WSDL2RPG – FAQ

FAQ How to Use Dynamic Arrays

Status of this Document

Date: 27.03.2012
Version: 1.6

Question

What do I have to do to use dynamic arrays in order to get around the 64k barrier?

Short Answer

In order to use dynamic arrays you have to set the DIM parameter to *NOMAX when generating your stub module. The *NOMAX special value is supported by WSDL2RPG v1.12 and higher.

Answer

Before I go into details you have to understand the terms “Heap Storage”, “Memory Manager” and “Dynamic Array”.

Heap Storage

A heap is a storage container that provides the memory for dynamic memory allocation. Whenever an application requests a memory block (e.g. RPG: `%allocate()`) it is allocated within a heap.

Usually storage is allocated within the default heap. But it is also possible to create a “private” heap and allocate memory blocks from there.

Allocated memory blocks must be freed and returned to the system once the memory is no longer in use (e.g. RPG: `deallocate`).

All associated memory blocks are automatically freed when a private heap is discarded (deleted). Whereas memory blocks allocated from the default heap must be freed one by one because the default heap cannot be deleted.

Memory Manager

The purpose of the WSDL2RPG memory manager is to keep track about the memory blocks that are dynamically allocated by a particular web service stub module.

It uses a private heap per stub module to fulfil incoming requests for memory allocation. The heap is associated to the web service module using a unique stub module ID. The memory manager uses a map to keep track of these key/value pairs. The key of such a map entry contains the stub module ID and the value is set to the heap identifier as returned by the “Create Heap” API.

This way it is easy to return all memory blocks of a given stub module to the system by simply discarding the heap.

Dynamic Array

A dynamic array as provided by WSDL2RPG is an array that dynamically allocates storage for its elements. This way it does not share the limitations of standard RPG arrays which are “fixed size” and “64k barrier” and it can hold up to 1.044.473 elements.

Prior to WSDL2RPG v1.15 a dynamic array could store up to 170.000 elements.

Using Dynamic Arrays

The first thing you have to do is to enable your stub modules for dynamic arrays when you create a new module. Set the ‘**DIM**’ parameter to the ‘***NOMAX**’ special value when you execute the WSDL2RPG command to let the generator know that you want to use dynamic arrays. Please do that for the stub module as well as for the test program. Sample commands:

```
WSDL2RPG
   URL('http://www.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL')
   SERVICE('CountryInfoServiceSoap' ('ListOfCountryNamesByCode'))
   SRCFILE(*LIBL/QWSDL2RPG) SRCMBR(WS0004) TYPE(*STUB) DIM(*NOMAX)


WSDL2RPG
   URL('http://www.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL')
   SERVICE('CountryInfoServiceSoap' ('ListOfCountryNamesByCode'))
   SRCFILE(*LIBL/QWSDL2RPG) SRCMBR(WS000401T) TYPE(*PGM) STUB(WS000401) DIM(*NOMAX)
```

Now, what happens? When the generator uses dynamic arrays it slightly changes the structure of array elements:

XML:

```
<xs:element
      name="tCountryCodeAndName"
      type="tns:tCountryCodeAndName"
      minOccurs="0"
      maxOccurs="unbounded"
      nillable="true" />
<xs:complexType name="tCountryCodeAndName">
  <xs:sequence>
    <xs:element name="sISOCode" type="xs:string" />
    <xs:element name="sName" type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

Standard RPG array structure:

```
D tns_RpgArrayOfTCountryCodeAndName_t...
D                 DS                  based(pDummy)
D                                     qualified
D  x                              10I 0
D  item                               likeds(tns_tCountryCodeAndName_t)
D                                     dim(DIM_A1)
```

Dynamic array structure:

```
D tns_RpgArrayOfTCountryCodeAndName_t...
D                 DS                  based(pDummy)
D                                     qualified
D  hItems                             like(wsdl_hArray_t)
 *   Type of hItems is: tns_tCountryCodeAndName_t
```

The '**x**' field is no longer needed, because the dynamic array always knows how many elements it has. Hence it is dropped for dynamic arrays.

On the other side the meaning of the '**item**' field has changed. Hence it has been renamed from '**item**' to '**hItems**' to indicate that now it is a handle of a dynamic array.

Please notice the comment right below each array:

```
Type of hItems is: tns_tCountryCodeAndName_t
```

The generator added the comment to let you know about the type of the array items.

Usually the handle of the array is used to access the array but you are also allowed to use a symbolic name.

It is highly recommended not to use symbolic names due to significant performance degrades when using a large number of arrays. Starting with v1.15 symbolic names are disabled by default.

The WSDL2RPG runtime service program exports the following procedures:

`Array_add`	Adds a given element to an array.
`Array_get`	Returns the element at the specified position in the array.
`Array_getNumE`	Returns the number of elements in the array.
`Array_getName`	Returns the name of the array.
`Array_getHandle`	Returns the handle of the array.
`Array_null`	Returns a NULL array.
`Array_isNull`	Returns cTrue if the specified array handle is null, else cFalse.

Despite of that the service program also exports the following procedures, these procedures are **not** intended to be used by the public:

`Array_new`	Produces a dynamic array.
`Array_delete`	Deletes a dynamic array.

Please use the memory manager in case you need to create or delete a dynamic array in order to associate the array to your stub module or detach the array from it:

`MemoryManager_createArray`	Produces a dynamic array.
`MemoryManager_deleteArray`	Produces a dynamic array.

Using the memory manager to create an array ensures that its memory gets freed when the memory manager is terminated.

Last but not least the generator added the next two procedures to your stub module:

```
StubModule_initializeMemoryManager
StubModule_terminateMemoryManager
```

The first one is used to initialize the memory manager for a given web service stub module. At this point a new heap is created and associated to the ID of the stub module.

The second procedure is used to terminate the memory manager. All memory blocks used by the dynamic arrays of that particular stub module are returned to the system by deleting the heap.

Keep in mind that you cannot access any array data after that point!

Linking the Stub Service Program

There is nothing special with creating the stub service program. Just use the CRTSRVPGM command as described in FAQ “How to Create a Test Program”.

Linking the Test Program

When linking the test program you need to add the following service programs to the CRTPGM command:

- BASICS1
- WSDL2RPGRT

See also “Linking the test program” in FAQ “How to Create a Test Program”.

Sending and Receiving Dynamic Arrays

Regardless of using dynamic arrays to send or receive data you have to initialize the memory manager prior calling the web service:

```
uuid = StubModule_initializeMemoryManager()
```

Note: The uuid returned by the procedure can later be used to create arrays.

Now you can create arrays, call your web service and process the response data.

Then, at the end of each and everything, do not forget to free resources and terminate the memory manager:

```
StubModule_initializeMemoryManager()
```

Receiving Dynamic Arrays

Receiving dynamic arrays is a bit easier than sending arrays because you do not need to create arrays. They are automatically produced by the generated stub module.

Please open example **WS000401** and search for “`createArray`”. You should find the following statement:

```
tns_tCountryCodeAndName.hItems =
   MemoryManager_createArray(getOperationUuid:
   %size(emptyItem): 'tCountryCodeAndName');
```

The first parameter contains the ID of the web service stub module and is used to associate the array to the **WS000401** stub module.

The second parameter is an empty array item. It is used to specify the size of an array element. A few lines above '**emptyItem**' is defined like data structure '**tns_tCountryCodeAndName_t**'. Do you remember the comment after the type definition of '**tns_RpgArrayOfTCountryCodeAndName_t**'?

```
Type of hItems is: tns_tCountryCodeAndName_t
```

The third parameter specifies the symbolic name of the array.

The symbolic name of the array can be used instead of the handle to access the array. You may find this way more comfortable than using a more or less long handle name. On the other hand using the symbolic name is significant slower than using the handle for large arrays. Last but not least the symbolic array name must be unique which sometimes is difficult to achieve.

For example use the following statement to get the number of elements in the array if you know the name of the array:

```
Array_getNumE('tCountryCodeAndName')

versus:

Array_getNumE(ListOfCountryNamesByCodeResponse...
              .ListOfCountryNamesByCodeResult...
              .tCountryCodeAndName.hItems)
```

If you do not know the name of the array just ask the array for its name like this:

```
tCountryCodeAndName =

      Array_getName(ListOfCountryNamesByCodeResponse...
                    .ListOfCountryNamesByCodeResult...
                    .tCountryCodeAndName...
                    .hItems);
```

And this is how to access a specific element of the array:

```
D curTCountryCodeAndName_A1...
D                 DS                  likeds(
D                                     tns_tCountryCodeAndName_t)
D                                     based(pX_A1)


pX_A1 = Array_get(tCountryCodeAndName: i);
sndPgmMsg('Name: ' + curTCountryCodeAndName_A1.sName);
```

This method of using a data structure based on a pointer gives "transparent" access to the element which means that you can even change the data of the element. Furthermore it is very fast because the data of the element must not be copied.

Use this method if you need a copy of the data for some reasons:

```
D curTCountryCodeAndName_A1...
D                 DS                  likeds(
D                                     tns_tCountryCodeAndName_t)
D                                     inz


Array_get(tCountryCodeAndName: i
          : %addr(curTCountryCodeAndName_A1));
sndPgmMsg('Name: ' + curTCountryCodeAndName_A1.sName);
```

Sending Dynamic Arrays

You can also use dynamic arrays to send data. In this case you first have to create an array before you can populate it:

```
D someArrayItem...
D                 DS                  likeds(
D                                     tns_someArrayItem_t)
D                                     inz


someRequestMsg.hItems =
   MemoryManager_createArray(uuid
                             : %size(someArrayItem)
                             : 'inputArray');
```

Then add items to the array as shown below:

```
clear someArrayItem;
someArrayItem.string  = 'Hello World';
someArrayItem.integer = 4711;

Array_add('inputArray'
          : %addr(someArrayItem);
```

Overview

```
      *
       *  Memory Manager UUID, used to allocate and free
       *  the memory blocks of the dynamic arrays of the
       *  SomePort_someOperation() web service.
      D uuid            S                   like(wsdl_uuid_t ) inz
       *
       *  Item of input array
      D arrayItem...
      D                 DS                  likeds(ns_arrayItem_t)
      D                                     inz
      *
       *  Item of response array
      D respItem...
      D                 DS                  likeds(ns_respItem_t)
      D                                     based(pRespItem)
       / free

          // Initialize memory manager prior using dynamic arrays
          uuid = SomeWebServiceStub_initializeMemoryManager();

          // Create and populate arrays
          req.hArray =
             MemoryManager_createArray(
                   uuid: %size(arrayItem): 'array');

          clear arrayItem;
          arrayItem.attr1 = 'aValue';
          Array_add('array': %addr(arrayItem));

          // Call the web service
          resp = SomePort_someOperation(req: errMsg);

          // Spin through array items
          arrayResp = Array_getName(resp.return.hItems);
          for i = 1 to Array_getNumE(arrayResp);
             pRespItem = Array_get(arrayResp: i)

             // Do whatever you want with respItem.*
             whatever = respItem.fooBaa;
          endfor;

          // Terminate memory manager and free memory
          SomeWebServiceStub_terminateMemoryManager();
```

Enabling Symbolic Names

It is highly recommended not to use symbolic names due to significant performance degrades when using a large number of arrays. Starting with v1.15 symbolic names are disabled by default.

If you really want to use symbolic names, you need to change the call to MemoryManager_attachService(). Set the second parameter to 'cTrue' to use symbolic names:

```
MemoryManager_attachService(uuid: cTrue); // symbolic names enabled
MemoryManager_attachService(uuid: cFalse); // symbolic names disabled
```

An application that sends an array with 20000 elements with an inner array having 3 elements required 25 minutes to handle the request when using symbolic names. After having disabled symbolic names the same request now takes 12 seconds!

Avoiding Duplicate Array Names

The following error message indicates duplicate array names:

```
Array name is not unique: inputArray
```

Usually it is the test program and not the web service stub that violates the unique array name rule. Why? Because the stub module calls `MemoryManager_createArray()` with parameter `'ensureUniqueName'` set to `'cTrue'`, whereas the generated test program does not specify that parameter to make sure that your array gets the specified name. But in certain cases that can be the reason for duplicate array names.

Luckily there are two options to fix the problem:

a) You do not mind the array name because you want to use the handle

If you do not mind the array name you can just drop it from the parameter list like that:

```
MemoryManager_createArray(uuid
                          : %size(someArrayItem));
```

In that case the memory manager uses a "Universally Unique Identifier" (UUID) for the array name.

b) You want to have unique array names for whatever reasons

If you want to use human readable symbolic names you can add parameter `'ensureUniqueName'` to the parameter list like that:

```
hArray = MemoryManager_createArray(uuid
                                   : %size(someArrayItem)
                                   : 'inputArray'
                                   : cTrue);
```

Then use `Array_getName()` to get the real name:

```
name = Array_getName(hArray);
```

If the specified array name is not unique, the memory manager adds an underscore "_" plus an integer ID to make the name unique. This way '`inputArray`' may be changed to '`inputArray_0`' or '`inputArray_1`' or '`inputArray_n`' where "n" is any integer value.

Your comments are important to me! Any comments sent to me are greatly appreciated.

thomas.raddatz@tools400.de