COM Types

An overview

Agenda

- COM Types
- Core IDL Data Types
- Automation Data Types
  - BSTR
  - Variant
  - BOOL
  - SafeArrays

COM Types

COM Type	IDL Keyword	Description
Interface	interface	A named set of semantically related methods
Coclass	coclass	A co name for a class that implements one or more interfaces
Enumeration	enum	Name value pairs (a way to avoid magic numbers)
Structures	struct	A collection of data types bound under a shared name
Unions	union	The C and C++ union is also supported in IDL, but no other languages understand that type!

The Core IDL Data Types (selection)

Are C centric and some of them are C/C++ only!

IDL Data Type	Description
boolean	8 bits. A C/C++ only data type.
byte	8 bits.
char	8 bits unsigned.
wchar	16-bit type for wide characters.
double	64-bit floating-point number.
float	32-bit floating-point number.
hyper	64-bit signed integer.
int	(16/) 32-bit signed integer.
long	32-bit signed integer.
short	16-bit signed integer.

Automation Data Types (selection 1)

Are the only data types accessible through late binding (IDispatch) and can be represented as a VARIANT data type

IDL Data Type	Description
VARIANT_BOOL	Automation compliant boolean.
double	64-bit floating-point number.
float	32-bit floating-point number.
int	(16/) 32-bit signed integer.
long	32-bit signed integer.
short	16-bit signed integer.
BSTR	The de facto COM string type
CY	8 byte fixed-point number (currency).
DATE	64-bit floating-point – number of days since Dec. 30, 1899.
enum	16-bit signed integer.

Automation Data Types(selection 2)

IDL Data Type	Description
struct	A C/C++ only data type.
IDispatch *	Pointer to IDispatch interface.
IUnknown *	Pointer to IUnknown interface. (an interface that is not derived from IDispatch)
VARIANT	Any variant-compliant type.
SAFEARRAY	A self-describing array of VARIANTS

BSTR

- COM strings must be Unicode.
- BSTR is a byte-length prefixed null-terminated array of Unicode characters.
- BSTR is the native string type in Visual Basic, but complicated to handle in C++.
- In C++ you have 3 choices for working with BSTR:
  - BSTR COM Library Functions (*are rather low level*)
  - If you don't use ATL use the _bstr_t wrapper class
  - If you do use ATL use the CComBSTR wrapper class

BSTR Rules

- OLE Automation defines the BSTR data type to handle strings that are allocated by one component and freed by another.

- **The caller handles BSTRs as follows:**
  - Free BSTR returned by called function or returned through a by- reference parameter of called function.
  - Free BSTR passed as a by-value parameter to a function.

- **The callee handles BSTRs as follows:**
  - Free BSTR passed in as a by-reference parameter before assigning a new value to the parameter. Do not free if a new value is not assigned to the by-reference parameter.
  - Do not free BSTR passed in as a by-value parameter.
  - Do not free BSTR that has been returned to caller.
  - If a BSTR passed in by the caller is to be stored by the callee, store a copy using SysAllocString(). The caller will release the BSTR that it passed in.
  - If a stored BSTR is to be returned, return a copy of the BSTR. The caller will release the BSTR that is returned.

- **Summary**
  - The callee frees a BSTR passed in as a by-reference parameter before assigning the parameter a new value. In all other cases, the caller frees the BSTR.

    (Reference MSDN Knowledge Base-article - 108934)

This rule applies to all kind of allocated memory!

BSTR COM Library Functions

Function	Description
`SysAllocString()`	Creates a BSTR based on an array of Unicode characters (typically `OLECHAR *`).
`SysReAllocString()`	Reallocates an existing BSTR to a new value.
`SysFreeString()`	Used to free memory attached to a BSTR created by `SysAllocString()`.
`SysStringLen()`	Return the character length of a BSTR.

Are low level. C++ programmers uses wrapper classes!

_bstr_t Server Side Programming

If you don't use ATL use the _bstr_t wrapper class

```
#include <comdef.h>  // for _bstr_t

STDMETHODIMP CCOMSvr::Foo1(/*[in]*/ BSTR Bstr){
  _bstr_t bstr = Bstr; // Don't release [in]-BSTRs
  ... DoSomeWork       // according to Q108934 (se MSDN)
  return S_OK;
}
```

```
STDMETHODIMP CCOMSvr::Foo2(/*[in, out]*/ BSTR *pBstr){
  _bstr_t bstr1 = *pBstr;  // Don't release
  ... DoSomeWorkWithINparameter
  ::SysFreeString(*pBstr); // Release before assigning a
                           // new string
  _bstr_t bstr2 = "A new string ...";
  *pBstr = bstr2.copy();   // Don't release
  return S_OK;
}
```

_bstr_t Client Side: C++

```
...
pCCOMSvr pCCOMSvr ("UsingBSTR.CCOMSvr.1");
_bstr_t bstr = "Test1";
pCCOMSvr->Foo1(bstr);
...
_bstr_t bstr2 = "Test2";
BSTR tmp = bstr2.copy();
pCCOMSvr->Foo2(&tmp);
_bstr_t bstr3(tmp,false);
...
WorkWith_bstr3
```

::SysFreeString() is called automatically when _bstr_t objects drops out of scope!

CComBSTR Server Side Programming

If you use ATL use the CComBSTR wrapper class

```
STDMETHODIMP CATLCOMSvr::Foo1(/*[in]*/ BSTR Bstr){
  CComBSTR bstr = Bstr; // Don't release [in]-BSTRs
 ... DoSomeWork
  return S_OK;
}
```

```
STDMETHODIMP CATLCOMSvr::Foo2(/*[in, out]*/ BSTR *pBstr){
  CComBSTR bstr1 = *pBstr; // Don't release
  ... DoSomeWorkWithINparameter
  ::SysFreeString(*pBstr);  // Release before assigning a
                            // new string
  CComBSTR bstr2 = " A new string ...";
  *pBstr = bstr2.Copy();    // Don't release
  return S_OK;
}
```

BSTR Client Side: C++

```
...
pCCOMSvr pCATLCOMSvr ("UsingBSTR.CATLCOMSvr.1");
CComBSTR bstr1 = "Test1";
pCATLCOMSvr->Foo1(bstr1);
...
CComBSTR bstr2 = "Test2";
BSTR tmp2 = bstr2.Copy();
pCATLCOMSvr->Foo2(&tmp2);
CComBSTR bstr3;
bstr3.Attach(tmp2);
...
workWith_bstr3
```

::SysFreeString() is called automatically when CComBSTR objects drops out of scope!

CComBSTR Scope Issues

- As with any well-behaved class, **CComBSTR** will free its resources when it goes out of scope.
- If a function returns a pointer to the **CComBSTR** string, this can cause problems, as the pointer will reference memory that has already been freed.
- In these cases, use the **Copy** method, as shown below.

```
// The correct way to do it
HRESULT MyGoodFunction(/*[out]*/ BSTR* bstrStringPtr)
{ // Create the CComBSTR object
  CComBSTR bstrString("Hello World");
  // Convert the string to uppercase  (do some work)
  bstrString.ToUpper();
  // Return a copy of the string.
  return bstrString.CopyTo(bstrStringPtr);
}
```

Explicitly Freeing the CComBSTR Object

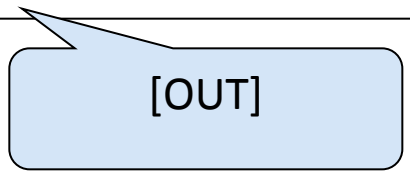
- It is possible to explicitly free the string contained in the **CComBSTR** object before the object goes out scope.
- If the string is freed, the **CComBSTR** object is invalid (~ can't be used for further work).

```
// Declare a CComBSTR object
CComBSTR bstrMyString( "Hello World" );
// Free the string explicitly
::SysFreeString(bstrMyString);
// The string will be freed a second time
// when the CComBSTR object goes out of scope,
// which is unnecessary but not an error.
```

Memory Leak Issues

- Passing the address of an initialized **CComBSTR** to a function as an **[out]** parameter causes a memory leak.
- In the example below, the string allocated to hold the string "Initialized" is leaked when the function OutString replaces the string.

```
CComBSTR bstrLeak(L"Initialized");
HRESULT hr = OutString(&bstrLeak);
```



- To avoid the leak, call the **Empty** method on existing **CComBSTR** objects before passing the address as an **[out]** parameter.
- Note that the same code would not cause a leak if the function's parameter was **[in, out]**.

ATL 7.0 String Conversion Classes and Macros

- ATL 7.0 introduces several new conversion classes and macros, providing significant improvements over the existing macros.
- The names of the new string conversion classes and macros take the form:

C*SourceType***2**[**C**]*DestinationType*[**EX**]

where:

- *SourceType* and *DestinationType* are described in the table below.
- [**C**] is present when the destination type must be constant.
- [**EX**] is present when the initial size of the buffer must be specified as a template argument.

SourceType/DestinationType	Description
A	ANSI character string.
W	Unicode character string.
T	Generic character string (equivalent to W when _UNICODE is defined, equivalent to A otherwise).
OLE	OLE character string (equivalent to W).

The recommended way of converting to a BSTR string is to pass the existing string to the constructor of **CComBSTR.**

BSTR / std::string Conversion

```
STDMETHODIMP CCOMSvr::Foo1(/*[in]*/ BSTR Bstr){
  //USES_CONVERSION; Not needed in ATL 7
  string str = CW2A(inBstr); // Unicode to ASCII conversion
  return S_OK;
}
```

```
STDMETHODIMP CCOMSvr::Foo2(/*[in, out]*/ BSTR *pBstr){
  string str1 = CW2A(*pBstr);
   ... DoSomeWorkWithINparameter
  ::SysFreeString(*pBstr);
  string str2 = "A new string ...";
  *pBstr = ::SysAllocString(CA2W(str2.c_str()));
  return S_OK;
}
```

Variant

- An essential data type in automation interfaces (IDispatch) is VARIANT.
- VARIANT is defined in OAIDL.H automation header file.
- A VARIANT can hold any possible automation data type.
- In C++ a VARIANT is a composite struct containing unions and structs.
- Think of a VARIANT as a struct of 2 fields
  - One field indicates the data type stored in the other field.
  - The other field holds the value
- USE COM helper functions when you need to work with VARIANTs

VARIANT Types

```
struct tagVARIANT {
  union {
  VARTYPE vt; // Current type stored in the VARIANT
  union {
      LONGLONG llVal;             // VT_I8.
      LONG lVal;                  // VT_I4.
      BYTE bVal;                  // VT_UI1.
      SHORT iVal;                 // VT_I2.
      FLOAT fltVal;               // VT_R4.
      DOUBLE dblVal;              // VT_R8.
      VARIANT_BOOL boolVal;       // VT_BOOL.
      SCODE scode;                // VT_ERROR.
      CY cyVal;                   // VT_CY.
      DATE date;                  // VT_DATE.
      BSTR bstrVal;               // VT_BSTR.
      IUnknown * punkVal;         // VT_UNKNOWN.
      IDispatch * pdispVal;       // VT_DISPATCH.
      SAFEARRAY * parray;         // VT_ARRAY.
      . // Many more
```

VARIANT Manipulation API Functions

Variant Manipulation Functions	Descriptions
`VariantChangeType()`	Converts a variant to another type.
`VariantChangeTypeEx()`	Converts a variant to another type, using a locale identifier (LCID).
`VariantClear()`	Releases resources and sets a variant to VT_EMPTY.
`VariantCopy()`	Copies a variant.
`VariantCopyInd()`	Copies variants that may contain a pointer.
`VariantInit()`	Initializes a variant.

VARIANTs in C++

```
// Create and initialize a VARIANT
VARIANT myVar;
VariantInit(&myVar);
// Setup the VARIANT to store a short
myVar.vt = VT_I2
myVar.iVal = 20;
```

BOOLs

```
// In plain C++
typedef int BOOL;
#define TRUE 1
#define FALSE 0

// In COM C++
typedef short VARIANT_BOOL;
#define VARIANT_TRUE ((VARIANT_BOOL)0xffff) // -1
#define VARIANT_FALSE ((VARIANT_BOOL)0)
```

BOOL C++ server

```
STDMETHODIMP CTestBools::ToggleCOMBool(
                         VARIANT_BOOL inBool,
                         VARIANT_BOOL *pOutBool)
{
  if (inBool == VARIANT_FALSE)
    *pOutBool = VARIANT_TRUE;
  else if (inBool == VARIANT_TRUE)
    *pOutBool = VARIANT_FALSE;
  else
    return E_FAIL;
  return S_OK;
}
```

SAFEARRAY

- It’s possible to use ordinary C arrays in IDL, but in automation interfaces (IDispatch) you have to use SAFEARRAYs.
- SAFEARRAY is defined in OAIDL.H automation header file.
- A SAFEARRAY can hold any possible automation data type.
- A SAFEARRAY can be multidimensional.

SAFEARRAY definition

```
typedef struct FARSTRUCT tagSAFEARRAY {
  unsigned short cDims; // Count of dimensions in this array.
  unsigned short fFeatures; // Flags used by the SafeArray
  unsigned long cbElements; // Size of an element of the array.
                            // Does not include size of pointed-to data.
  unsigned long cLocks; // Number of times the array has been
                        // locked without corresponding unlock.
  void HUGEP* pvData;   // Pointer to the data.
  SAFEARRAYBOUND rgsabound[1]; // One bound for each dimension.
  } SAFEARRAY;
```

SAFEARRAYBOUND represents the bounds of one dimension of the array. The lower bound of the dimension is represented by lLbound, and cElements represents the number of elements in the dimension. The structure is defined as follows:

```
typedef struct tagSAFEARRAYBOUND {
  unsigned long cElements;
  long lLbound;
  } SAFEARRAYBOUND;
```

Working with SAFEARRAY

```
STDMETHODIMP CComServer::Foo(SAFEARRAY** ppStrings){
  USES_CONVERSION;
  SAFEARRAY* pSA = *ppStrings; // Get local copy of strings.
  UINT numbOfDims = SafeArrayGetDim(pSA); // Be sure we don't have a
  if(numbOfDims != 1)                     // multidimentional array.
    return E_INVALIDARG;
  VARTYPE vt = 0;              // Be sure we have strings in the array.
  SafeArrayGetVartype(pSA, &vt);
  if(vt != VT_BSTR)
    return E_INVALIDARG;
  long ubound = 0;             // Get upper bound of array.
  SafeArrayGetUBound(pSA, 1, &ubound);
  BSTR* temp = NULL;          // Now show each string.
  SafeArrayAccessData(pSA, (void**)&temp);
  for(int i = 0; i <= ubound; i++)
    {
        MessageBox(NULL, W2A(temp[i]), "BSTR says...", MB_OK);
    }
  SafeArrayUnaccessData(pSA);
  return S_OK;
}
```

SAFEARRAY in ATL 4.0

- Before ATL 4.0 arrived working with SAFEARRAY required a lot of coding.
- ATL 4.0 supplies 2 helper template classes:
  - CComSafeArray
  - CComSafeArrayBounds
- For more info see VS Online Help

Working with SAFEARRAY in ATL 4.0

```
#include "stdafx.h"
#include "atlsafe.h"
int main(int argc, char* argv[]) {
  char cElement;
  char cTable[2][3] = {'A','B','C','D','E','F'};
  LONG aIndex[2]; // Declare the variable used to store the indexes
  CComSafeArrayBound bound[2]; // Define the array bound structure
  bound[0].SetCount(3);
  bound[0].SetLowerBound(0);
  bound[1].SetCount(3);
  bound[1].SetLowerBound(0);
  // Create a new 2x3 dimensional array
  pSar = new CComSafeArray<char>(bound,2);
  // Use SetAtMultiDimensional to store characters in the array
  for (int x = 0; x < 2; x++) {
    for (int y = 0; y < 3; y++) {
      aIndex[0] = x;
      aIndex[1] = y;
      ATLASSERT(pSar->MultiDimSetAt(aIndex,cTable[x][y]) == S_OK);
    }
  }
```