *= -50;
   //else
   //   n *= 33;

   UCHAR* output = out;
   const bool neg = n < 0;
   if (!n)
       *output++ = '0';

   // We handle negative numbers in a clumsy way, repeating the loop.
   // We cannot do -n because positive range is
      // one less than negative range
   // and we did not want to switch to int64.
   while (n > 0)
   {
       *output++ = n % 10 + '0';
       n /= 10;
   }
   while (n < 0)
   {
       *output++ = -(n % 10) + '0';
       n /= 10;
   }

   if (neg)
       *output++ = '-';

   return output - out; // ptrdiff_t handled as int
}
```

First, we assign a pointer to fill the output and a boolean to indicate whether the original number is negative. As a special case, when the quantity is zero, we write the character zero (ASCII(32)) in the output to avoid printing a blank string.

When we receive a negative number, we cannot reverse the sign and carry on because, in the specific case of the smallest value (the most negative), it would fail due to the negative range for numbers being one more than the positive range in absolute value. Instead of working with int64 to ensure we will have room for any number, we simply do two loops, one for positive and one for negative. They exclude one to another naturally. Since we are writing the number in reverse order, we assign the negative sign (if needed) at the end.

Finally, we calculate the number of bytes written as the difference of the pointer that filled the output (*out*) and the output itself, that is, the starting value for *out*.

Our Pascal version is a direct mapping of the preceding C++ function:

```
// Do not publish this helper.
// Given an int32 and an output buffer of SLONG2TEXT size, converts the number
// to a string representation, reversed. It doesn't put the terminator.
// Probably you have found in many programming textbooks much better versions.
function reversed_string_from_int32(n: SLONG; outstr: PUChar): Integer;
var
   output: PUchar;
   neg: Boolean;
begin
   // Test, only done to check negative and bigger numbers.
   //if n mod 2 <> 0 then
   //   n := n* -50
   //else
   //   n := n* 33;

   output := outstr;
   neg := n < 0;
   if n = 0 then
   begin
      output^ := Ord('0');
      Inc(output);
   end;
```

```
    // We handle negative numbers in a clumsy way, repeating the loop.
    // We cannot do -n because positive range is one less than negative range
    // and we didn't want to switch to int64.
    while n > 0 do
    begin
       output^ := n mod 10 + Ord('0');
       Inc(output);
       n := n div 10;
    end;
    while n < 0 do
    begin
       output^ := -(n mod 10) + Ord('0');
       Inc(output);
       n := n div 10;
    end;

    if neg then
    begin
       output^ := Ord('-');
       Inc(output);
    end;

    Result := PChar(output) - PChar(outstr) // ptrdiff_t handled as int
 end;
```

The only differences arise because we deal with unsigned characters: we can use them like numbers but, when interacting with characters singly, we have to convert them to numbers using *Ord()*.

Remember, in the negative case, *-(n mod 10)* will give the correct result whereas *(-n) mod 10* can overflow if the most negative value in the range is received.

Finally, our result is again the difference in bytes (forced to PChar) between the temporary variable and the buffer that was its initial value.

```
 declare external function p_array2text
 int by scalar_array,
 varchar(100) by descriptor
 returns parameter 2
 entry_point 'p_array2text' module_name 'phoenix';
```

Finally, we get to our function that reads an array. This is an idea that resembles the blob sampling but, here, all elements are shown, separated by commas.

In our declaration, we request an array of integers and the engine will obey that. If the array passed (typically, a column of a table) does not have integers, the engine tries to perform the conversion of each element. If the conversion cannot be done, the engine will throw an error.

However, we decided to play safe in case we change the UDF declaration without updating the code. We return NULL if:

- The number of dimensions is zero (the engine got a NULL field and converted it to an empty array, similarly to NULL values being converted to empty blobs when blobcallback is used.
- The type of the array's elements is unknown (probably the first condition is enough because both will happen simultaneously if the array is empty).
- The array's address is NULL (the same precaution we took with the address member of *paramdsc*).
- The array's flags indicate that we got a NULL field.
- The type of the array elements is not SLONG. This cannot happen unless we change the UDF declaration.

However, we can't simply return the null pointer here. Indeed, the type of this function is **void**. The reason is that we told the engine it had to free the descriptor and the varying structures we passed to it. However, in this case, we asked for a VARCHAR(100) passed by descriptor to be created for us and passed to the function as an output parameter. Therefore, to signal NULL, we must activate the NULL flag in the *flags* data member of the descriptor.

Now we have a tricky piece of logic that can be handled in several ways, some no doubt better. We have to calculate how much room we have for output. In previous cases, receiving a descriptor as the output parameter, you could discover the maximum length of the output that was allocated by the engine dynamically. Thus, even if you changed the declaration, the code would not need to change.

In the case at hand, we have asked for a VARCHAR(100) to be created. The varying structure (*paramvary*) will have its length initialized to zero but the descriptor's length will be the total allocated space, including the VARCHAR_PREFIX, as expected.

First, *output* will be the variable we use to fill and traverse our result descriptor's data member. We set an indicator to zero and branch according to the three string field subtypes. We do that to be able to experiment by changing to declaration to use an ouput parameter by descriptor of type CHAR, VARCHAR or CSTRING (we currently declare VARCHAR) without having to recompile the UDF:

- If we get a CHAR, we do nothing (no case for it). The descriptor's address is all we need to write the result.
- If we get a VARCHAR (the case we declared), we remember that the descriptor's address data member is actually pointing to a *paramvary*, so we set the indicator to 2 (VARCHAR_PREFIX, the size of paramvary's *vary_length*, a 16-bit signed quantity) and advance the output pointer by that number of bytes. We canott write over the length, but after it; and it is filled afterwards. That is similar to casting *pout->dsc_address* to *paramvary* and assigning output to *vary_string*.
- If we get a CSTRING, we set our indicator to 1, for the null terminator.

Writing past the allocated output area will trash or shutdown the engine, so it is imperative to avoid that. We calculate the final position, a constant pointer named *end*. This will point one byte past the last position available for us to write data.

That position, in turn, is the output pointer advanced (in bytes) by the length of the data provided by the descriptor minus our calculated indicator. Remember, for VARCHAR we had to skip the length in *paramvary* and for CSTRING we have a position reserved to write the null terminator finally.

Verification of the logic is left to the reader. Hint: remember to look at the end of the function to see how the parts fit.

Since we are here for a demonstration and not for an award for cleverest or most complex logic, we chose to do two loops. In the first loop, we calculate the total number of elements that the array contains.

Notice that Firebird, like programming languages, allows different dimensions of the array to have different numbers of elements. Therefore, the number of dimensions mutiplied by the number of elements in the first dimension is not the answer. We have to multiply the number of elements in all dimensions.

But Firebird goes beyond that and allows a dimension to be declared either by size or by range.

In the first case, it will convert the declaration to a range with a default starting position.

In the second case, the range is a starting and an ending index for the elements of the dimension. The indices can be negative, positive or zero, provided that the starting index is lower than or equal to the ending index.

Refer to the definition of *scalar array descriptor* and the child *sad_repeat* structure to understand the layout (we didn't say the *nested sad_repeat* because in Pascal does not support a syntax for nesting declarations in this manner).

Some examples to clarify

Let's assume these field declarations are wrapped in a CREATE TABLE statement:

- v int[3] is an array with one dimension and three elements. The range default to be 1 up to 3.
- v int[3, 4] is an array with two dimensions. The first dimension has tree elements (1 to 3) and the second, four elements (1 to 4).
- v varchar(10)[-1:5] is an array with one dimension and seven elements. The range effectively starts at -1 and goes up to 5.
- v timestamp[2, 0:1, -10:-9] is an array with three dimensions and two elements in each dimension. The implicit range for the first dimension is 1 to 2, the explicit range for the second is 0 to 1 and the explicit range for the third is -10 to -9.

Therefore, to get the number of elements in a dimension, we need the upper index minus the lower index plus one. Our code sets the element counter to 1 (we will be multiplying elements in each dimension so zero as initializer would be a typical newbie error) and goes to review each dimension. We like references and used them to avoid copying the *sad_repeat* structure

```
        const scalar_array_desc::sad_repeat& elem = input->sad_rpt[loop];
        total_elems *= (elem.sad_upper - elem.sad_lower + 1);
  but people that prefer pointers could have used equally
        const scalar_array_desc::sad_repeat* elem = &input->sad_rpt[loop];
        total_elems *= (elem->sad_upper - elem->sad_lower + 1);
```

After our multiplication, we make a simple check for overflows. If the sign of our multiplicand became negative, then we had an overflow (remember, in theory, the number of dimensions and the range inside a dimension are all 32-bit signed integers, so we could overflow).

> Granted, the most accurate method to know if an overflow happened is to check the processor's flags, but we don't want to go to that level. After all, using such a huge array with an example wouldn't be practical. We could use an int64 multiplicand if we wanted to handle bigger array declarations.

If total_elems became negative, we reset it to zero before reporting an error. This is because C and C++ programmers tend to be minimalist: testing a value means implicitly it is different from zero and can be used as a boolean. As the comment in the code says

"avoid problems in the while(total_elems--) below."

We expect to be able to report the error in the output if it is too small (due to a declaration error, for example). Our naive condition while(total_elems--)

assumes a well behaved *total_elems,* equal to or bigger than zero. Unless we reset the value to zero, execution would enter a big loop through all the negative range until the space for the result were exhausted or an error occurred.

To report the error, we copy the string ERROR (including the null terminator implicitly) to a local buffer and to the *Result* without the null terminator. Since we set *total_elem*s to zero, the second loop is not executed: we break the loop to continue execution in the last part of the function.

We return NULL in the unlikely event that there is not even enough room even for five bytes (perhaps the declaration of the *Result* was too little and, under normal conditions, maybe even one array element won't fit). Once again, since we have an output parameter and we are dealing with a descriptor, we do this by activating the NULL flag.

```
// This function takes an array (converted to simplified form by the engine)
//and fills a string with it. NULL is not signaled directly, but since arrays
// with zero dimensions cannot be created, we'll assume that it may be used
// as another way to tell us about NULL. We only handle integer arrays.
void p_array2text(const scalar_array_desc* const input, paramdsc* const pout)
{
   if (!input->sad_dimensions || input->sad_desc.dsc_dtype == dtype_unknown
                              || !input->sad_desc.dsc_address
                              || (input->sad_desc.dsc_flags & DSC_null)
                              ||input->sad_desc.dsc_dtype != dtype_long)
   {
       pout->dsc_flags |= DSC_null;
       return;
   }

   UCHAR* output = pout->dsc_address;
   int indicator = 0;
   if (pout->dsc_dtype == dtype_varying)
   {
       indicator = VARCHAR_PREFIX;
              // We'll fill vary_length at the end, make space for it
       output += VARCHAR_PREFIX;
   }
```

```
else if (pout->dsc_dtype == dtype_cstring)
     indicator = 1;

 const UCHAR* const end = output + pout->dsc_length - indicator;

 // We'll handle elements sequentially. Otherwise, we need a fixed number
 // of nested loops (static) or a recursive helper routine (dynamic).
 int total_elems = 1;
 for (int loop = 0; loop < input->sad_dimensions; ++loop)
 {
     const scalar_array_desc::sad_repeat& elem =
                 input->sad_rpt[loop];
     total_elems *= (elem.sad_upper - elem.sad_lower + 1);
     if (total_elems < 0)
     {
        total_elems = 0; // avoid problems in the
                                   // while(total_elems--) below.
        char msg[] = "ERROR";
        // We did overflow the multiplier. Better go out.
        if (output + sizeof(msg) - 1 < end)
        {
           for (const char* p = msg; *p; ++p)
                                         // We do not copy the terminator
               *output++ = *p;

           break; // exit from the for()
        }
        else
        {
           // Too little output space, cannot put
                                // a message; let's return NULL.
           pout->dsc_flags |= DSC_null;
           return;
        }
     }
 }
```

```
   const SLONG* data = reinterpret_cast<const SLONG*>(input-
>sad_desc.dsc_address);
   UCHAR temp[SLONG2TEXT];
   while (total_elems--)
   {
       int len = reversed_string_from_int32(*data++, temp);
       if (output + len + 1 < end)
       {
          // We got the string reversed.
                       // Now, fix it while we copy it.
          while (len--)
             *output++ = temp[len];

          *output++ = ';';
       }
       else
          break;
   }

   // Adjust the length.
   pout->dsc_length = output - pout->dsc_address; // [***]
   switch (pout->dsc_dtype)
   {
   case dtype_text:
       // Nothing else to do
       break;
   case dtype_varying:
       {
          // Space for paramvary->vary_length
                       // is already counted implicitly
          // in [***] and therefore
                       // we do not increment pout->dsc_length here.
          // Instead, we should acknowledge the real length
                       // in paramvary.
          paramvary* const v =
                              reinterpret_cast<paramvary*>(pout->dsc_address);
          v->vary_length = pout->dsc_length - VARCHAR_PREFIX;
```

```
        }
        break;
    case dtype_cstring:
        *output = 0; // Put the null terminator
        pout->dsc_length += 1; // Space for the null terminator
        break;
    default: // Did someone declare an output param of type <> string?
        pout->dsc_length = 0;
        pout->dsc_flags |=  DSC_null;
        break;
    }
}
```

Assuming all went well in our first loop, now we have our variable *total_elems* with the total number of elements throughout all dimensions in the array.

To make the code more readable, we define an auxiliary variable pointing to the *address* member of the descriptor inside the input scalar array descriptor. We make it a temporary of size SLONG2TEXT, to make it big enough to hold the maximum possible result from the function: a 32-bit signed number written as a string but in reverse order.

We enter the second loop, cycling as many times as there are elements found.

In turn, execution puts the elements, one after another, in the *sad_desc.dsc_address* pointer. Since we have hard-coded our function to process SLONG inputs (equivalent to the SQL integer type), we cast the input as pointer to SLONG, to make it easier to traverse. We only have to increment the pointer by one; the rest is done by the compiler.

Once we have the number converted to string, we also have the count of bytes it needed. We check whether the current output position, plus that length, plus one for the comma is still smaller than the *end* marker we have pointing one position beyond our available space to write.

- If we have space, we copy the number (in reverse order, to compensate for the conversion function) and append the comma.
- Otherwise we stop the loop since we cannot write any more bytes.

After the loop, we have to adjust our structures. The length in the result descriptor is the difference between the current position of the writer and the original position where it started from. This is a difference in bytes. Then we branch to make data type specific calculations:

❑ For CHAR, we have nothing else to do.

- For VARCHAR, we know from previous examples that the length of the varying should be the nett length of the data, i.e., the descriptor's length (already set) minus the VARCHAR_PREFIX.
- For type CSTRING we write the terminator and increment the length by one. Here lies the reason for the indicator being 1 for this type at the beginning. The C-style string is not a native type in SQL and the engine counts the null terminator as part of the data. CSTRINGs are transient containers. They are never written to the database. Their use is solely to communicate with a UDF if the function declaration requests it.

Unlike input parameters received by descriptor, which can contain any data the user sends, ouput parameters by descriptor are created by the engine to give the UDF space to write to. The declaration therefore matches exactly what the engine creates. We did no check at the beginning to verify that we received the right type of output. Only a wrong DDL statement could cause that problem.

If the output type is none of the three string types, we set the length to zero and set the descriptor's flags to activate the NULL indicator.

Acute eyes may notice that the case for CHAR is handled in a suspicious way: the CHAR type demands that strings of this type be padded with null, since the full declared length is used. In our case, instead of adjusting *pout->dsc_length* for CHAR and letting the engine try to handle that, we should have left that length untouched, instead filling any remaining unfilled output space with blank characters. This is left as an excercise.

Some notes

- You can have up to 16 dimensions.
- The underlying support for arrays is blobs. Since blobs can grow up to 2GB (despite the fact the maximum chunk you can read at a time is 64K-1), arrays can be really large. Whether such size makes sense from the perspectives of good design or efficiency is another question.
- Arrays cannot be nested.. In other words, you cannot define arrays of arrays. Consider ing you have multidimensional arrays, that is not a hardship.
- Be prepared to deal with the API, or to work through a higher-level db connectivity package to save the time and effort you would otherwise spend trying to get arrays to work for you. Your only option in SQL statements is to read each element separately.
- You can't use blobs as array elements.
- It is worth noting that all our UDFs work on the assumption that the declarations for string types involve no character set attributes other than charset NONE or ASCII. In order to handle multi-byte character sets (MBCS) we would have to review the logic.

- The *varying* structure and the descriptor always count bytes. If someone were going to write an UDF that handles different charsets automatically, the only way would be to request the input parameters by descriptor, so the charset could be determined at run-time. With MBCS, though, some of our assumptions like finding the null terminator byte-by-byte or padding with blanks could be completely wrong.

The Pascal version

Anticipating that the Pascal code would become quite complex, before attempting the Pascal version we created a helper function to mimic the C++ post-decrement (decrement after the statement where the variable appears) and avoid mistakes in the conversion. The function which, of course, we could have used in the earlier functions, simply sets the result to the received value, then decrements the received value:

```
// Do not publish this helper.
// This makes C++ expressions like while (var--) straightforward.
function PostDec(var i: Integer): Integer;
begin
   Result := i;
   Dec(i)
end;
```

Now, to the Pascal version of our last function. We chose to use *msg* as an array to make the code similar to the C++ version. We have SLONG2TEXT to represent the largest string representation of an int32. We then test for dimensions being zero, the type of the array's elements being unknown, the data (address) being nil or the flags indicating SQL NULL to return NULL.

In this case, since the descriptor and its space are created by the engine, thanks to the UDF declaration, we have a procedure, not a function. Thus, we cannot return nil to indicate we are returning SQL NULL. Instead, we set the output descriptor's flags to activate the null flag. Also, having logic to handle only the SQL integer type, we signal SQL NULL if the input descriptor's type is not *dtype_long*. Some conditional messages in dialog boxes are again used to let us know what is going on inside, if necessary.

Our *output* variable points to the data set up by the engine for placing the result. Our *indicator* variable is used to subtract the length of auxiliary information from the space the engine has provided for output. That is, reserve VARCHAR_PREFIX (two bytes) for VARCHAR output and reserve one byte for the null terminator in CSTRING output.

The function can discover the size in bytes from the output descriptor (*pout*) provided by the engine; so the declaration (set at 100 currently) can be changed without recompiling the function, and the returning type can be changed, as well, from VARCHAR to CHAR or to CSTRING. We can handle any of them, provided the mechanism stays as BY DESCRIPTOR.

Finally, because we must not overflow the memory provided by the engine, the *endoutput* variable signals the end of the buffer by a simple formula: first, the beginning of the output buffer, then increment it by the number of bytes provided in our output descriptor minus the aforementioned *indicator* variable.

Since the number of elements in the array is the product of the elements in each array dimension, we initialize our *total_elems* to 1, ready for the loop that reads the dimensions. The API is C-oriented, so we must remember our cycle is from zero to the number of dimensions minus 1. We can use the loop variable directly (it's easy to forget to subtract one inside the cycle in each usage).

For each dimension, the array *sad_rpt* in the special array descriptor has a position with an element that carries the lower and upper indices of the array. Hence, the difference between the two, multiplied by the previous dimension's element count, gives us the number of elements so far in the dimensions already visited.

- If our multiplier (*total_elems*) goes to negative, we assume an overflow in the multiplication caused the problem. If there is enough space, we try to put the error message "ERROR"; otherwise, we signal that we are returning SQL NULL.
- If we manage to accommodate the error message, we break the loop to pass execution to the final part of the function to calculate the correct output length and we will show truncated output.

If we could only set the null indicator, meaning SQL NULL, we simply exit the function, because the engine will not attempt to read the data.

The way we define and copy the string "ERROR" is unnatural to Pascal programmers, but if we use the StrPCopy function, it will copy the null terminator, also, converting the logic in

```
const
        msg2: string[5] = 'ERROR';
```

and later

```
        if PChar(output) + length(msg2) + 1 < endoutput then
        begin
           StrPCopy(PChar(output), msg2);
           Inc(output, length(msg2));
           break; // exit from the for()
        end
```

because we have to consider one byte for the terminator that we won't use. Again, we tried to stick to extreme C++ resemblance to make it clear that both versions do the same. The Pascal version could be customized to suit the style of the language better:

```
// This function takes an array (converted to simplified form by the engine)
//and fills a string with it. NULL is not signaled directly, but since arrays
// with zero dimensions cannot be created, we'll assume that it may be used
// as another way to tell us about NULL. We only handle integer arrays.
procedure p_array2text(input: Pscalar_array_desc; pout: Pparamdsc);
const
   msg: array[0..5] of char = 'ERROR'#0;
var
   temp: array[0..SLONG2TEXT - 1] of UCHAR;
   output, endoutput: PUchar;
   indicator, total_elems, loop, len: Integer;
   data: ^SLONG;
   elem: ^sad_repeat;
   p: PChar;
   v: Pparamvary;
begin
   if ((input^.sad_dimensions = 0)
    or (input^.sad_desc.dsc_dtype = dtype_unknown)
    or (input^.sad_desc.dsc_address = nil)
    or ((input^.sad_desc.dsc_flags and DSC_null) > 0)
    or (input^.sad_desc.dsc_dtype <> dtype_long)) then
   begin
       pout^.dsc_flags := pout^.dsc_flags or DSC_null;
       Exit;
   end;

   output := pout^.dsc_address;
   indicator := 0;
   if pout^.dsc_dtype = dtype_varying then
   begin
       indicator := VARCHAR_PREFIX;
       // We'll fill vary_length at the end, make space for it
       Inc(output, VARCHAR_PREFIX)
   end
   else if (pout^.dsc_dtype = dtype_cstring) then
       indicator := 1;
   endoutput := output;
   Inc(endoutput, Integer(pout^.dsc_length) - indicator);
   // We'll handle elements sequentially. Otherwise, we need a fixed number of
   // nested loops (static) or a recursive helper routine (dynamic).
   total_elems := 1;
   for loop := 0 to input^.sad_dimensions - 1 do
   begin
       elem := @(input^.sad_rpt[loop]);
       total_elems := total_elems * (elem^.sad_upper - elem^.sad_lower + 1);
       if total_elems < 0 then
       begin
          // Avoid problems in the while PostDec(total_elems) > 0 below.
          total_elems := 0;
          // We did overflow the multiplier. Better go out.
{$IFDEF GUI_MSG}
          ShowMessage(Format('array2text, sizeof=%d, %.30s', [sizeof(msg),
msg]));
{$ENDIF}
          if PChar(output) + sizeof(msg) - 1 < endoutput then
          begin
             p := msg;
             while p^ <> #0 do // We don't copy the terminator
             begin
                 output^ := Ord(p^);
                 Inc(output);
                 Inc(p);
             end;

             break; // exit from the for()
          end
          else
          begin
             // Too few output space, can't put a message; let's return NULL.
             pout^.dsc_flags := pout^.dsc_flags or DSC_null;
             Exit;
          end;
       end;
   end;
{$IFDEF GUI_MSG}
   ShowMessage(Format('array2text, total_elems=%d', [total_elems]));
{$ENDIF}

   data := Pointer(input^.sad_desc.dsc_address);
   while PostDec(total_elems) > 0 do
   begin
       len := reversed_string_from_int32(data^, @temp[0]);
       Inc(data);
       if PChar(output) + len + 1 < endoutput then
       begin
          // We got the string reversed. Now, fix it while we copy it.
          while PostDec(len) > 0 do
          begin
             output^ := temp[len];
             Inc(output);
          end;

          output^ := Ord(';');
          Inc(output);
       end
       else
          break;
   end;

   // Adjust the length.
   pout^.dsc_length := PChar(output) - PChar(pout^.dsc_address); // [***]
   case pout^.dsc_dtype of
       dtype_text:
          ; // Nothing else to do
       dtype_varying:
       begin
          // Space for paramvary^.vary_length is already counted implicitly
          // in [***] and therefore we don't increment pout^.dsc_length here.
          // Instead, we should acknowledge the real length in paramvary.
```

```
          v := Pparamvary(pout^.dsc_address);
          v^.vary_length := pout^.dsc_length - VARCHAR_PREFIX
       end;
       dtype_cstring:
       begin
          output^ := 0; // Put the null terminator
          Inc(pout^.dsc_length); // Space for the null terminator
       end;
       else begin // Did someone declare an output param of type <> string?
          pout^.dsc_length := 0;
          pout^.dsc_flags := pout^.dsc_flags or DSC_null;
       end;
    end;
 end;
```

Finally we are ready to copy as many elements as we can. Remember, if there was an error, we set *total_elems* to zero to avoid copying garbage or overflowing the buffer provided by the engine. Our *data* variable points to the input information, stored in the *data* member of a Paramdsc (*sad_desc*) inside the scalar array descriptor. Since we know we are receiving SLONGs, we already made *data* a pointer to SLONG. We use the Pointer function to go from one pointer type to another, since Pascal has the strange property of allowing a typed pointer to receive the value of an untyped pointer.

For as many elements as we find, we convert each element to its string representation using our helper function *reversed_string_from_int32*, previously described.

When we increment the pointer, *data*, by one, the compiler automatically increases it by the number of bytes of the type pointed to (SLONG in this case, with four bytes), making it ready in the next element.

- If the output we got in the *temp* variable, plus the comma, fit in the remaining output space, we copy it byte by byte and then copy the comma that separates each element.
- Otherwise, we break the loop. since we are unable copy more.

Notice how we used our helper function *PostDec* twice, to mimic C++ behavior.

Finally, we set the length of the output descriptor. In reality, we are adjusting what the engine gave, anticipating that the result will be equal to or smaller than the original value. If that is not the case, the logic is flawed somewhere; but we don't check for that contrary conditon.

As before, the difference is measured in bytes, so the PUChar have to be converted to PChar to be subtracted.

- For CHAR, we are done.
- For VARCHAR, we need to set the paramvary's length to the same length minus the VARCHAR_PREFIX.
- For CSTRING, we include the null terminator.
- In case the function declaration was wrong and the output descriptor was not either CHAR, VARCHAR or CSTRING, we detect it here and return NULL.

Remember, all our examples imply the assumption that we were working with ASCII or at least character sets that have single-byte characters.

9—The output

First we show the commands that were executed to create the output. We assume the database "phoenix.fdb" was already created and the phoenix.sql script was run on it to declare the UDFs. The commands were copied here to show how those functions can be used in an isql session:

```
isql phoenix.fdb
SQL>
select p_sumchar1('xyz') from rdb$database;
select p_sumchar2('xyz') from rdb$database;
select p_sumchar3('xyz') from rdb$database;
select p_sumchar1(NULL) from rdb$database;
select p_sumchar2(NULL) from rdb$database;
select p_sumchar3(NULL) from rdb$database;

select p_lastchar1('firebird') from rdb$database;
select p_lastchar1('firebirdfirebirdfirebirdfirebird') from rdb$database;
select p_lastchar1('firebirdfirebirdfirebirdfirebi') from rdb$database;
select p_lastchar2('firebird') from rdb$database;
select p_lastchar3('firebird') from rdb$database;
select p_lastchar1(NULL) from rdb$database;
select p_lastchar2(NULL) from rdb$database;
select p_lastchar3(NULL) from rdb$database;

select p_reverse1('platypus') from rdb$database;
select p_reverse2('platypus') from rdb$database;
select p_reverse3('platypus') from rdb$database;
select p_reverse1(NULL) from rdb$database;
select p_reverse2(NULL) from rdb$database;
select p_reverse3(NULL) from rdb$database;

select p_generate_blob(5, 10) from rdb$database;
set blob all;
select p_generate_blob(5, 10) from rdb$database;
select p_defragment_blob(p_generate_blob(5, 10), 0) from rdb$database;
select p_defragment_blob(p_generate_blob(5, 10), 30) from rdb$database;
select p_defragment_blob(NULL, 0) from rdb$database;
select p_sample_blob(p_generate_blob(5, 10), 3) from rdb$database;
select p_sample_blob(p_generate_blob(5, 10), 5) from rdb$database;
select p_sample_blob(p_generate_blob(1, 10), 5) from rdb$database;
select p_sample_blob(p_defragment_blob(p_generate_blob(5, 10), 30), 5) from
rdb$database;
select p_sample_blob(NULL, 1) from rdb$database;

select p_intersperse('haydn', 'beethoven') from rdb$database;
select p_intersperse(cast('haydn' as varchar(10)), cast('beethoven' as
varchar(10))) from rdb$database;
select p_intersperse('haydn', 'beethoven') from rdb$database;
select p_reverse2(p_intersperse('haydn', 'beethoven')) from rdb$database;
select p_intersperse('', '') from rdb$database;
select p_intersperse('', '') || '*' from rdb$database;
exit;
```

This small database was filled with a tool capable of filling arrays instead of having to use the API for this example. Many thanks to Geoff Worboys:

```
isql array.fdb
show tables;
show tables array_table;
select array_field from array_table;
set blob 0;
select array_field from array_table;
select array_field[1, 1], array_field[2, 2], array_field[3, 3] from
array_table;
select p_array2text(array_field) from array_table;
select p_array2text(NULL) from rdb$database;
exit;
```

As expected, the output is the same for both the C++ and Pascal versions, as expected. Here's the captured output from the preceding commands in both databases:

```
F:\fb2dev\fbbuild\firebird2\temp\debug\firebird\bin>isql phoenix.fdb
Database:  phoenix.fdb


SQL> select p_sumchar1('xyz') from rdb$database;


  P_SUMCHAR1
============
         363


SQL> select p_sumchar2('xyz') from rdb$database;


  P_SUMCHAR2
============
         363


SQL> select p_sumchar3('xyz') from rdb$database;


  P_SUMCHAR3
============
         363


SQL> select p_sumchar1(NULL) from rdb$database;


  P_SUMCHAR1
============
           0


SQL> select p_sumchar2(NULL) from rdb$database;


  P_SUMCHAR2
============
           0
```

```
SQL> select p_sumchar3(NULL) from rdb$database;

  P_SUMCHAR3
============
           0

SQL> select p_lastchar1('firebird') from rdb$database;

P_LASTCHAR1
===========



SQL> select p_lastchar1('firebirdfirebirdfirebirdfirebird') from rdb$database;

P_LASTCHAR1
===========
Statement failed, SQLCODE = -802
arithmetic exception, numeric overflow, or string truncation

SQL> select p_lastchar1('firebirdfirebirdfirebirdfirebi') from rdb$database;

P_LASTCHAR1
===========
i

SQL> select p_lastchar2('firebird') from rdb$database;

P_LASTCHAR2
===========
d

SQL> select p_lastchar3('firebird') from rdb$database;

P_LASTCHAR3
===========
d
```

```
SQL> select p_lastchar1(NULL) from rdb$database;

P_LASTCHAR1
===========
<null>

SQL> select p_lastchar2(NULL) from rdb$database;

P_LASTCHAR2
===========
<null>

SQL> select p_lastchar3(NULL) from rdb$database;

P_LASTCHAR3
===========
<null>

SQL> select p_reverse1('platypus') from rdb$database;

P_REVERSE1
===============================
supytalp

SQL> select p_reverse2('platypus') from rdb$database;

P_REVERSE2
===============================
supytalp

SQL> select p_reverse3('platypus') from rdb$database;

P_REVERSE3
===============================
supytalp

SQL> select p_reverse1(NULL) from rdb$database;
```

```
P_REVERSE1
==============================



SQL> select p_reverse2(NULL) from rdb$database;

P_REVERSE2
==============================



SQL> select p_reverse3(NULL) from rdb$database;

P_REVERSE3
==============================



SQL> select p_generate_blob(5, 10) from rdb$database;

  P_GENERATE_BLOB
=================
              0:1
==============================================================================
P_GENERATE_BLOB:
BLOB display set to subtype 1. This BLOB: subtype = 0
==============================================================================

SQL> set blob all;
SQL> select p_generate_blob(5, 10) from rdb$database;

  P_GENERATE_BLOB
=================
              0:2
==============================================================================
P_GENERATE_BLOB:
```

```
0123401234501234560123456701234567801234567890123456789001234567890101234567890
01201234567890123
===============================================================================

SQL> select p_defragment_blob(p_generate_blob(5, 10), 0) from rdb$database;

P_DEFRAGMENT_BLOB
=================
              0:6
===============================================================================
P_DEFRAGMENT_BLOB:

0123401234501234560123456701234567801234567890123456789001234567890101234567890
01201234567890123
===============================================================================

SQL> select p_defragment_blob(p_generate_blob(5, 10), 30) from rdb$database;

P_DEFRAGMENT_BLOB
=================
              0:a
===============================================================================
P_DEFRAGMENT_BLOB:

0123401234501234560123456701234567801234567890123456789001234567890101234567890
01201234567890123
===============================================================================

SQL> select p_defragment_blob(NULL, 0) from rdb$database;

P_DEFRAGMENT_BLOB
=================
           <null>

SQL> select p_sample_blob(p_generate_blob(5, 10), 3) from rdb$database;
```

```
P_SAMPLE_BLOB

===============================================================================
=
012,012,012,012,012,012,012,012,012,012,

SQL> select p_sample_blob(p_generate_blob(5, 10), 5) from rdb$database;

P_SAMPLE_BLOB

===============================================================================
=
01234,01234,01234,01234,01234,01234,01234,01234,01234,01234,

SQL> select p_sample_blob(p_generate_blob(1, 10), 5) from rdb$database;

P_SAMPLE_BLOB
===============================================================================
0,01,012,0123,01234,01234,01234,01234,01234,01234,

SQL> select p_sample_blob(p_defragment_blob(p_generate_blob(5, 10), 30), 5)
from rdb$database;

P_SAMPLE_BLOB

===============================================================================
=
01234,45678,45678,90123,

SQL> select p_sample_blob(NULL, 1) from rdb$database;

P_SAMPLE_BLOB

===============================================================================
=
<null>
```

```
SQL> select p_intersperse('haydn', 'beethoven') from rdb$database;

P_INTERSPERSE
============================================================
<null>

SQL> select p_intersperse(cast('haydn' as varchar(10)), cast('beethoven' as
varchar(10))) from rdb$database;

P_INTERSPERSE
============================================================
hbaeyedtnhoven

SQL> select p_intersperse('haydn', 'beethoven') from rdb$database;

P_INTERSPERSE
============================================================
hbaeyedtnhoven

SQL> select p_reverse2(p_intersperse('haydn', 'beethoven')) from rdb$database;

P_REVERSE2
==============================
nevohntdeyeabh

SQL> select p_intersperse('', '') from rdb$database;

P_INTERSPERSE
============================================================



SQL> select p_intersperse('', '') || '*' from rdb$database;

CONCATENATION
=============================================================
*
```

```
SQL> ^Z

F:\fb2dev\fbbuild\firebird2\temp\debug\firebird\bin>isql array.fdb
Database:  array.fdb
SQL> show tables;
       ARRAY_TABLE
SQL> show tables array_table;
ARRAY_FIELD                     ARRAY OF [1:3, 1:3]
                                INTEGER Nullable
SQL> select array_field from array_table;

      ARRAY_FIELD
=================
             80:0
           <null>

SQL> set blob 0;
SQL> select array_field from array_table;

      ARRAY_FIELD
=================
             80:0
           <null>

SQL> select array_field[1, 1], array_field[2, 2], array_field[3, 3] from
array_table;

 ARRAY_FIELD  ARRAY_FIELD  ARRAY_FIELD
============ ============ ============
           1            5            9
      <null>       <null>       <null>

SQL> select p_array2text(array_field) from array_table;

P_ARRAY2TEXT
```

```
==============================================================================
=
1;4;7;2;5;8;3;6;9;

P_ARRAY2TEXT

==============================================================================
=
-50;132;-350;66;-250;264;-150;198;-450;
```

```
SQL> select p_array2text(NULL) from rdb$database;

P_ARRAY2TEXT

================================================================================
=
<null>

SQL> ^Z
```

The sources for both Delphi and MSVC6 projects are included in separate archives, including debugging versions of the libraries (DDLs) compiled with both environments.

The C++ version contains the script with the UDF declarations.