C++2Any Overview

Vadim Zeitlin

October 20, 2006

Contents

0 What is C++2Any ?

C++2Any is a generalized interface generator, along the lines of SWIG. It allows to provide interfaces to other languages for the existing C++ libraries.

The main difference from SWIG is that C++2Any directly parses the C++ headers instead of requiring the user to create special `.i` files with the interface descriptions. C++2Any also offers better customizability and support for C and COM backends.

1 Using C++2Any

1.1 Restrictions on the C++ sources

Although C++2Any is supposed to parse the normal headers, it doesn't parse arbitrary C++ code. Here is the list of most important restrictions:

- Exporting template classes is not supported
- Function overloading is not supported
- Only default constructor is exported
- Operators are ignored

1.2 Support for non-standard types

Built-in support for the standard C++ types such as `int` and `double` is not enough except for the simplest of projects. Because of this C++2Any may be extended to handle any other types which may appear in the source C++ code easily. Of course, different types must be treated differently and not all of them can be simply mapped to a type supported by the current backend, sometimes a non trivial transformation has to happen.

To allow for such transformations of arbitrary complexity, the wrapper code generated by C++2Any uses user-defined C++ template functions to perform the translation between C++ and backend types. There are two such functions: `To<>()` transforms its arguments to the specified C++ type and `From<>()` does the reverse transformation. Both of these functions must be defined inside `C2A::`*`backend`* namespace where the *`backend`* is the name of the current backend.

For example, in C backend the standard `std::string` class is mapped to `char *` and so the following specializations of the template functions must be defined:

- `C2A::C::To<std::string>(const char *)` which simply creates a C++ string from a C string
- `C2A::C::From<char *>(const std::string& s)` which should allocate and return a new C string containing the copy of the given C++ string object

2 Getting Started with C++2Any

In this chapter we show how to quickly start using C++2Any with your classes to produce COM servers wrapping them. Note that you need to have support classes for the COM wrappers to work and that the C++2Any templates being used may have to be adjusted for your COM support classes if they are different from the ones provided by TT-SOLUTIONS.

2.1 A very simple example

Please start by checking that you can create a working COM server from the file `examples/hello/hello1.h`. Here are the steps you should follow:

1. Copy the files `hello1.h` and `hello_impl1.cpp` to just `hello.h` and `hello_impl.cpp` respectively. Although not strictly necessary, of course, this will allow you to try other `helloN` examples easier later just by copying another pair of files to the same names but by keeping the same project file and all the rest.
2. Change to directory `examples\hello` and run C++2Any using a command like this:

   ```
   cpp2any /t../../templates /oCOM /DLIBRARY=HelloLib
                          hello.h
   ```

   which will generate the files `include/COM/hello.h`, `src/COM/hello.cpp` (the directories must exist) and `hello.idl` file under the directory `COM` specified as output one.
3. If you are using Microsoft Visual C++ 6 or 7, you may directly open the example project file `hellocom.dsp`. Otherwise you will have to create your own project file or makefile to build the COM DLL. This is not complicated, you just have to include all of the following files in it:

- the file `hello_impl.cpp` which implements the functions declared in `hello.h`
- COM support files (see `hellocom.dsp` for a full list)
- the files generated by C++2Any and the `hellocom.idl` file

4. Building this project should create a DLL with appropriate name. Before it can be used as a COM server, it has to be registered:

```
regsvr32 hellocom.dll
```

5. Now you should be able to use the COM server in the same way as any other one. For example, you may open `hello1.bas` program in the Visual Basic environment and run it there: the C++ function will be called.

2.2 Preparing C++ sources

Going through `examples/hello/hello1.h` you may now see that it is just a plain C++ header file with Doxygen (also known as javadoc) comments. The Doxygen comments are not mandatory, they are simply used to document the code so you could perfectly well omit them entirely. Only one of them is special to C++2Any : it is the `@iid long-string-of-hex-digits` one in the line 15. This special comment is recognized by C++2Any and specifies the interface id for the COM interface `IHello` representing the C++ class `Hello`.

This comment was the only modification needed for this – admittedly, extremely simple – header. But even it is not really necessary as the value of the IID can be alternatively[1] specified on the C++2Any command line, just as the value of any other variable. So the original header could have been left unmodified and you could run C++2Any like this:

```
cpp2any /DIID_Hello=xx-x-x-x-xxx ...  hello1.h
```

Other special comments may appear in C++ sources, please see chapter 3 for details. However there are usually going to be only very few of them.

2.3 Presenting other C++2Any features

The `hello1` example is, on purpose, the most trivial one possibly, so there is not much to see here. Other examples show more interesting aspects of C++2Any :

[1] In fact, it can appear both in the header file and on the command line and in this case the latter one takes precedence

hello2 Here you can see how C++2Any automatically handles the C strings (`const char *` argument).

hello2b Alternatively, C++ `std::string` is also handled correctly.

hello3 The parameters of other user-defined classes are supported

hello3b ... just as return types. Note that to build this and the previous examples you will need to uncomment the `Number` coclass definition in `hellocom.idl`.

hello4 Standard container types (only `std::vector` and similar user-defined types) may be used without problem.

hello5 C++ inheritance is correctly translated into the backends which support inheritance (such as COM) or emulated as well as possible for the others (C).

3 Special C++ Comments

Normally keeping the C++ sources pristine is a good idea, however you may need, or want, to add some special comments in them to when using C++2Any . One such comment was already shown in the "Getting Started" chapter, here all of them are detailed.

3.1 Special comments syntax

The special keywords must appear in the comment preceding the class, method or another declaration, in order to be recognized by C++2Any . They use the same syntax as usual Doxygen comments, that is they may be prefixed either by `@` or by `\`. If there is an argument, it should follow the comment after one or more spaces.

3.2 General comments

The special comments in this section apply to all backends.

rename Allows to rename a class, enum or method when exporting it. By default, the same names are used in the generated code as in C++ sources but this special keyword allows to change this.

noexport This keyword may be used with any C++ declaration. If it appears before a class method, it is not exported, i.e. will not be visible at all in the generated wrappers and will be invisible from the outside. If it appears before a class or enum declaration, the entire class or enum is omitted. Note, however, that

if another class derives from a “noexported” class, the (public) methods of the base class would still be exported as part of the derived class. On the other hand, if a method is marked with this keyword in a base class, it is not exported even if its declaration appears again in a derived class, so it isn't necessary to repeat it in all derived classes.

nocreate This keyword may be used with C++ classes only. When it appears, the class is still exported but the objects of this class cannot be created directly by the user. Of course, for the class to be at all useful, you should have some functions elsewhere returning objects of this class – otherwise the user would never see it. This keyword is usually used with abstract base classes but not only.

noimpl Indicates that although this method should be exported, the implementation for it shouldn't be generated by C++2Any . This, of course, means that it will have to be written manually and included alongside the code generated by C++2Any . This keyword should be rarely used, but sometimes the exported should do something more than just calling the corresponding C++ method and it allows for this. This comment is only valid for the functions, not classes.

All of the keywords above except for `rename` can be followed by a comma-separated list of backends they apply to. If no such list is present, the option applies to all backends by default. For example,

```
/* @noexport COM */
```

can be used to prevent a class or a function from being exported in the COM interface but still be exported as a Java class or a method.

3.3 COM-specific comments

Some other special comments are currently only used with the COM backend:

iid A valid UUID[2] must follow. This comment may only appear in front of a class declaration and allows to specify the IID for the COM interface generated for this class. If it is not used in the C++ sources, the variable `IID_<ClassName>` must be defined elsewhere as it is required by COM.

property Indicates that the method should be exported as a property accessor, see section 5.3 for more details. This comment can only be used at method level.

[2] universally unique identifier

iterator This keyword (and related indexed_iterator one) are used to implement support for OLE Automation. They have a required argument which is the name of the class method providing access to collection elements, see 5.4 for further details.

3.4 Java-specific comments

A special `enumprefix` comment can be used with the enum declarations to indicate the common prefix for the enum elements names which should be removed when generating their Java names. Please see 6.3 for more details.

4 C++2Any Invocation

C++2Any is a command line tool and so should be run from the command line or shell prompt. As any command line program, it has a number of command line options (described in this chapter) but in the simplest case you can run the program just by giving it the name of the input file to process:

```
cpp2any filename.h
```

To customize C++2Any you should use the options described below.

Notice that it is also possible to put the options in a `C2AFLAGS` environment variable: if it exists, its contents will be used as if it appeared on the command line, however if the same option is present both in environment variable and on command line, the latter has higher priority, i.e. it is possible to override the environment flags on the command line.

The C++2Any options come in two kinds: global ones and the backend-specific options. The global options affect all backends and have simple names while the backend-specific options always start with the backend name followed by underscore as prefix and only affect the behaviour of the specific backend. All global options are documented here while the backend options are documented in the chapters covering the corresponding backend.

4.1 Input Options

`/I:`*`dir`* Specifies an additional directory with C++ header files. This is similar to a quasi-standard `/I` C/C++ compilers option and may appear multiple (or zero) times in the command line.

`/t:`*`dir`* Specifies the directory with the template files

4.2 Output Options

`/f:`*`list`* The output format(s) to use. The list may be either a single format (e.g. `COM`), a combination of them (`COM,C,XLL`) or the special keyword `all` meaning to generate all formats.

`/o:`*`dir`* Specifies the directory for the output files, the values of other output options are relative to this directory unless they are specified as absolute paths.

`/n:`*`name`* The basename of the generated files. By default, it is the same as input file name.

The backends provide other output options allowing finer grained control over the location of the output files.

4.3 Definition Options

The **/D** ("define") option may be used to set the template variable values from the command line. Its syntax is **`/D:`***`VAR`* or **`/D:`***`VAR=VALUE`* to either just define the variable (so that conditional expansion constructs using it would evaluate to true) or to define it with the given value.

There is also **/d** option which can be used to define C preprocessor symbols which will be used during the parsing of input C++ headers. As for the **/D** above, the option takes the symbol name and, optionally, its value which defaults to `1` if not specified. Notice that the special `__CPP2ANY__` is automatically predefined in any case.

4.4 Custom Type Options

The **/m** ("map") option is used to map custom types to the backend-specific types. As such, this option doesn't really exist as a global option but each backend defines it and it has the same meaning for all of them. For example, `/COM_m:string=BSTR` means that COM backend should map all string objects to `BSTR` COM string type. In a similar way, there is an **/me** ("map enum") option which can be used to let C++2Any know that the given type (which presumably comes from the sources not parsed by it) is, in fact, an enumeration. The latter option may have a value to specify which type this enum is to be mapped to or it may not have it in which case it simply specifies that the given type is to be treated as an enum.

There are also two other versions of this option for the template types. **/mp** ("map pointer") may be used to indicate that the given template type must be treated as a smart pointer, for example `/COM_mp:auto_ptr`.

And **/mc** ("map container") means that the given template is a container type, for example `/COM_mc:vector`. Note that all of these examples are really not needed because C++2Any has built-in knowledge of the standard types such as `string`, `auto_ptr<>` or `vector<>`. However the same syntax may be used for any other user-defined types as well.

4.5 Miscellaneous Options

/version Shows the program version and copyright notice and exits.

@file The *file* can contain further options (but not input filenames) which are processed as if they appeared on the command line itself. The empty lines and lines starting with `'#'` (comments) are ignored.

5 COM Backend

This section describes how the generation of COM wrappers for C++ code works. To be more precise, we use COM and OLE Automation here interchangeably, even though OLE Automation is a rather restricting subset of COM only. However to be able to use COM components from languages such as Visual Basic, Visual Basic for Applications, VBScript or JScript, the COM interfaces must be OLE Automation compatible so this is what we are mostly interested in.

COM backend generates the following kinds of output files:

class.idl Interface definition file used by COM. It must be included in the target project and compiled with `MIDL` to produce the C++ interfaces description file `project_interfaces.h`, the IIDs and CLSIDs definition file `project_i.c` and, most importantly, the type library `project.tlb`.

class.h For each exported class the header with the declaration of the class implementing the interface corresponding to this class is generated.

class.cpp C++ file implementing the class declared above by forwarding all of its methods to the real class.

5.1 Backend-specific Output Options

In addition to the general options described in the section 4.2, COM backend provides the following ones:

`/COM_idl:`*`path`* The path for the generated IDL file, by default it is the same as the value of **/o** option.

/COM_cpp:*path* The path for the generated C++ files, by default it is `src/COM`.

/COM_h:*path* The path for the generated header files, by default it is `include/COM`.

For all the options taking a *path* argument, it may be either an (existing) directory or the full path name of the output file – this latter possibility allows to have different names for the IDL and CPP files, for example.

5.2 Main IDL File

C++2Any generates a COM IDL[3] file for each of the C++ headers, however there is one IDL file which must be created manually and which should include all of them. The reason for not generating is that, quite simply, the tool doesn't know about all the files in the project and so if it had to generate it, you would have to always run it over all the headers at once which owuld be impractical. Also, this main IDL file is very simple as it just includes all the other (complex) IDL files which are generated automatically.

The IDL file in question should have the following contents:

```
// standard IDL files always needed
import "unknwn.idl";
import "oaidl.idl";

// import all generated IDL files, one for each input header
import "hello.idl";
...
import "goodbye.idl";

[
    helpstring("Your user-readable description goes here"),
    uuid(0162c8a5-bf51-44e8-b660-46063727bdf6),     // LIBID_HELLO
    version(1.0)
]
library HelloLib // this should be the same as LIBRARY variable
{
    importlib("stdole2.tlb");

    // the remaining block must be repeated for each
    // exported creatable class
    [
```

[3] Interface Description Language

```
        helpstring("Description of this class"),
        uuid(691cf048-6f72-48cc-9d9d-5642741e0804 )   // CLSID_Hello
    ]
    coclass Hello
    {
        [ default ] interface IHello;
    };
};
```

The UUIDs appearing above must, of course, be changed for your project. The new UUIDs are generated with the help of a standard `uuidgen` command line tool, simply run it to create them (`uuidgen` can be found in the Microsoft Visual Studio Tools directory).

The `helpstring` parts are not mandatory but may be helpful for the user.

Finally, notice that `coclass` declaration should only be used for the classes which may be directly created by the external code. If an object of a class can only be returned by a method of another class and can't be created by the user, this declaration shouldn't be used.

5.3 Properties support

OLE Automation has the notion of "properties" which are mapped to `put_PropName` and `get_PropName` methods in C++. C++2Any does the reverse mapping, that is it can map C++ setter or getter methods to an OLE property in the generated code. This is not done automatically, however, because it is not always appropriate but a special keyword `@property` must be used in the comment for the methods which should be mapped to property accessors.

All of read-only, read-write and even write-only properties are supported. You don't have to specify whether the method is reading or writing the property as the tool supposes that all const methods are for reading and everything else is for writing.

Finally, a property typically doesn't have the same name as the method, i.e. for an accessor named `GetResult()` the property name should be `Result`. C++2Any knows several standards prefixes (`Set`, `Put` for writing and `Get`, `Is` for reading) and discards them automatically when generating the property name (if the method is called just `Set` or `Get`, the property name becomes `Value`). If it still doesn't get it right, you may specify the property name explicitly by putting it after `@property`, e.g. `@property Result`. And if C++2Any mistakenly maps a C++ method to a property, you can force exporting it as a COM method by using another special comment: `@method`.

5.4 Collection support

OLE Automation clients often expect to access sequences of objects as *collections* rather than arrays. C++2Any currently has support for collections with integer indices. To indicate that collection methods should be generated for a class, either `indexed_iterator` or just `iterator` special comment should be used in the class declaration. The latter one declares a collection whose elements can only be accessed sequentially while the former one defines an additional `Item` COM method allowing to access the collection elements by index.

In either case, the name of the method providing access to the collection contents should be specified as parameter. This method must return a standard container object which contains the collection elements. Note that this iterator method will not be exported in the usual way by C++2Any .

Here is a brief example:

```
class Leg { ... }; // exported normally

/// @iterator GetLegs
class Animal
{
public:
    ...
    const std::list<Leg>& GetLegs() const;
};
```

Remark: in this example `indexed_iterator` keyword cannot be used because `list` doesn't provide random access to elements, a `vector` would have to be used if random access is essential.

Finally please note that if a base class has either `iterator` or `indexed_iterator` comment, it is inherited by all of the derived classes which automatically become OLE Automation collections as well. C++2Any will flag as an error any attempt to specify either of these comments for a class which already inherits them from a base class.

5.5 Generic Limitations

OLE Automation is not case-sensitive, so exporting two functions or classes differing only by case is not going to work. Overloaded functions are not supported by COM neither.

5.6 Namespaces

C++ namespaces become COM type libraries. As the type libraries can't nest, the type library name is, by default, the concatenation of all namespaces containing it, e.g. for

```
namespace Out
{
    namespace In
    {
        class Foo { ... };
    }
}
```

the type library would be named `Out_In`.

5.7 Classes

Exported C++ classes map one to one to COM interfaces. Single inheritance is supported and maps to COM interfaces inheritance.

The main difference between classes and interfaces is that the latter don't have a constructor and can't be created. The creatable objects in COM are the so-called coclasses and C++2Any creates a coclass for each C++ class by default.

5.8 Methods

Only input parameters (values and constant pointers or references) are allowed in OLE Automation compatible COM interfaces as OLE Automation supports only a single return value (at most).

Static class members are not exported as they are not directly supported by COM.

5.9 Parameters and Return Values

Simple types are passed as is, with the exception of `bool` which is mapped to COM-specific `VARIANT_BOOL`. The strings, whether of C (`const char *`) or C++ (`std::string`) kind, are mapped to the OLE Automation Basic string (`BSTR`) data type. C arrays and C++ vectors are mapped to OLE `SAFEARRAY`s.

Variables of user-defined exports types are always passed as a pointer to their corresponding interface class. Parameters of all other types cannot be passed to COM without explicitly defining how they should be handled.

5.10 Error Handling

In OLE Automation, each method returns an `HRESULT` value which contains the error code. All C++ exceptions raised inside the user functions called from COM wrappers are translated to COM error codes. Common exceptions (such as `std::bad_alloc`) are recognized automatically and mapped to the appropriate value (`E_OUTOFMEMORY`) but the user should define the COM error codes for the non-standard exceptions. By default, all of them are mapped to `E_UNEXPECTED` error code which is a generic catch all COM error and doesn't provide any useful information to the method caller.

6 Java Backend

This section describes how the generation of Java wrappers for C++ code works. Globally speaking, for each C++ class, C++2Any generates a shadow Java class which has the same – or as close as possible – methods as the C++ class. All methods of the Java class are declared as `native` which means that they are implemented in C++, using JNI, where they are forwarded to the methods of the original class.

Java backend generates the two different output files for each input one:

class.java For each exported C++ class the Java file containing a matching Java class definition.

class.cpp C++ file implementing the native methods of the class above by forwarding all of its methods to the real class.

6.1 Backend-specific Output Options

In addition to the general options described in the section 4.2, Java backend provides the following ones:

`/Java_cpp:`*`path`* The path for the generated C++ files, by default it is the current directory.

`/Java_java:`*`path`* The path for the generated Java files, also current directory by default.

6.2 Packages

In Java, related classes are usually organized in packages and C++2Any has support for this. If the `PACKAGE` variable is defined (e.g. by using `/D` command line switch), the following changes occur:

- Each generated Java file is marked as being part of the package and each class is declared as `public`.
- `.java` files are created in the subdirectory with the same name as the package except that all dots are replaced by filesystem path separator (e.g. slash or backslash) and have the same name as the class they implement.

Please note that both of the above items are required by Java and can't be changed.

6.3 Enumerations

Java has no support for C `enum`s so their elements are represented, as it is custom in Java world, by public static final constants of an otherwise empty interface. Unfortunately, the C++ methods taking enum parameters only can be mapped to Java methods taking arbitrary integers resulting in a loss of type safety. If this is a serious problem justifying the overhead of the alternative solution, the templates may be modified to produce object wrappers for the enum elements instead of the simple constants.

Please note that there is a special rule for the names of the enum elements in Java code: it is a common practice to prefix the elements names with the name of the enum itself in C++ code, to avoid clashes with other identifiers and to make it clear which enum the element corresponds to, e.g.:

```
enum Pet
{
    Pet_Cat,
    Pet_Dog
};

...

enum UnixCommand
{
    UnixCommand_Cat,
    UnixCommand_Grep
};
```

However in Java the enum values are enclosed in an interface having the name of the enum itself and using `UnixCommand.UnixCommand_Cat` is unnecessarily verbose. Because of this, C++2Any automatically removes the prefix from enum elements in Java output so that just `UnixCommand.Cat` can be used instead.

If the enum elements prefix is not the same as the enum name, the special `enumprefix` comment can be used to indicate it:

```
/// @enumprefix Unix
enum UnixCommand
{
    Unix_Cat,
    Unix_Grep
};
```

And the same comment can be also used to prevent the automatic renaming of the enum elements from taking place at all by simply writing `enumprefix None` (provided that enum elements don't really start with `None`).

6.4 Limitations

Java imposes having a single class in a file being part of a package and so does C++2Any . As enums are represented as interfaces, this means that C++ headers can only have a single class or a single enum declared in them.

Other current limitations which may disappear in future versions:

- Default values for function parameters are not supported.
- Overloaded operators are not supported.
- No Unicode support, although it is available on Java side.

7 C Backend

This section describes how the generation of C wrappers for C++ code works.

7.1 Classes

As C doesn't have built-in support for obejct-oriented programming, we emulate C++ class methods using the C functions. For each class `ClassName` we introduce a special type representing it:

```
typedef struct ClassName_Dummy *CLASSNAME_HANDLE;
```

and also for constant pointers:

```
typedef const struct ClassName_Dummy *CONST_CLASSNAME_HANDLE;
```

which are passed as the first parameter to all wrapper functions of non-const and const, respectively, non-static class methods. Notice the use of opaque `ClassName_Dummy` type: this is used for class type checking at compilation time as the C compiler will generate warning for class type different from the type needed by the function.

To call a base class method for a child object user must pass to the wrapper function a converted object pointer. Child class to base class conversion functions are generated for all valid conversions by C++2Any .

7.2 Methods

The following code is generated for exported class. Each public method is wrapped into the function:

```
int NamespaceName_ClassName_MethodName(...);
```

Please note that from here on, the `NamespaceName_ClassName` part may be omitted in order to simplify the presentation.

The return value is always `int` which is used as the success/failure indicator as explained in section 7.7.

For static class members no `ClassHandle` parameter is generated. It is worth mentioning that if a class copy constructor and destructor are not declared, appropriate wrapper functions are still generated because a C++ compiler generates default realizations of these functions. Moreover if no constructor is defined, wrapper function for default constructor is generated. Wrapper functions for constructor have the following form:

```
int ClassName_Create(CLASSNAME_HANDLE* newobject, ...);
```

and for destructor:

```
int ClassName_Destroy(CLASSNAME_HANDLE* object);
```

7.3 Parameters and Return Values

Simple types are passed as the parameters in the same way as in C++ code. Complex types can be passed in three ways: by pointer, by reference and by value. Let's consider each of them. In case of passing parameters by pointer or by reference for functions:

```
func(..., SomeClass *a, ...);
func(..., SomeClass &a, ...);
```

the following wrapper is generated:

```
func(..., SOMECLASS_HANDLE a, ...);
```

In case of passing parameters by value for a function:

```
func(..., SomeClass a, ...);
```

the following function is generated:

```
func(..., CONST_SOMECLASS_HANDLE a, ...);
```

For passing such parameters as `const SomeClass`, `CONST_COMECLASS_HANDLE` is used.

For a function returning a value the last parameter is used for passing it to the caller. For simple types it should be a pointer to a variable of this type created by user, to which a return value will be assigned. Returning a reference is a special case. It's substituted by returning a pointer because there are no references in C language. Such parameter isn't generated for functions returning `void`. For example:

```
void func(...);
int intFunc(...);
float* floatprtFunc(...);
int& intrefFunc(...);
double SomeClass::doubleMethod(...);
```

are translated to:

```
int func(...);
int intFunc(..., int* result);
int floatprtFunc(..., float** result);
int intrefFunc(..., int** result);
int SomeClass_doubleMethod(SOMECLASS_HANDLE classHandle, ..., doubl
```

An interface to functions returning values of complex type is generated the same way except `SomeClass` is substituted by `SOMECLASS_HANDLE` and `const SomeSlass` – by `CONST_SOMECLASS_HANDLE`. User should delete temporary object that is created inside the wrapper function and returned if the original function returns object but not pointer or reference. This case will be pointed out in the comment to the generated prototype of wrapper function. For example:

```
SomeClass* ptrFunc(...);
SomeClass& refFunc(...);
SomeClass func(...);
```

are translated to:

```
int ptrFunc(...,SOMECLASS_HANDLE* result);
int refFunc(..., SOMECLASS_HANDLE* result);
// NOTE: this function returns a temporary object,
// be sure to destroy it yourself
int func(..., SOMECLASS_HANDLE* result);
```

7.4 Class Members

Warning: This section is not implemented yet.

For public class members the following functions are generated. For simple types:

```
class SomeClass { public: char m_Var; };
```

is translated to:

```
int Get_m_Var(CONST_SOMECLASS_HANDLE ClassHandle, char* Val);
int Set_m_Var(SOMECLASS_HANDLE ClassHandle, char Val);
```

For complex types only method `Get_m_Var()` is generated. For example:

```
class SomeClass { public: SomeClass2 m_Var; };
```

is translated to

```
int Get_m_Var(SOMECLASS_HANDLE ClassHandle, SOMECLASS2_HANDLE* Val)
```

7.5 Other Declarations

For typedefs and enums declared inside class declaration typedefs or enums named `NamespaceName_ClassName_TypedefName` or `NamespaceName_ClassName_EnumName` are generated. For the global typedefs and enums, the `ClassName` part is omitted.

In generated typedef complex types names are replaced by the appropriate `CLASS_HANDLEs`.

7.6 Templates

We are not supporting export of non-specified templates. So template support is limited to specified templates that are used in the part of C++ code that we are exporting. The Name of such class is constituted from template name and names of all its type parameters and values of constant parameters. For example:

```
SomeClass<char, 5>
```

will be translated into

```
SomeClass_char_5
```

7.7 Error Handling

As mentioned previously, all generated functions return an integer value which is zero if the function has succeeded and non-zero error code in case of error. In the latter case, the user can use the functions:

```
int GetErrorCode();
int GetErrorString(char* buf, int bufsize);
int GetExtendedErrorString(char* buf, int  bufsize);
```

for obtaining the last error code and error description string.

`buf` parameter of `GetErrrorString` and `GetExtendedErrorString` is an user-allocated buffer for a error description string (extended description), and `bufsize` parameter is the size of this buffer. If buffer size is less than needed then the string is truncated. The function returns the necessary buffer length.

Note that any (unhandled) C++ exceptions are translated to C error codes as well and that it is possible to return more rich error information to C code by using one's own exception classes.