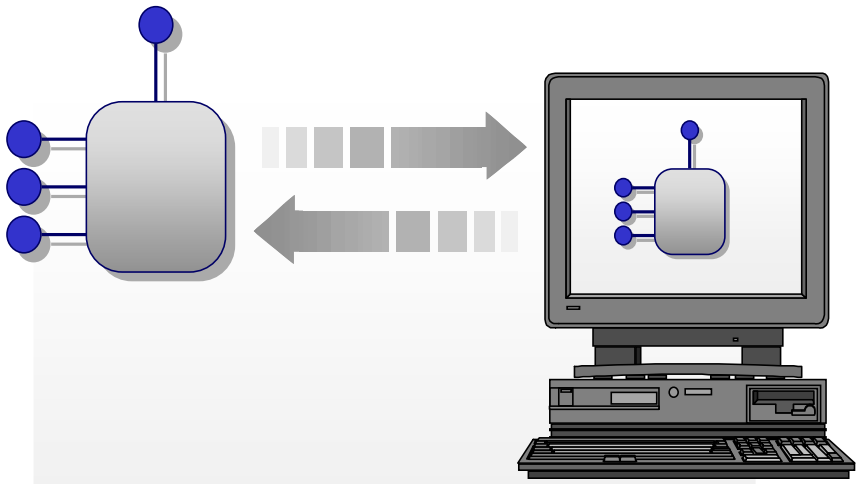
Strings, Variants and SafeArrays

QA-IQ



COM Programming

119

Chapter Overview

QA-IQ

- **Objective**
  - **Introduce COM's standard string, array and general purpose data types**
- **Chapter content**
  - **OLECHAR**
  - **BSTR, CComBSTR, _bstr_t**
  - **VARIANT, CComVariant, _variant_t**
  - **Safe Arrays**
- **Practical content**
  - **Pass BSTR, VARIANTS and SafeArrays between C++ and VB**
- **Summary**

120

In this chapter, we will discuss `OLECHAR`s, `BSTR`s, `VARIANT`s and SafeArrays and the various associated data types. A good understanding of these types is vital for implementing COM classes in C++.

Because, the fundamental types (`OLECHAR`s, `BSTR`s, `VARIANT`s and SafeArrays) are difficult to use directly in C++, a set of wrapper classes have been developed. You should use the wrapper classes wherever possible to simplify coding and increase readability.

The wrapper classes where developed by two separate teams at Microsoft. The compiler team produced the `_bstr_t` and `_variant_t` wrapper classes. These classes use reference counting and exception handling. The ATL team produced the `CComBSTR` and `CComVariant` wrapper classes. These classes are simpler than their compiler counterparts and do not use reference counting and exception handling.

OLECHAR

QA-IQ

- COM use OLECHAR for characters
  - defined differently on different operating systems
  - mapped to wchar_t on Win32 platforms
  - 16 bit characters are UNICODE

```
typedef wchar_t        OLECHAR;
typedef wchar_t*       LPOLECHAR;
```

```
#ifdef UNICODE
    typedef    wchar_t TCHAR ;
    #define    _tprintf
    wprintf;
    #define    _T(x)   L ## x
    #define    _TEXT(x)       L
## x
#else
    typedef    char    TCHAR ;
    #define    _tprintf
    printf;
    #define    _T(x)   x
    #define    _TEXT(x)       x
#endif
```

121

Since COM is a platform independent specification, the characters and strings used by COM must be defined in non Microsoft specific manner. Therefore COM defines its own character type called the `OLECHAR` and defines strings as an array of `OLECHAR`.

Obviously the `OLECHAR` must be mapped to a specific data type on each COM platform. On the Win32 platform, the `OLECHAR` is mapped to `wchar_t`, the Unicode character type.

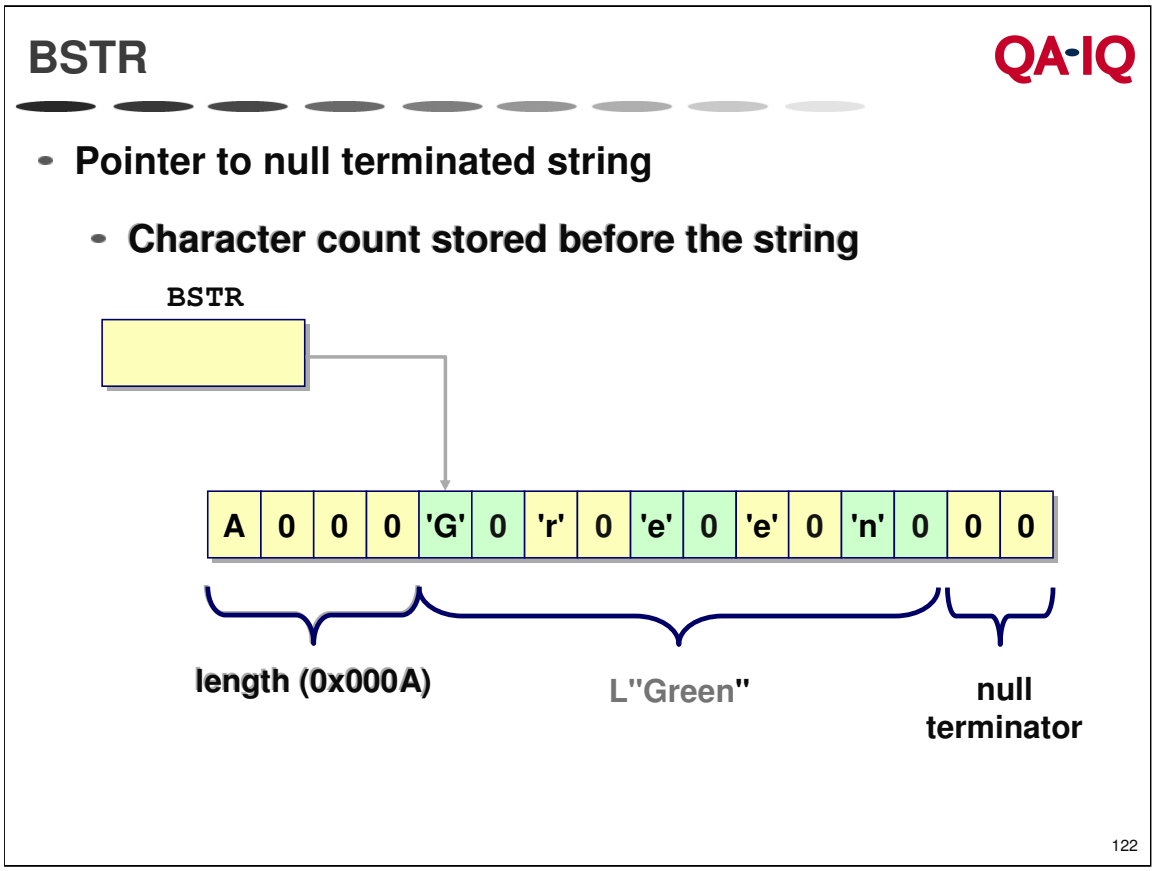
You don't have to use `OLECHAR` arrays in your COM classes; you could use ANSI (`char*`) or UNICODE(`wchar_t*`) strings directly. It is recommended that you follow the accepted COM practice and use UNICODE.

Note that you can write COM classes that are targeted to both ANSI and UNICODE by using the `TCHAR` and associated macros. This is very convenient for applications that run on both Windows 9x and Windows NT. The native string type for Window 9x is ANSI; the native string type for NT is UNICODE.

ATL also provides a series of macros for changing strings from one data type into another. These are contained within atlconv.h, and require the user to add the statement

```
USES_CONVERSION
```

to the top of each method where they will be used.



The `BSTR` is the native string type for Visual Basic. The `BSTR` is actually a pointer to a UNICODE string. Note that the length of the string is stored before the string data (and note how the count is represented). The string data is null terminated (with a 16 bit null character), but may also contain null characters embedded within the string. The string length is determined by the character count, not the first null character.

`BSTR`s cause problems for C++ programmers because they look like the `wchar_t*` type. However, pre-pending the string length in front of the string data ensures that BSTR is a distinct type. So don't be surprised to see:

```
typedef BSTR wchar_t*
```

The two types are different!

SysAllocString() and SysFreeString()

QA-IQ

- **BSTR is just a pointer**
  - **Storage allocated separately**

```
BSTR b;
b = SysAllocString( L"Sunday" );
p->EnterDay( b );
SysFreeString( b );
```

  - **Storage usually allocated by Client**
    - **Server allocates `[out]` parameters**
  - **Storage always deallocated by Client**

Client		*Server*
Allocate/Deallocate	`[in]`	**-**
Allocate/Deallocate	`[in,out]`	**Reallocate**
Deallocate	`[out]`	**Allocate**

123

Note that (as far as C++ is concerned) the `BSTR` type is just a pointer. To utilise a `BSTR` you must allocate storage for the string length and string data. The Win32 API provides just the function:

```
BSTR b = SysAllocString(L"Sunday");
```

You must remember to clean up this storage when you've finished with the `BSTR`:

```
b = SysFreeString(b);
```

Since BSTR is the native string type of Visual Basic you will often be using BSTRs to transfer strings between VB and C++. Is essential you understand who is responsible for allocating and deallocating the `BSTR` storage. If a client (often VB) passes a BSTR to your COM object (the server) then the client is always responsible for cleaning up the storage. The COM server is allowed to move the storage in the case of `[in,out]` parameters and obviously has to allocate the storage for `[out]` parameters but that is all.

CComBSTR

QA-IQ

- **ATL smart pointer class**
  - **wraps a BSTR**
  - **CTOR calls SysAllocString()**
  - **DTOR calls SysFreeString()**
  - **use Copy() to get BSTR**

```
class CComBSTR
{
public:
  BSTR m_str;

  CComBSTR(LPCOLESTR pSrc);
  CComBSTR& operator=(LPCOLESTR pSrc);
  operator BSTR() const;
  BSTR Copy() const;
  CComBSTR& operator+=(const CComBSTR& bstrSrc);
  HRESULT WriteToStream(IStream* pStream);
};
```

[in,out]

```
void ConvertString( BSTR *p )
{
  CComBSTR upper(*p);
  SysFreeString(*p);
  upper.ToUpper();
  *p = upper.Copy();
}
```

124

To use `CComBSTR` include the header:

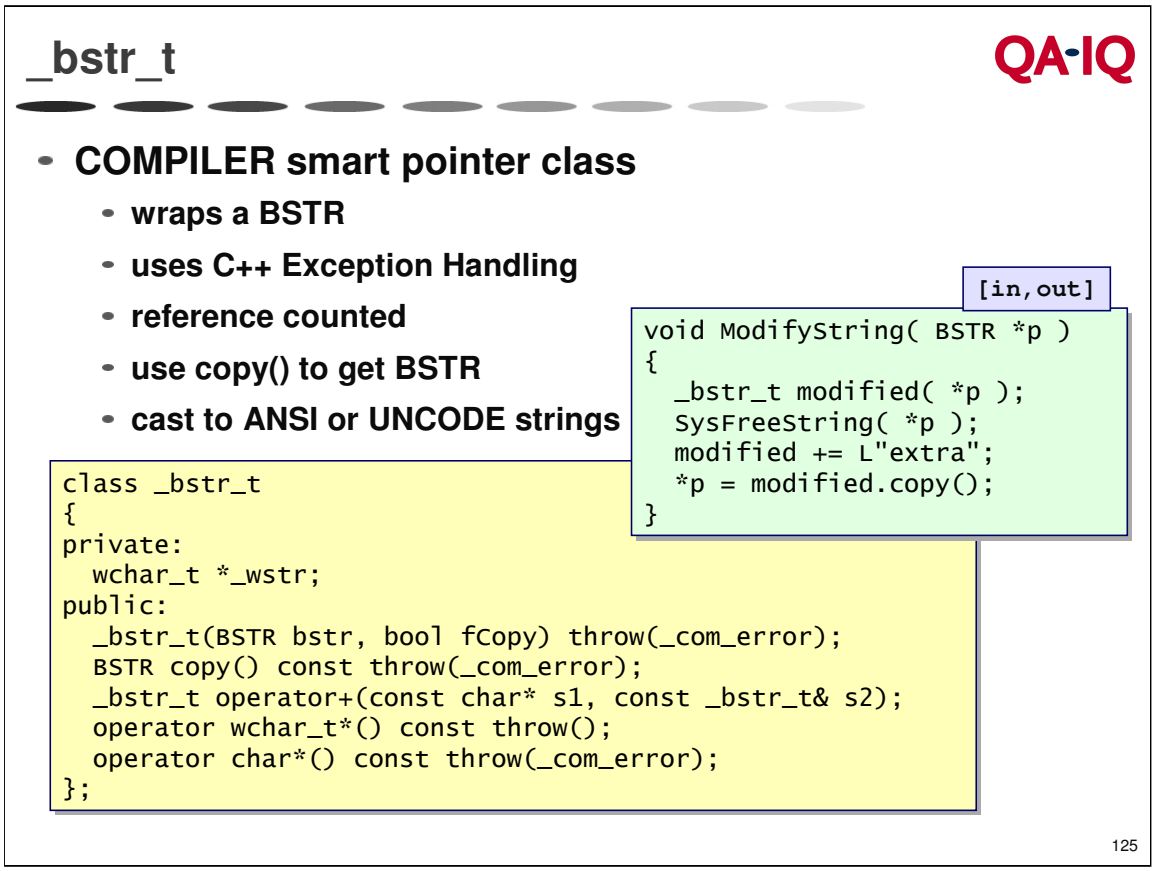
```
#include <atlbase.h>
```

The `CComBSTR` is a wrapper class for `BSTR`. The more common methods are shown above. The `CComBSTR` constructor will allocate storage for the encapsulated `BSTR` and save you having to call `SysAllocString()`. Similarly the destructor will deallocate storage for the encapsulated `BSTR` and save you having to call `SysFreeString()`. The constructor shown uses a pointer to a `OLESTR` which is equivalent to a `BSTR` (note, though, that a `BSTR` could contain embedded nulls, which would result in string truncation!).

When passing `BSTR`s from Visual Basic to C++ it is important to differentiate between `[in]` and `[out]` parameters. When the string is passed as an input parameter, the method should take a `BSTR`, but when passed as an output (or input/output) the method should take a `BSTR*`:

```
void InputAString(BSTR s)
{
  CComBSTR message(s);
  message.ToUpper();
}

void ModifyAString(BSTR *ps)
{
  CComBSTR upper(*ps);
  SysFreeString(*ps);           // remove old storage
  upper.ToUpper();
  *ps = upper.Copy();           // allocate new storage
}
```



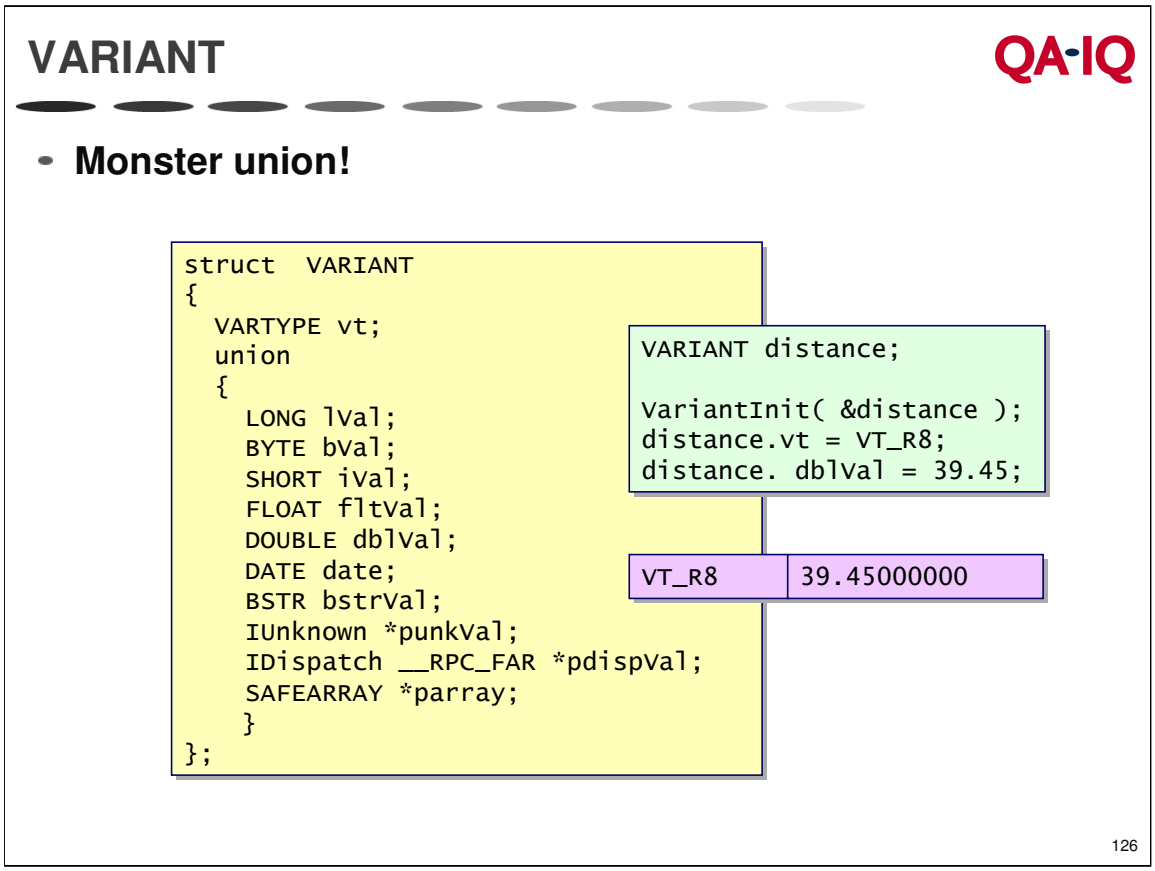
To use `_bstr_t` include the header:

```
#include <comdef.h>
```

The `_bstr_t` class is also a wrapper for a `BSTR`. This class was developed by the C++ compiler team and is considerably more sophisticated than the `CComBSTR` class. The class has a completely different set of member functions from `CComBSTR`.

The `_bstr_t` is reference counted for efficiency and the class throws exceptions when things go wrong. If you are using the ATL note that exceptions are switched off by default; you will have to explicitly change the project settings to support `_bstr_t`. Bear in mind that although support for exception handling will add about 4K to the size of your COM DLL, the cost is often worth it. If you use the `#import` directive you automatically (by default) use the compiler support classes and must enable exception handling. This behaviour can be changed using the `raw_native_types` option.

Aside from the above, the choice between `CComBSTR` and `_bstr_t` is a matter of choice. Examine the member functions of each class in order to decide which class is most appropriate. One advantage of the `_bstr_t` class is the set of cast operators defined. These overloaded operators make it simple to convert to `wchar_t*` and `char*`.



The `VARIANT` data is used extensively in Visual Basic and VB script to describe variables whose type can change dynamically. This is somewhat alien to the C++ programmer, but allows VB code such as:

```
Dim x As Variant
x = 100.00
x = "Hello"
```

To allow for dynamic type changes, the `VARIANT` is defined as a discriminating union. The discriminator holds the type as a 4 byte code and the rest of the union contains the data. Examples of valid discriminators are shown below; they include all the valid types in Visual Basic.

The data stored in the `VARIANT` can be as simple as an integer (discriminator `VT_I4`) or as complex as an array of doubles (discriminator `VT_R8 | VT_ARRAY`).

Before a `VARIANT` can be used it must be initialised using the `VariantInit()` API function. There are a number of other API functions defined to manipulate `VARIANT`s. As you can imagine, using these `VARIANT` types will be problematical in C++. But help is at hand; there are wrapper classes for the `VARIANT` developed by the same teams that developed the wrappers for `BSTR`s.

VARTYPE

QA-IQ

- **All automation types supported**

VT_EMPTY	nothing
VT_I2	2 byte signed int
VT_I4	4 byte signed int
VT_R4	4 byte real
VT_R8	8 byte real
VT_DATE	date
VT_BSTR OLE	Automation string
VT_DISPATCH	IDispatch *
VT_VARIANT	VARIANT *
VT_UNKNOWN	IUnknown *
VT_I1	signed char
VT_ARRAY	SAFEARRAY*

127

The `VARIANT` union allows all types used by VB and VB script. These are also the only types support for OLE Automation. Therefore if you are using a dual interface you will be using these types already.

Note that you do not have to wrap up integers, reals, safe arrays etc. in a `VARIANT` to pass data between VB and C++. You are allowed to do so, but you can pass the raw types if you prefer.

The situation is completely different with VB script. VB script only supports the `VARIANT` type so you must pass all parameters as `VARIANT`s. Passing parameters as `VARIANT`s is slow and cumbersome. Because of this, some COM developers provide separate interfaces for scripting languages (all parameters are `VARIANT`s) and for other languages (parameters are passed by raw types and not as `VARIANT`s).

CComVariant

QA-IQ

- **ATL smart pointer class**
  - **wraps a VARIANT**
  - **inherits from VARIANT**
  - **CTOR takes automation types**
  - **use Detach() to get at VARIANT**

```
void CreateVariant(VARIANT *p)
{
    CComVariant number(27.5);
    number.Detach(p);
}
```

```
class CComVariant : public VARIANT
{
public:
  CComVariant(const VARIANT& varSrc);
  CComVariant(int nSrc);
  CComVariant(double dblSrc);
  CComVariant(IDispatch* pSrc);
  HRESULT Attach(VARIANT* pDest);
  HRESULT Detach(VARIANT* pDest);
  HRESULT ChangeType(VARTYPE vtNew;
  HRESULT WriteToStream(IStream* pStream);
};
```

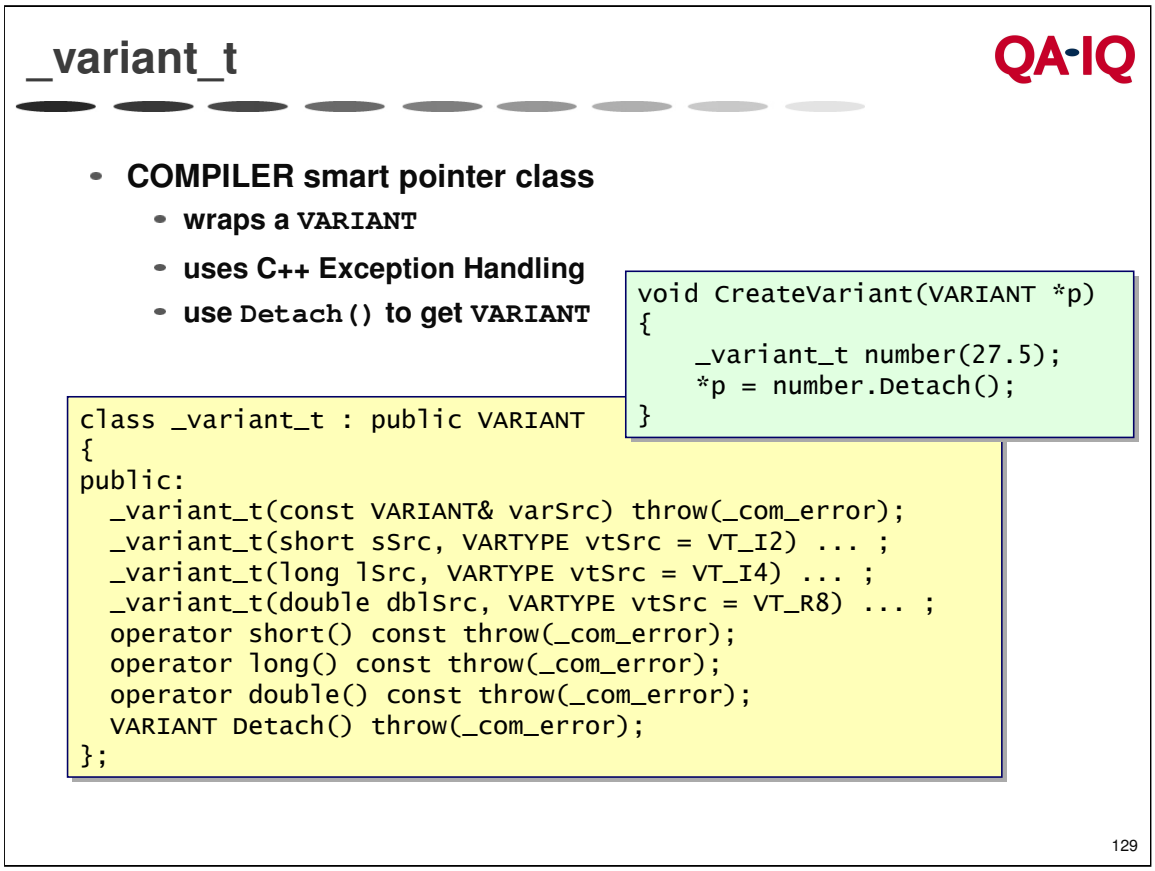
128

To use `CComVariant` include the header:

```
#include <atlbase.h>
```

The `CComVariant` is a wrapper class for the `VARIANT` type. The more common methods are shown above. The `CComBSTR` constructor will allocate storage for the encapsulated `VARIANT` and set the discriminator to the correct type. This is much easier that using the `VARIANT` API functions.

Use the `CComVariant` constructor to copy a `VARIANT`. If you want to make changes to a variant use the `Attach()` method to reference a `VARIANT`. You then then change the `VARIANT` using the `CComVariant` methods. When you are finished call the `Detach()` method.



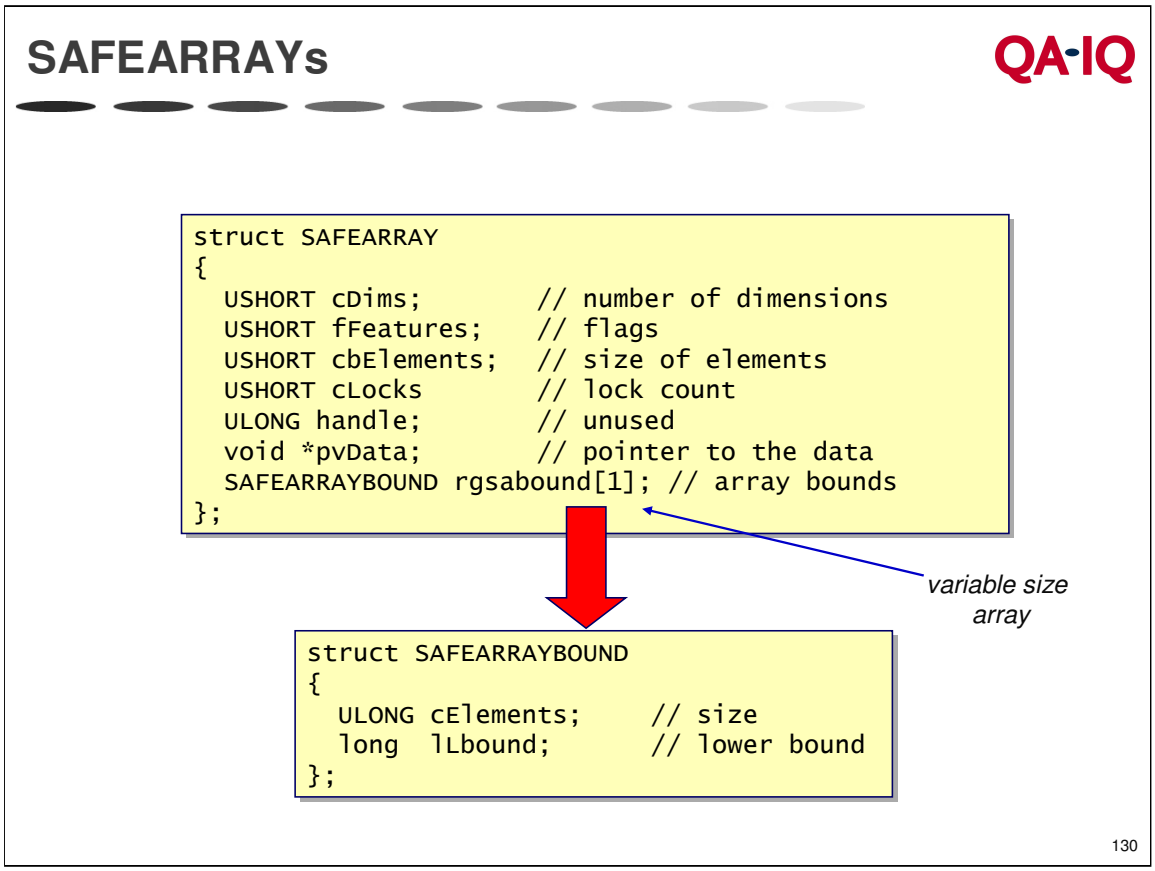
To use `_variant_t` include the header:

```
#include <comdef.h>
```

The `_variant_t` class is also a wrapper for a `VARIANT`. This class was also developed by the C++ compiler team. As you can see, the class has a completely different set of member functions from `CComVariant`.

As you would expect, this class also uses exception handling. Note that many member functions of the class are defined with a throw list of type `_com_error`. This means the methods will only throw `_com_error` objects, you do not have to code for other possibilities.

Again the choice between `CComVariant` and `_variant_t` is a matter of taste, but `_variant_t` does define a lot of very handy cast operators.



Passing arrays between different programming languages presents COM with special problems. That's because every language handles arrays differently! So COM dictates that you use a `SAFEARRAY` type to pass arrays as parameters. As we see the arrays can have multiple dimensions.

The `SAFEARRAY` type basically contains a pointer to all the elements of the array and a pointer to the dimension information. The array bounds are stored in a separate structure called a `SAFEARRAYBOUND`. A number of other structure members are defined as illustrated above.

Unlike the `BSTR` and the `VARIANT`, there are no wrapper classes for safe arrays. To manipulate safe arrays you must use the Win32 API functions.

Using SAFEARRAYs

QA-IQ

- **Single dimensional SafeArrays**
  - **SafeArrayCreateVector() creates a 1D array**
  - **SafeArrayAccessData() locks data and allows you access**
  - **SafeArrayUnaccessData() unlocks data**

```
void CreateArray( SAFEARRAY** ppArray )
{
  *ppArray = SafeArrayCreateVector( VT_I4, 0, 5 );
  long* data = 0;
  SafeArrayAccessData( *ppArray,
                       reinterpret_cast<void**>(&data) );

  for(int i = 0; i < 5; i++)
  {
    cin >> data[i];    // populate array
  }

  SafeArrayUnaccessData( *ppArray );
}
```

131

Multidimensional safe arrays are tedious to set up in C++, but one dimensional safe arrays are easy.

Simply call the `SafeArrayCreateVector()` API to allocate space for the array and its dimensions.

Use `SafeArrayAccessData()` to access the array. This API increments the lock count of an array, and retrieves a pointer to the array data.

When you are finished with the array use `SafeArrayUnaccessData()`. This API decrements the lock count of an array, and invalidates the pointer retrieved by the `SafeArrayAccessData()` function.

Summary

QA-IQ

- Strings should be passed as BSTRs
  - Can use wrapper classes such as CComBSTR and _bstr_t
- VARIANTs are large discriminated unions
  - Up to programmer to make sure types are correct
  - Scripting clients must pass all parameters as Variants
  - Can use wrapper classes such as CComVariant and _variant_t
- As all languages handle arrays differently they must be passed as SAFEARRAYs in COM
  - A wrapper class is provided for ATL 7.0

132