Typy danych

Typy danych kompatybilne z `oleautomation`

Standard COM udostępnia podstawowy zbiór typów, które mogą stanowić argumenty funkcji:

Typ danej	Rozmiar
unsigned char BYTE	8 bitów
VARIANT_BOOL	16 bitów
double	64 bity
float	32 bity
int	32 bity
long	32 bity
short	16 bitów
CY CURRENCY	8 bajtów

Typ danej	Rozmiar
BSTR	wskaźnik
DATE	64 bity
SCODE	32 bity
enum type	32 bity
VARIANT	unia o długości zależnej od typu
IDispatch*	wskaźnik
IUnknown*	wskaźnik
SAFEARRAY(*type*)	
*type**	

BSTR

W systemie Windows do zakodowania znaków i łańcuchów znaków stosuje się dwa standardy:
- ANSI (znak opisany 8 bitami)
- Unicode (16 bitowe kodowanie znaków)

Zestaw znaków ANSI jest bardzo ograniczonym zbiorem (255 znaków), jednak akceptowany jest przez wszystkie współczesne systemy operacyjne.
Zestaw znaków Unicode nie jest kompatybilny pomiędzy starszymi systemami Windows (95, 98) a ostatnimi wersjami systemu (Me, 2000, XP). Jest on jednak typowym sposobem kodowania dla systemów wspierających DCOM i aby uniknąć niepotrzebnych konwersji to przyjęto je do opisu zmiennych łańcuchowych.

BSTR

Dla kodowania Unicode zdefiniowano typ `wchar_t` dla przechowujący znak. Na bazie tego typu zdefiniowano następujące typy danych:

```
typedef  wchar_t OLECHAR;
typedef  OLECHAR* LPOLESTR;
typedef  const OLECHAR* LPCOLESTR;
```

Stałe znakowe dla takich ciągów muszą być poprzedzone prefiksem L:

```
LPOLESTR szStr = L"Unicode string";
```

BSTR

Zdefiniowano również funkcje kopiujące i wyznaczające długość łańcuchów Unicode:

```
wchar_t* wcscpy(wchar_t* target, const wchar_t* source);
size_t   wcslen(const wchar_t* source);

OLECHAR* ocscpy(LPOLESTR target, LPCOLESTR source);
size_t   ocslen(LPCOLESTR source);
```

BSTR

Na potrzebę kodu, który przewidziany jest do kompilacji zarówno w wersji ANSI jak i Unicode zdefiniowano typ danych TCHAR i makro T():

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

TCHAR* pStr = _T("Compilable as ANSI and UNICODE");
```

BSTR

Aby uprościć transfer danych przez serwer COM i pozwolić na przesyłanie znaku *null* wbudowanego w łańcuch, na bazie LPOLESTR opracowany został typ BSTR. Typ ten przechowuje wskaźnik do obszaru zawierającego łańcuch znaków Unicode, przed którym znajduje się 4 bajtowa liczba określająca długość łańcucha w bajtach.

BSTR

Takie podejście ma następujące zalety:

- **nie ma potrzeby określania długości łańcucha poprzez poszukiwanie końcowego znaku *null***
- **możliwa jest transmisja łańcuchów z wbudowanym znakiem *null***
- **w obszarze pamięci zmiennej BSTR można zapisać dowolnego typu dane, a cały obszar pamięci o podanej długości zostanie przesłany**

BSTR API

Nietypowa struktura typu BSTR wymaga zastosowania specjalnych funkcji alokujących pamięć:

```
BSTR SysAllocString(const OLECHAR* src);
BSTR SysAllocStringLen(const OLECHAR* src,
                       unsigned int size);
BSTR SysAllocStringByteLen(const OLECHAR* src,
                           unsigned int size);
void SysFreeString(BSTR bstr);
INT  SysReAllocString(BSTR* pbstr,
                      const OLECHAR* src);
INT  SysReAllocStringLen(BSTR* pbstr,
                         const OLECHAR* src,
                         unsigned int size);
UINT SysStringLen(BSTR bstr);
UINT SysStringByteLen(BSTR bstr);
```

BSTR API

Zdefiniowany zostały również zestaw funkcji konwertujący różne typy łańcuchów:

A2BSTR	OLE2A	T2A	W2A
A2COLE	OLE2BSTR	T2BSTR	W2BSTR
A2CT	OLE2CA	T2CA	W2CA
A2CW	OLE2CT	T2COLE	W2COLE
A2OLE	OLE2CW	T2CW	W2CT
A2T	OLE2T	T2OLE	W2OLE
A2W	OLE2W	T2W	W2T

```
A   - ANSI
OLE - OLECHAR
T   - TCHAR
W   - WCHAR
C   - const
```

BSTR
Zarządzanie pamięcią

Podczas operacji na zmiennej BSTR należy pamiętać o zasadach zarządzania pamięcią przez system COM – klient jest odpowiedzialny za alokację i zwalnianie pamięci. Jedynie dla parametru typu [in, out] serwer może zwolnić pamięć w celu zaalokowania nowej dla zwracanej wartości.

BSTR
Zarządzanie pamięcią – strona serwera

```
STDMETHODIMP CStringObjectUsingRawBstr::get_Value(
  /*[out, retval]*/ BSTR *pValue)
{
  *pValue = m_str; //Client can free
                   //m_str unexpectively

  //m_str=NULL;
  return S_OK;
}

STDMETHODIMP CStringObjectUsingRawBstr::get_Value(
  /*[out, retval]*/ BSTR *pValue)
{
  *pValue = SysAllocString( m_str );
  return S_OK;
}
```

BSTR
Zarządzanie pamięcią – strona serwera

```
STDMETHODIMP CStringObjectUsingRawBstr::put_Value(
  /*[in]*/ const BSTR const NewValue )
{
  m_str=NewValue;  //Client can free
                   //m_str unexpectively
  return S_OK;
}

STDMETHODIMP CStringObjectUsingRawBstr::put_Value(
  /*[in]*/ const BSTR const NewValue )
{
  if( SysReAllocString(&m_str, NewValue) )
    return S_OK;
  else
    return E_OUTOFMEMORY;
}
```

BSTR
Zarządzanie pamięcią – strona serwera

```
STDMETHODIMP CStringObjectUsingRawBstr::Swap(
  /*[in, out]*/ BSTR* pSwapValue)
{
  BSTR str = NULL;

  // Ownership transferred to str
  str =  *pSwapValue;

  // Set pSwapValue to own m_str
  *pSwapValue = m_str;

  // Set m_str to own str
  m_str = str;

  return S_OK;
}
```

BSTR
Zarządzanie pamięcią – strona serwera

```
STDMETHODIMP CStringObjectUsingRawBstr::Append(
  /*[in]*/ const BSTR const  Suffix,
  /*[out, retval]*/ BSTR* pNewValue)
{
  HRESULT hr=S_OK;
  int nLength = SysStringLen( m_str );
  int nSuffixLength = SysStringLen(Suffix);

  // Resize m_str to include the length of sSuffix
  if (!SysReAllocStringLen(&m_str,m_str,nLength+nSuffixLength )){
    pNewValue = NULL;
    hr = E_OUTOFMEMORY;
  } else {
    // Copy sSuffix to the end of m_str
    memcpy(m_str+nLength, Suffix, nSuffixLength*sizeof(OLECHAR));
    // Set pNewValue to return m_str to the client
    *pNewValue = SysAllocString( m_str );
  }
  return hr;
}
```

BSTR
CComBSTR wrapper

```
CComBSTR( );
CComBSTR( int nSize );
CComBSTR( int nSize, LPCOLESTR sz );
CComBSTR( int nSize, LPCSTR sz );
CComBSTR( LPCOLESTR pSrc );
CComBSTR( LPCSTR pSrc );
CComBSTR( const CComBSTR& src );

void Append( const CComBSTR& bstrSrc );
void Append( LPCOLESTR lpsz );
void Append( LPCSTR lpsz );
void Append( LPCOLESTR lpsz, int nLen );

void AppendBSTR( BSTR p );

void Attach( BSTR src );
BSTR Detach( );

BSTR Copy( ) const;

void Empty( )
```

BSTR
CComBSTR wrapper

```
unsigned int Length( ) const;

bool LoadString( HINSTANCE hInst, UINT nID );
bool LoadString( UINT nID );

HRESULT ReadFromStream( IStream* pStream );
HRESULT WriteToStream( IStream* pStream );


operator BSTR( ) const;

CComBSTR& operator =( LPCOLESTR pSrc );
CComBSTR& operator =( LPCSTR pSrc );
CComBSTR& operator =( const CComBSTR& src );

CComBSTR& operator +=( const CComBSTR& bstrSrc );

BSTR* operator &( );

bool operator !( );
```

BSTR
`_bstr_t` wrapper

W porównaniu do CComBSTR klasa _bstr_t jest bardziej efektywna, ponieważ stosuje *reference counting*, co redukuje niepotrzebne alokacje pamięci. Klasa ta stosuje wyjątki do zgłaszania nieprawidłowych sytuacji.

```
_bstr_t( ) throw( );
_bstr_t( const _bstr_t& s1 ) throw( );
_bstr_t( const char* s2 ) throw( _com_error );
_bstr_t( const wchar_t* s3 ) throw( _com_error );
_bstr_t( const _variant_t& var ) throw ( _com_error );
_bstr_t( BSTR bstr, bool fCopy ) throw ( _com_error );

BSTR copy( ) const throw(_com_error );
unsigned int length ( ) const throw( );

operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const throw( _com_error );
operator char*( ) const throw( _com_error );
```

BSTR
_bstr_t wrapper

```
_bstr_t& operator=( const _bstr_t& s1 ) throw ( );
_bstr_t& operator=( const char* s2 ) throw( _com_error );
_bstr_t& operator=( const wchar_t* s3 ) throw( _com_error );
_bstr_t& operator=( const _variant_t& var ) throw( _com_error );

_bstr_t& operator+=( const _bstr_t& s1 ) throw( _com_error );
_bstr_t operator+( const _bstr_t& s1 ) const throw( _com_error );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1 );
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1 );

bool operator!( ) const throw( );

bool operator==( const _bstr_t& str ) const throw( );
bool operator!=( const _bstr_t& str ) const throw( );
bool operator<( const _bstr_t& str ) const throw( );
bool operator>( const _bstr_t& str ) const throw( );
bool operator<=( const _bstr_t& str ) const throw( );
bool operator>=( const _bstr_t& str ) const throw( );
```

VARIANT

`VARIANT` jest złożonym typem danych , za pomocą którego można przekazać różnego typu dane.

```
// Overloading is not possible in IDL
HRESULT Item([in] BSTR Key, [out] IStopwatch** Stopwatch);
HRESULT Item([in] long Index, [out] IStopwatch** Stopwatch);

// Overloading functionality can be achieved using VARIANT type
HRESULT Item([in] VARIANT Key, [out] IStopwatch** Stopwatch);
```

VARIANT

```
struct  tagVARIANT
{
 union
 {
  struct  __tagVARIANT
  {
   VARTYPE vt;
   WORD wReserved1;
   WORD wReserved2;
   WORD wReserved3;
   union
   {
    LONG          lVal;
    BYTE          bVal;
    SHORT         iVal;
    FLOAT         fltVal;
    DOUBLE        dblVal;
    VARIANT_BOOL  boolVal;
    SCODE         scode;
```

```
    CY            cyVal;
    DATE          date;
    BSTR          bstrVal;
    CHAR          cVal;
    USHORT        uiVal;
    ULONG         ulVal;
    INT           intVal;
    UINT          uintVal;
    IUnknown      *punkVal;
    IDispatch     *pdispVal;
    SAFEARRAY     *parray;

    PVOID         byref;

    LONG          *plVal;
    BYTE          *pbVal;
    SHORT         *piVal;
    FLOAT         *pfltVal;
    DOUBLE        *pdblVal;
    VARIANT_BOOL  *pboolVal;
    SCODE         *pscode;
```

```
    CY            *pcyVal;
    DATE          *pdate;
    BSTR          *pbstrVal;
    CHAR          *pcVal;
    USHORT        *puiVal;
    ULONG         *pulVal;
    INT           *pintVal;
    UINT          *puintVal;
    IUnknown      **ppunkVal;
    IDispatch     **ppdispVal;
    SAFEARRAY     **pparray;

    DECIMAL       *pdecVal;

   } VARIANT_NAME_3;
  }VARIANT_NAME_2;
  DECIMAL decVal;
 } VARIANT_NAME_1;
};
```

VARIANT

LONG	VT_I4
BYTE	VT_UI1
SHORT	VT_I2
FLOAT	VT_R4
DOUBLE	VT_R8
VARIANT_BOOL	VT_BOOL
SCODE	VT_ERROR
CY	VT_CY
DATE	VT_DATE
BSTR	VT_BSTR
CHAR	VT_I1
USHORT	VT_UI2
ULONG	VT_UI4
INT	VT_INT
UINT	VT_UINT
DECIMAL	VT_DECIMAL
IUnknown *	VT_UNKNOWN
IDispatch *	VT_DISPATCH
SAFEARRAY *	VT_ARRAY

LONG *	VT_BYREF|VT_I4
BYTE *	VT_BYREF|VT_UI1
SHORT *	VT_BYREF|VT_I2
FLOAT *	VT_BYREF|VT_R4
DOUBLE *	VT_BYREF|VT_R8
VARIANT_BOOL *	VT_BYREF|VT_BOOL
SCODE *	VT_BYREF|VT_ERROR
CY *	VT_BYREF|VT_CY
DATE *	VT_BYREF|VT_DATE
BSTR *	VT_BYREF|VT_BSTR
CHAR *	VT_BYREF|VT_I1
USHORT *	VT_BYREF|VT_UI2
ULONG *	VT_BYREF|VT_UI4
INT *	VT_BYREF|VT_INT
UINT *	VT_BYREF|VT_UINT
DECIMAL *	VT_BYREF|VT_DECIMAL
IUnknown **	VT_BYREF|VT_UNKNOWN
IDispatch **	VT_BYREF|VT_DISPATCH
SAFEARRAY **	VT_BYREF|VT_ARRAY
VARIANT *	VT_BYREF|VT_VARIANT
PVOID	VT_BYREF

VARIANT API

```
VOID    VariantInit(VARIANT *pVariant);
        //Sets VARIANT as VT_EMPTY

HRESULT VariantClear(VARIANT *pVariant);
        //Clears data structures:
        //calls Release() for an interface type
        //or SysFreeString() for BSTR type
        //and then sets VARIANT as VT_EMPTY

HRESULT VariantCopy(VARIANT *pDestination, VARIANT *pSource);
        //Copies data from one VARIANT into another
        //Any data members that require
        //reference counting will be incremented appropriaterly
```

VARIANT API

```
HRESULT VariantCopyInd(VARIANT *pDestination, VARIANT *pSource);
        //Copies data from one VARIANT into another
        //but clears destination variant before copy.
        //Furthermore any data items that are pointers
        //will be changed to standard data types
        //(e.g. VT_BYREF|VT_BSTR into VT_BSTR)

HRESULT VariantChangeType(VARIANT *pDestination, VARIANT *pSource,
                          unsigned short wFlags, VARTYPE vt);
        //Coerces one variant into another.
        //Allows changes to/from string representation
        //Coercion is done through base types.
```

VARIANT
CComVariant wrapper

```
CComVariant( );
CComVariant( const CComVariant& varSrc );
CComVariant( const VARIANT& varSrc );
CComVariant( LPCOLESTR lpsz );
CComVariant( LPCSTR lpsz );
CComVariant( BSTR bstrSrc );
CComVariant( bool bSrc );
CComVariant( int nSrc );
CComVariant( BYTE nSrc );
CComVariant( short nSrc );
CComVariant( long nSrc, VARTYPE vtSrc = VT_I4 );
CComVariant( float fltSrc );
CComVariant( double dblSrc );
CComVariant( CY cySrc );
CComVariant( IDispatch* pSrc );
CComVariant( IUnknown* pSrc );

HRESULT Attach( VARIANT* pSrc );
HRESULT Detach( VARIANT* pSrc );
HRESULT ChangeType( VARTYPE vtNew, const VARIANT* pSrc = NULL );
HRESULT Clear( );
HRESULT Copy( const VARIANT* pSrc );
```

VARIANT
CComVariant wrapper

```
HRESULT ReadFromStream( IStream* pStream );
HRESULT WriteToStream( IStream* pStream );

CComVariant& operator =( const CComVariant& varSrc );
CComVariant& operator =( const VARIANT& varSrc );
CComVariant& operator =( LPCOLESTR lpsz );
CComVariant& operator =( LPCSTR lpsz );
CComVariant& operator =( BSTR bstrSrc );
CComVariant& operator =( bool bSrc );
CComVariant& operator =( int nSrc );
CComVariant& operator =( BYTE nSrc );
CComVariant& operator =( short nSrc );
CComVariant& operator =( long nSrc );
CComVariant& operator =( float nSrc );
CComVariant& operator =( double nSrc );
CComVariant& operator =( CY cySrc );
CComVariant& operator =( IDispatch* pSrc );
CComVariant& operator =( IUnknown* pSrc );

bool operator ==( const VARIANT& varSrc );
bool operator !=( const VARIANT& varSrc );
```

VARIANT
`CComVariant` wrapper

Uwagi:

- **CComVariant nie wspiera bezpośrednio danych opisanych przez referencję.**
- **Nie ma operatorów konwersji typu**
- **Nie ma operatorów <, >, <=, >=**
- **CComVariant dziedziczy bezpośrednio od VARIANT**

```
CComVariant comvariant(42);
VARIANT vatiant;
BSTR bstr;

VariantInit(&variant);
if (SUCCEEDED(VariantCopy(&variant, &comvariant))){
  if (SUCCEEDED(VariantChangeType(&variant, &variant,0,VT_BSTR)))
    bstr = SysAllocString(variant.bstrVal);
  VariantClear(&variant);
}
```

VARIANT
_variant_t wrapper

```
_variant_t( ) throw( );
_variant_t( const VARIANT& varSrc ) throw( _com_error );
_variant_t( const VARIANT* pVarSrc ) throw( _com_error );
_variant_t( const _variant_t& var_t_Src ) throw( _com_error );
_variant_t( VARIANT& varSrc, bool fCopy ) throw( _com_error );
_variant_t( short sSrc, VARTYPE vtSrc = VT_I2 ) throw( _com_error );
_variant_t( long lSrc, VARTYPE vtSrc = VT_I4 ) throw( _com_error );
_variant_t( float fltSrc ) throw( );
_variant_t( double dblSrc, VARTYPE vtSrc = VT_R8 ) throw( _com_error );
_variant_t( const CY& cySrc ) throw( );
_variant_t( const _bstr_t& bstrSrc ) throw( _com_error );
_variant_t( const wchar_t *wstrSrc ) throw( _com_error );
_variant_t( const char* strSrc ) throw( _com_error );
_variant_t( bool bSrc ) throw( );
_variant_t( IUnknown* pIUknownSrc, bool fAddRef = true ) throw( );
_variant_t( IDispatch* pDispSrc, bool fAddRef = true ) throw( );
_variant_t( const DECIMAL& decSrc ) throw( );
_variant_t( BYTE bSrc ) throw( );

void Attach( VARIANT& varSrc ) throw( _com_error );
VARIANT Detach( ) throw( _com_error );
```

VARIANT
_variant_t wrapper

```
void Clear( ) throw( _com_error );
void ChangeType(VARTYPE vtype,const _variant_t* pSrc=NULL) throw(_com_error);
void SetString( const char* pSrc ) throw( _com_error );

_variant_t& operator=( const VARIANT& varSrc ) throw( _com_error );
_variant_t& operator=( const VARIANT* pVarSrc ) throw( _com_error );
_variant_t& operator=( const _variant_t& var_t_Src ) throw( _com_error );
_variant_t& operator=( short sSrc ) throw( _com_error );
_variant_t& operator=( long lSrc ) throw( _com_error );
_variant_t& operator=( float fltSrc ) throw( _com_error );
_variant_t& operator=( double dblSrc ) throw( _com_error );
_variant_t& operator=( const CY& cySrc ) throw( _com_error );
_variant_t& operator=( const _bstr_t& bstrSrc ) throw( _com_error );
_variant_t& operator=( const wchar_t* wstrSrc ) throw( _com_error );
_variant_t& operator=( const char* strSrc ) throw( _com_error );
_variant_t& operator=( IDispatch* pDispSrc ) throw( _com_error );
_variant_t& operator=( bool bSrc ) throw( _com_error );
_variant_t& operator=( IUnknown* pSrc ) throw( _com_error );
_variant_t& operator=( const DECIMAL& decSrc ) throw( _com_error );
_variant_t& operator=( BYTE bSrc ) throw( _com_error );
```

VARIANT
_variant_t wrapper

```
bool operator==( const VARIANT& varSrc ) const throw( _com_error );
bool operator==( const VARIANT* pSrc ) const throw( _com_error );
bool operator!=( const VARIANT& varSrc ) const throw( _com_error );
bool operator!=( const VARIANT* pSrc ) const throw( _com_error );

operator short( ) const throw( _com_error );
operator long( ) const throw( _com_error);
operator float( ) const throw( _com_error );
operator double( ) const throw( _com_error );
operator CY( ) const throw( _com_error );
operator bool( ) const throw( _com_error );
operator DECIMAL( ) const throw( _com_error );
operator BYTE( ) const throw( _com_error );
operator _bstr_t( ) const throw( _com_error );
operator IDispatch*( ) const throw( _com_error );
operator IUnknown*( ) const throw( _com_error );
```

VARIANT
IDL optional parameters

Dla typu VARIANT możliwe jest użycie atrybutu `optional` w deklaracji metody interfejsu:

```
HRESULT MyFunc([in, optional] VARIANT Param1,
               [out, optional] VARIANT Param2);
```

W przypadku pominięcia argumentu, docelowa zmienna typu VARIANT będzie miała przyporządkowany typ VT_EMPTY.

SAFEARRAY

Typem danych, który może być wykorzystany do przesyłania tablic jest `SAFEARRAY(`*`type`*`)`. Parametr *`type`* może być każdym ze standardowych typów COM.

```
HRESULT Test([out] SAFEARRAY(BSTR) *ppArrayBstr);
```

SAFEARRAY API

```
SAFEARRAY *SafeArrayCreate(VARTYPE vt,
                           unsigned int nDims,
                           SAFEARRAYBOUND *pBound);
// tworzy nDims wymiarową tablicę
// o zakresach indeksów zdefiniowanych w pBound

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

SAFEARRAY API

```
HRESULT SafeArrayPutElement(SAFEARRAY *pSafeArray,
                            long *pIndices,
                            void *pVoid);
// ustawia wartość elementu o współrzędnych
// zdefiniowanych w wektorze pIndices

HRESULT SafeArrayGetElement(SAFEARRAY *pSafeArray,
                            long *pIndices,
                            void *pVoid);
// pobiera elementu o współrzędnych
// zdefiniowanych w wektorze pIndices
```

SAFEARRAY API

```
HRESULT SafeArrayCopy(SAFEARRAY *pSafeArray,
                      SAFEARRAY **ppSafeArrayOut);
// kopiuje tablicę z ppSafeArrayOut do pSafeArray.
// pSafeArray powinna zostać wcześniej utworzona
// przez SafeArrayCreate

HRESULT SafeArrayGetLBound(SAFEARRAY *pSafeArray,
                           unsigned int nDim,
                           long *pnLowerBound);
HRESULT SafeArrayGetUBound(SAFEARRAY *pSafeArray,
                           unsigned int nDim,
                           long *pnUpperBound);
// pobierają górny i dolny zakres indeksów
// dla wskazanego wymiaru tablicy
```

SAFEARRAY API

```
HRESULT SafeArrayDestroy(SAFEARRAY *pSafeArray);
// kasuje tablicę z pamięci właściwie dealokując
// wszystkie elementy

HRESULT SafeArrayRedim(SAFEARRAY *pSafeArray,
                       SAFEARRAYBOUND *pNewBounds);
// zmienia wymiary tablicy. Gdy nowy wymiar jest
// mniejszy od oryginalnego, usuwane elementy są
// właściwie dealokowane
```

SAFEARRAY
Przekazywanie zmiennej liczby argumentów

Za pomocą `SAFEARRAY` możliwe jest zdefiniowanie metody interfejsu, który będzie przyjmował zmienną liczbę i różne typy argumentów. Metoda taka musi posiadać atrybut `vararg`, a ostatnim argumentem musi być `SAFEARRAY(VARIANT)`.

```
[vararg] HRESULT Button([in] long Src, [in] SAFEARRAY(VARIANT) psa);
```

Enumerations

Enumerations podobnie jak w języku C++ stanowią spósob definiowania zestawu stałych typu integer. Tworzą kod programu bardziej czytelnym. Struktura deklaracji w języku IDL ma postać:

```
typedef [attributes] enum tag {
        [attributes] enum list
} enumname;
```

Enumerations

```
typedef
[uuid(EEBF6D1E-8EF1-4acf-9E5F-4D95E01D698A),
 helpstring("Work days enumeration")]
enum {
 [helpstring("1st day - Monday")]    tmMonday=0,
 [helpstring("2nd day - Tuesday")]   tmTuesday,
 [helpstring("3rd day - Wednesday")] tmWednesday,
 [helpstring("4th day - Thursday")]  tmThursday,
 [helpstring("5th day - Friday")]    tmFriday,
} TmWorkDays;


HRESULT SetNightShift([in] TmWorkDays as NightShiftDay);
```

Typy danych niekompatybilne z `oleautomation`

COM wspiera również inne, również złożone, typy danych, należy jednak pamiętać, że nie są one kompatybilne z językami skryptowymi takimi jak VBScript czy JScript i metody używające te dane nie będą poprawnie przez te języki obsługiwane.
Należy do nich zaliczyć:

- **zmienne łańcuchowe char*, wchar_t* zakończone zerem**
- **wektory i tablice danych o znanych rozmiarach**
- **struktury**
- **unie**

Zmienne łańcuchowe

Aby kompilator mógł odróżnić zwykły wskaźnik od wskaźnika, które wskazuje na łańcuch tekstowy kończony zerem (char*, wchar_t*) dla tego typu argumentu należy zdefiniować należy zdefiniować atrybut `string`:

```
HRESULT SetName([in, string] const char* pName);
HRESULT GetSetName([in, out, string] char** pName);
```

Aby uprościć definicję łańcuchów tekstowych zdefiniowane są typy LPSTR, LPCSTR, LPWSTR i LPCWSTR dostępne w IDL i ich odpowiedniki w C++:

```
typedef [string] CHAR *LPSTR;
typedef [string] const CHAR *LPCSTR;
typedef [string] WCHAR *LPWSTR;
typedef [string] constWCHAR *LPCWSTR;
```

Wektory i tablice o stałej wielkości

Jeśli chcemy przekazać tablicę o stałej wielkości, to jej rozmiar powinniśmy jawnie podać przy argumencie:

```
HRESULT SetVector([in] float Array[10]);
HRESULT GetSetVector ([in, out] float Array[10]);
HRESULT SetVector ([out] float Array[10]);

HRESULT SetArray([in] float Array[10][10]);
HRESULT GetSetArray([in, out] float Array[10][10]);
HRESULT SetArray([out] float Array[10][10]);
```

Wektory i tablice o zmiennej wielkości

Do przekazywania tablic o zmiennej długości można użyć dodatkowych atrybutów wskazujących na parametry przekazujące długość bufora oraz parametry optymalizujące transfer danych:

```
size_is()
length_is()
first_is()
last_is()
max_is()
```

Wektory i tablice o zmiennej wielkości `size_is()`

Atrybut `size_is()` definiuje, który parametr (musi być typu `[in]`) określa rozmiar tablicy, która ma być przesłana. Podstawową niedogodnością przy zastosowaniu tego typu atrybutu dla tablicy typu `[out]` jest to, że komponent może przesłać nie większą niż określona porcję danych. W przypadku, gdy zwracana jest mniejsza ilość danych, to mimo to transmitowana jest zadeklarowana ilość danych.

```
HRESULT SetVector([in] int x, [in, size_is(x)] float* Array);

HRESULT SetVector([in] int x, [in] int y,
                  [in, size_is(x,y)] float** Array);
```

Wektory i tablice o zmiennej wielkości `length_is()`

Jeśli komponent może zwracać tablicę o zmiennym wymiarze, można użyć atrybutu `length_is()`. Wskazuje on na parametr typu [out], w którym zwrócony zostanie faktyczny rozmiar tablicy. Ponieważ alokacją pamięci zawsze zajmuje się klient i on przygotowuje bufor do odbioru danych, to maksymalny rozmiar jest z góry określony.

```
HRESULT GetVector([out] int* ret_size,
 [out, length_is(*ret_size)] float Array[10]);

HRESULT GetVector([in] int alloc_size, [out] int* ret_size,
 [in, size_is(alloc_size), length_is(*ret_size)] float* Array);

HRESULT GetSetVector([in, out] int* actual_size,
 [in, out, size_is(*actual_size), length_is(*actual_size)] float* Array);
```

Wektory i tablice o zmiennej wielkości
`first_is()`, `last_is()`, `max_is()`

Atrybuty `first_is()` i `last_is()` używane są do wskazania indeksu pierwszego i/lub ostatniego elementu tablicy, który ma zostać przesłany. Atrybut `first_is()` może być używany razem z `length_is()` i określa ile danych począwszy od wskazanego indeksu ma zostać przesłanych. Atrybuty `first_is()` i `last_is()` mogą użyte z `size_is()`.

Ponieważ docelowa tablica jest zerowana przed odebraniem danych, to pominięte jej fragmenty będą miały zerowe wartości.

Atrybuty `max_is()` jest odmianą `size_is()`, z tą różnicą, że określa indeks ostatniego elementu tablicy, a nie jej rozmiar.

UWAGA: Pierwszy element tablicy ma indeks zero.

Struktury

Składnia definicji struktury w języku IDL jest podobna do tej stosowanej w C++. Dodatkowo każdego z pół można stosować atrybuty takie jak dla parametrów metod za wyjątkiem `[in]`, `[out]`. Struktury powinny być samoopisujące, tzn. dla łańcuchów znakowych i tablic należy stosować takie atrybuty jak `[string]`, `[size_is]`, `[length_is]`, itp.:

```
typedef struct
{
  int Size;
  [size_is(Size)] LPSTR* Data;
} ASCIIVector;
```

Unie

IDL wspiera unie z dyskryminatorem typu. Istnieją dwa sposoby deklaracji unii. Pierwszy z nich zawiera deklarację pola, które stanowi dyskryminator typu:

```
typedef union switch (int l1) UNION1 {
  case 0: float f1;
  case 1: double d2;
  default: ;
} STRUCT1;
```

Mapowane są one na następującą strukturę w języku C++:

```
typedef struct {
  int l1;
  union {
    float f1;
    double d2;
  } UNION1;
} STRUCT1;
```

Unie

Drugi typ deklaracji pozwala na podanie jedynie typu dyskryminatora, który znajduje o się poza strukturą unii:

```
typedef [switch_type(int)] union {
  [case 0] float f1;
  [case 1] double d2;
  [default] ;
} UNION2;
```

Do wskazania na zmienną opisującą zawarty typ stosuje się atrybut `switch_is()`:

```
typedef struc
{
  int l1;
  [switch_is(l1)] UNION2 w;
} STRUCT;

HRESULT SetData([in] int DataType, [switch_is(DataType), in] UNION2 Data);
```