

Axis C++ Memory Management Guide

<!-- --> <!-- -->

1. Axis C++ Memory Management Guide

1.0 Version

Feedback: axis-c-dev@ws.apache.org

1.1. Introduction

This guide records the memory management semantics and some of the rationales of the design decisions on which memory management semantics are based on. Being a C/C++ application, it is a must that all users as well as developers have a clear understanding on how to deal with memory allocation and deallocation.

The allocation and deallocation mechanisms are centered around the serializer and deserializer operations in Axis C++. At the moment, because Axis C++ support both C and C++ clients and services, the deserializer uses *malloc()* for memory allocation. Hence C++ users may have to live with *free()* (instead of *delete*) for deallocation. Of course one could still use *delete* from C++ programs, however this may not guarantee the cleanest memory deallocation.

1.2. Allocation/De-allocation Semantics

1.2.1. Parameters

*malloc()**new*

1.2.2. Other Objects

1. Any outbound objects created, either by a client application or by a handler, must be managed by the creator him(her)self. The Axis C++ engine does not delete any objects (HeaderBlocks, Attributes, BasicNodes etc.).
2. Any inbound objects that are created by the Axis C++ Engine as a result of deserialization should be deallocated by a target handler or the client application. Axis C++ Engine deallocates only the headerblocks that will remain in the deserializer (headerblocks with no target handler).

1.2.3. Return Values

The values returned by the Axis C++ engine must be memory cleaned by the user written code. The C++ code generated by the WSDL2Ws tool does contain destructors. However, at the moment, it is not guaranteed that the destructor of a generated class would clean all the pointer members (Some members are cleaned while others are not). Hence the users must have a look at the generated code to understand the semantics of memory cleaning. In case of C code, of course the user must take care of memory cleaning.

Please note that in case of arrays, both for C and C++, the Axis C++ engine returns a struct. Hence it is a must that the memory is cleaned properly.

C++ Example:

```
// testing echoIntegerArray xsd__int_Array arrint; // Parameter for method call
arrint.m_Array = new int[ARRAYSIZE]; arrint.m_Size = ARRAYSIZE; for (x = 0; x <
ARRAYSIZE; x++) { arrint.m_Array[x] = x; } printf ("invoking echoIntegerArray...\n");
xsd__int_Array arrintResult = ws.echoIntegerArray (arrint); // Deal with the return
value if (arrintResult.m_Array != NULL) { printf ("successful\n"); // Clean memory of
the returned array free(arrintResult.m_Array); } else printf ("failed\n"); // Clean
memory allocated for parameter delete [] arrint.m_Array;
```

1.3. Dealing with SOAP Headers

1.3.1. From Stubs

IHeaderBlock is a virtual class that defines the interface to deal with SOAP headers. You can create an IHeaderBlock at the client side using the API provided with Stub classes.

```
IHeaderBlock* Stub::createSOAPHeaderBlock(AxisChar * pachLocalName,
AxisChar * pachUri);
```

Note 1: It is advisable that if a user wants to delete a Hheader Block, (s)he uses the API provided by the Stub class to do so.

```
void deleteCurrentSOAPHeaderBlock();
```

Note 2: IHeaderBlock destructor will take care of the Header Block member variables and cleans them; BasicNodes (i.e its children) and the Attributes.

1.3.2. From Handlers

If the Header Blocks are created within a Handler then it is the responsibility of the Handler writer to clean those. For that the user can write the cleanup code either in the fini() method or in the destructor of the Handler, depending on the following situations

- If it is a session handler which needs to maintain its state, then the cleanup has to be done in the destructor.
- If it is a request type handler the clean up can be done in the fini() method of the Handler. Here the writer has to explicitly write the clean up code.

If a target handler access a Header Block created by the deserializer then it is the responsibility of the Handler to delete it.

1.4. Open Issues

As C++ is an object oriented language, one would ideally like to leverage constructors and destructors for memory management. However, the Axis C++ engine uses structs in some cases (e.g. Arrays) and uses *malloc()* to allocate memory. Hence the C++ programmer would be forced to use *free()* at times. When using *malloc()* and *free()* constructors and destructors are not called. However, as the Axis C++ engine currently supports both C and C++, it is not simple to replace all *malloc()* with *new* or *free()* with *delete*. At the same time, there are some places where *new* and *delete* are being used. They too cannot be replaced with *malloc()* and *free()* overnight. This memory management complexity is the price paid for dual support of C and C++. Efforts are under way to clean up the memory management mix-ups and still support both C and C++. Currently the proposed solution is to make the Axis C++ engine pure C++ and use a wrapper mechanism to support C.

When an array is de-serialized it uses C style memory re-allocation mechanism in the present code. C++ does not support *realloc()* and if we use *new* instead we have to allocate fresh memory blocks each time we need to increase the array size. This can be more expensive than using *realloc()*. Again the price paid for efficiency is that one has to use *free()* and not *delete []* from C++ code.