USING EXCEPTIONS AS GOTO OPERATOR

Yuriy Solodkyy

August 8, 2005

Abstract

In this essay we'll try to go over a controversial issue of using exceptions mechanism in C++ as a flow control statement. This subject used to raise a lot of debates between supporters and opposers of the approach during author's work at Quadrox NV (Belgium), which makes it a good point for discussion. In this essay we'll try to present pros and cons of this approach and its alternative that doesn't use exceptions as well as propose a different solution that combines better of the two by employing standard language facilities of C++.

1 COM backgrounds

COM stands for Component Object Model and is a unified component model used in Windows operating systems family. It is based on C++ virtual calls dispatching mechanism and fixes binary layout of virtual tables in the object module so that applications that use COM can be written in languages other then C++. Together with its distributed counterpart DCOM it provides similar capabilities to what CORBA and other component models do. Since COM mechanisms are based on language constructs available in C++, for the sake of simplicity and without loosing much generality we'll further concentrate on COM applications written in C++.

COM interface is nothing else than an abstract class in C++, *all* of whose methods are pure virtual. Objects may support multiple interfaces, pointers to which can be obtained via explicit call to QueryInterface method of IUnknown interface from which all COM interfaces are derived. For further discussion it's also important to note that accordingly to the COM standard, *methods of any compliant COM interface may not throw exceptions*, as exceptions mechanism of C++ may not be supported by other languages, while those languages may still be used to write applications that use COM objects written in C++.

Example of interface:

```
class IUnknown {
public: // NOTE: Only public methods, all pure virtual, no nested types, no data members etc.
    virtual HRESULT QueryInterface(REFIID riid, void** ppvObject) throw() = 0;
    virtual ULONG   AddRef(void)  throw() = 0;
    virtual ULONG   Release(void) throw() = 0;
};

class IMyInterface : public IUnknown {
public:
    virtual HRESULT MyMethod1(int   n) throw() = 0;
    virtual HRESULT MyMethod2(char* s) throw() = 0;
    virtual HRESULT MyMethod3(IOtherInterface* pOther) throw() = 0;
    virtual long    MyMethod4(short h) throw() = 0;
};
```

There are some restrictions on data types that can be passed in and out of COM methods. While the exact principles can be found in [1] for this discussion we'll only note that structures and unions are typically passed by pointer, while basic data types are small enough to be passed by value. Since COM methods can not throw exceptions, in vast majority of cases method's return type will be HRESULT (typedefed to long), which represents

operation's status code. Roughly values greater or equal than 0 represent successful codes, while values less then 0 represent failure. COM headers provide two complementary macros SUCCEEDED() and FAILED(), that check value of type HRESULT for success or failure of operation. Any other result that has to be returned from a COM method is returned via passed in pointer.

2 Historical Backgrounds

To simplify work with COM, Microsoft created set of utility classes that can be split into 2 categories: those that assert upon invalid usage but still leave error handling to you and those that throw exceptions upon invalid usage and wrap COM error handling into C++ exceptions handling. First are part of Active Template Library (ATL) [2] and second are part of so called "Native COM support" [3].

ATL	Native COM Support	Description
`CComPtr<T>`	`_com_ptr_t<T>`	Smart pointer that handles automatic reference counting on interface
`CComQIPtr<T>`		Smart pointer that also adds automatic interface querying via member templates
`CComBSTR`	`_com_bstr_t`	Wrapper around COM string type BSTR
`CComVariant`	`_com_variant_t`	Wrapper around COM variant type VARIANT

Table 1: ATL and Native COM support classes

Example:

```
CComPtr<IMyInterface>    pMy1 = p; // p should be of type IMyInterface
CComQIPtr<IMyInterface>  pMy2 = p; // p may be of any interface type,
                                   // IMyInterface will be queried, 0 stored if not supported
_com_ptr_t<IMyInterface> pMy3 = p; // p may be of any interface type, IMyInterface will
                                   // be queried & exception thrown if not supported
pMy1->MyMethod2("a"); // will assert if underlain interface pointer is 0. pMy2 - the same
pMy3->MyMethod2("a"); // will throw  if underlain interface pointer is 0
```

To make things more uniform in the second approach designers added utility function CheckError that does the following:

```
void CheckError(HRESULT hr) {
    if (FAILED(hr))
        throw _com_error(hr);
}
```

It is this function that caused most of debates on whether it is appropriate or not to use it.

2.1 Sample code

The following sample code was found on the Internet by searching for words _com_utils::CheckError. We only list here it's short excerpt relevant to the discussion, while full source can be found at [4].

```
//this class encapsulates attachment and detachment of
//the dynamic properties
class PropertyAdmin
{
    static bool m_bInitialized;
    static AcRxClass* m_pClass;
    static CComObject<CSimpleProperty>* m_pSimple;
    static CComObject<CCategorizedProperty>* m_pCategorized;
    static CComObject<CEnumProperty>* m_pEnum;
```

```
public:
    static void initialize();
    static void uninitialize();
    static bool isInitialized() { return m_bInitialized;}
};

void PropertyAdmin::initialize()
{
    if (m_bInitialized)
        return;
    m_bInitialized = true;
    try
    {
        CComPtr<IPropertyManager> pPropMan;
        if ((pPropMan.p = GET_OPMPROPERTY_MANAGER(m_pClass))==NULL)
            _com_issue_error(E_FAIL);
        _com_util::CheckError(CComObject<CSimpleProperty>::CreateInstance(&m_pSimple));
        m_pSimple->AddRef();
        _com_util::CheckError(CComObject<CCategorizedProperty>::CreateInstance(&m_pCategorized));
        m_pCategorized->AddRef();
        _com_util::CheckError(CComObject<CEnumProperty>::CreateInstance(&m_pEnum));
        m_pEnum->AddRef();
        _com_util::CheckError(pPropMan->AddProperty(m_pSimple));
        _com_util::CheckError(pPropMan->AddProperty(m_pCategorized));
        _com_util::CheckError(pPropMan->AddProperty(m_pEnum));
    }
    catch(const _com_error& )
    {
        uninitialize();
        acutPrintf("\nSimpleDynProps: initialize failed!!!\n");
        return;
    }
}

void PropertyAdmin::uninitialize()
{
    if (!m_bInitialized)
        return;
    m_bInitialized = false;
    try
    {
        CComPtr<IPropertyManager> pPropMan;
        if ((pPropMan.p = GET_OPMPROPERTY_MANAGER(m_pClass))==NULL)
            _com_issue_error(E_FAIL);
        if (m_pSimple)
        {
            _com_util::CheckError(pPropMan->RemoveProperty(m_pSimple));
            m_pSimple->Release();
        }
        if (m_pCategorized)
        {
            _com_util::CheckError(pPropMan->RemoveProperty(m_pCategorized));
```

```
            m_pCategorized->Release();
        }
        if (m_pEnum)
        {
            _com_util::CheckError(pPropMan->RemoveProperty(m_pEnum));
            m_pEnum->Release();
        }
    }
    catch(const _com_error& )
    {
        acutPrintf("\nSimpleDynProps: uninitialize failed!!!\n");
    }
}
```

This is probably not the best example for the purpose of our discussion as it has many other problems, however those probably can be seen as lack of clarity on COM support designers' side as way to many people make similar mistakes. Some of those problems are:

- Non employing RAII principle for member allocation. This partially is related to the fact that while designers provided smart pointers for interface management, they didn't provide such pointers for direct object management, which sometimes is needed (when COM object is written in ATL too and doesn't need external creation capabilities).
- Mixing both ATL and native COM support together

The actual point of discussion is employing the following pattern, which is clearly seen in the sample too:

```
HRESULT CMyCOMObject::MyMethod3(IOtherInterface* pOther)
{
    try
    {
        if (something_is_wrong_with(pOther))
            _com_issue_error(E_INVALIDARG);
        if (something_is_wrong_with(this))
            _com_issue_error(E_UNEXPECTED);

        _com_util::CheckError(pOther->Method1(3.14));
        _com_util::CheckError(MyMethod1(42));
        DoSomeOtherComputations();
        _com_util::CheckError(pOther->Method2("aaa"));
        _com_util::CheckError(MyMethod2("bbb"));
    }
    catch(const _com_error& e)
    {
        return e.Error();
    }

    return S_OK;
}
```

Function _com_issue_error(HRESULT) throw(_com_error) here unconditionally throws _com_error initialized with given HRESULT value. As can be seen exceptions are thrown and caught in the same function just for the purpose of redirecting flow to the end of the function where error code is returned. None of other exceptions are typically caught, none are rethrown and as in the sample code above even the error code is ignored. Typical unit

of work, wrapped into a method or free function will contain from 5 to 15 calls to COM methods, all wrapped into call to CheckError() and guarded by a try-catch block that only catches what it throws: _com_error. Error code of the failed call will be returned as a result of the combined function.

We believe the reason this approach became so widespread lays in default parameters (which not many were bothering to change) of import directive that was translating interface definitions from IDL to C++. These defaults were oriented for dispatch interfaces and were generating higher-level wrapper methods apart from pure interface methods. Those were returning type of outgoing parameters instead of HRESULT while throwing exceptions upon errors. As a result, class that represented imported interface had a mix of similar methods some of which were throwing, while some were not and not every user of the imported interface understood the difference. Eventually, to make things more predictable and not think too much on whether particular call should be wrapped into a try-catch block or simply if-statement, people began to use CheckError for all calls just to make sure it will throw on error if original method doesn't.

2.2 Pros and cons

The pluses of this approach can be summarized as following:

- Developer don't have to thoroughly check the outcome of each call, all the following functionality of the block will be canceled once the error was detected. The workflow will also be automatically redirected to a given place in the function.
- Debugger steps via one statement at a time. This seems to be a minor issue at first, but in practice this was the main reason of abandoning other approaches.
- Uniformness - your code is less error prone to mistakes due to improper error checking and control flow redirection.
- Local variables can be declared at the point of first use.

There are quite a few negative sides though:

- Since methods of COM interfaces can not throw accordingly to COM specifications, we are bringing exceptions into domain that doesn't use them on its own.
- No extra information about the context of an error is available, at the end you will be left with the same HRESULT code you would have had with manual error handling. One may argue though that CheckError can be extended to accept a string or resource handle with error description should failure happen, however this would increase even more performance overhead of each call.
- Significant performance losses in those cases when call to method may fail in regular situations (e.g. timeout waiting for synchronization object to become available, which happens quite often in video processing for example). In this case it also significantly obscures debugging as debuggers typically allow you to break on C++ exception without possibility of specifying on which. As a result, you wait for a particular exception to happen but is kicked into debugger on regular intervals for something not really exceptional.
- Increase in code size because of necessity to put exception frames. [5]
- Requires all the methods that use CheckError to have try-catch block around them. When this is not followed thoroughly (e.g. when methods are split into helper methods that don't have to do that and interface methods that do) it can potentially create non-compliant COM interface implementations that throw exceptions. This can be especially important when copy-pasting is used in favor of proper code reusage.

3 Alternatives

While question of whether exceptions are used properly in this scenario is still debatable, everybody seems to agree that usage of "goto" operator here is not a good idea. At least we've never seen it in the books on COM, numerous samples and real code we had a chance to work with. However usage of exceptions in the above scenario is nothing else then hiding of goto-statement behind a curtain of advanced language possibilities.

Thorough look at what developers would actually like to achieve in the above scenario can be summarized as following:

- Execute chain of interface calls while they succeed.
- Interrupt chain if any of the calls fails.
- Return error code of failed operation as a status code of the whole operation.

C and C++ provide means for the above chaining via && operator, combining it with nested assignment provides us a simple alternative solution:

```
HRESULT CMyCOMObject::MyMethod3(IOtherInterface* pOther)
{
    HRESULT hr = something_is_wrong_with(pOther)
                    ? E_INVALIDARG
                    : something_is_wrong_with(this)
                        ? E_UNEXPECTED
                        : S_OK;
    SUCCEEDED(hr)                           &&
    SUCCEEDED(hr = pOther->Method1(3.14))   &&
    SUCCEEDED(hr = MyMethod1(42))           &&
    (DoSomeOtherComputations(),true)        &&
    SUCCEEDED(hr = pOther->Method2("aaa")) &&
    SUCCEEDED(hr = MyMethod2("bbb"));
    return hr;
}
```

3.1 Pros and Cons

Approach has very few pluses in comparison to the solution with exceptions, but those are typical debate points in discussions on which of readability, efficiency or robustness of code are more import.

- No exception handling overhead, code is as efficient as it would have been with explicit flow redirection.
- The whole chain can be put into the condition of if-statement, splitting effectively logic into parts. CheckError approach has to treat such situation by non-wrapping call into a CheckError and making sure to call a raw interface method rather then a wrapped one that throws.
- Since all calls still have to be wrapped into another call (SUCCEEDED here, but it can be something like DBG_SUCCEEDED or ASSERTIVE_SUCCEEDED) approach can greatly assist in debugging: via macro substitutions in debug builds the exact place of the failure can be dumped into the log, immediately broken into debugger or asserted upon.

And quite some negative sides of it just add points to the other approach:

- The biggest drawback of this approach for which people were refusing to use it was the fact that all the calls in a chain were part of the same statement and majority of debuggers use statement as a unit of advancing through code. This means that if you know that call to MyMethod2 in the above chain fails and you want to figure out why, you need to step in and out into all the previous calls as well as calls to constructors and

functions used to compute actual parameters of them before you can get into MyMethod2. Note that simply putting breakpoint in MyMethod2 is not always an option since there may be multiple calls to MyMethod2 in the chain or just because it takes much more time to locate a method in code and put a breakpoint there than to step over a few calls before it.

- Since SUCCEEDED is defined as a macro, code relies on the fact that parameter will be used only once in substitution. This can be easily fixed by creating a similar function, though people often prefer well known macro to a safe proprietery function. The last one can also help with information about the context of an error similarly to the reasoning that was given in CheckError approach.
- Often times calls to COM interface methods has to be interleaved with calls to non COM related routines, which breaks the chain similarly to how DoSomeOtherComputations() did in the example above.
- Variables should be declared before the chain and not at the point where they are first needed.

Minor improvement of this approach that was primarily targeting debugger problem made code more obscure and didn't gain many new supporters:

```
SUCCEEDED(hr) && SUCCEEDED(hr = pOther->Method1(3.14));
SUCCEEDED(hr) && SUCCEEDED(hr = MyMethod1(42));
DoSomeOtherComputations();
SUCCEEDED(hr) && SUCCEEDED(hr = pOther->Method1("aaa"));
SUCCEEDED(hr) && SUCCEEDED(hr = MyMethod2("bbb"));
```

4 The power of C++

Ideally we want to be able to write code similar to the following:

```
hr = pOther->Method1(3.14);
hr = MyMethod1(42);
DoSomeOtherComputations();
hr = pOther->Method1("aaa");
hr = MyMethod2("bbb");
```

but have all the subsequent calls canceled once FAILED(hr) becomes true. To make this possible we first need to find a way of canceling a particular call based on certain condition.

Overloading `operator->` won't help us in this because it doesn't give us control over actual call [6]. Nevertheless C++ has a solution for this, which is often forgotten for some reason. `operator->*` can also be overloaded in C++ but unlike `operator->` it is a binary operator that accepts pointer to member as its second argument. Having that we can make it return a functor, whose `operator()` will later do actual call. To conserve space we'll only present here the case for handling methods with 2 arguments, though appropriate support for different (predefined) amount of arguments can easily be added.

```
template <class T, class R, class A1, class A2>
class call2
{
public:
    call2(T* p, R (__stdcall T::*method)(A1, A2)) : m_p(p), m_method(method) {}
    R operator()(A1 a1, A2 a2) { return (m_p->*m_method)(a1,a2); }
protected:
    T* m_p;
    R (__stdcall T::*m_method)(A1,A2);
};
```

```
template <class T, class R, class A1, class A2>
class call2<const T, R, A1, A2>
{
public:
    call2(const T* p, R (__stdcall T::*method)(A1,A2) const) : m_p(p), m_method(method) {}
    R operator()(A1 a1, A2 a2) const { return (m_p->*m_method)(a1,a2); }
protected:
    const T* m_p;
    R (__stdcall T::*m_method)(A1,A2) const;
};

template <class T>
class dispatcher
{ // Possibilities provided by CComPtr et al are omitted but assumed
public:
    explicit dispatcher(T* p) : m_p(p) {}

    template <class R, class A1, class A2>
    call2<T, R, A1, A2> operator->*(R (__stdcall T::*method)(A1,A2)) {
        return call2<T,R,A1,A2>(m_p, method);
    }

    template <class R, class A1, class A2>
    call2<const T, R, A1, A2> operator->*(R (__stdcall T::*method)(A1,A2) const) {
        return call2<const T,R,A1,A2>(m_p, method);
    }
private:
    T* m_p;
};
```

which would allow us to make calls as following:

```
dispatcher<IMyInterface> p = get_my_interface_from_somewhere();
HRESULT h = (p->*&IMyInterface::Method1)(19);
long    l = (p->*&IMyInterface::Method4)(17);
```

Specialization of call2 template for const T is needed to be able to handle calls to const members, though because const methods are rarely used in COM interfaces, we'll skip cases for their treatment from the following explanations just to conserve space. They can be easily added by analogy if needed. MSVC compiler's keyword __stdcall designates FORTRAN calling convention used in COM.

Since `operator=` in C++ has lower precedence then `operator()`, while we want to be able to cancel computation based on current value of hr, we need to defer actual computation for one more step to assignment operator. For this we are going to return a proxy object as a result of `call2<T>::operator()` capable of executing initial call when casted to initial result type. The adequate changes will be:

```
template <class T, class R, class A1, class A2> class apply2;

template <class T, class R, class A1, class A2>
class call2
{
public:
    call2(T* p, R (__stdcall T::*method)(A1, A2)) : m_p(p), m_method(method) {}
    apply2<T,R,A1,A2> operator()(A1 a1, A2 a2) {
        return apply2<T,R,A1,A2>(m_p,m_method,a1,a2);
```

```
    }
protected:
    T* m_p;
    R (__stdcall T::*m_method)(A1,A2);
};

template <class T, class R, class A1, class A2>
class apply2 : public call2<T,R,A1,A2>
{
    typedef call2<T,R,A1,A2> base;
public:
    apply2(T* p, R (__stdcall T::*method)(A1,A2), A1 a1, A2 a2) :
        base(p,method), m_a1(a1), m_a2(a2) {}
    operator R() const { return (m_p->*m_method)(m_a1, m_a2); }
private:
    A1 m_a1;
    A2 m_a2;
    using base::m_p;
    using base::m_method;
};
```

With this in hand the task of canceling call based on current value of hr becomes rather trivial:

```
class CComResult
{
public:
    CComResult(HRESULT hr = S_OK) : m_hr(hr) {}
    operator HRESULT() const { return m_hr; }
    operator    bool() const { return SUCCEEDED(m_hr); }

    template <class T, class A1, class A2>
    CComResult& operator=(const apply2<T,HRESULT,A1,A2>& apl)
    {
        if (SUCCEEDED(m_hr))
            m_hr = static_cast<HRESULT>(apl);
        return *this;
    }
private:
    HRESULT m_hr;
};
```

Note, we don't parameterize `CComResult::operator=` on the return type to make sure that this functionality will only be used for COM interface methods that return HRESULT. We left parameterization on return type in dispatcher, call and apply template classes simply to allow customizations of this approach for return types with other interpretations. The above example can now be rewritten as following:

```
HRESULT CMyCOMObject::MyMethod3(IOtherInterface* pOther)
{
    CComResult hr = something_is_wrong_with(pOther)
                        ? E_INVALIDARG
                        : something_is_wrong_with(this)
                            ? E_UNEXPECTED
                            : S_OK;
    hr = (pOther->*&IOtherInterface::Method1)(3.14);
```

```
    hr = (this->*&IMyInterface::MyMethod1)(42);
    if (hr) DoSomeOtherComputations();
    hr = (pOther->*&IOtherInterface::Method2)("aaa"));
    hr = (this->*&IMyInterface::MyMethod2)("bbb"));
    return hr;
}
```

There is a small problem we forgot to take into account here. What if user will decide to not write result of a particular call into hr, but still perform call via `operator->*`? This may happen for example when the outcome of a particular call is not important and shouldn't affect the final result. In such scenario it's not difficult to imagine that assignment was there before developer decided that this call shouldn't affect the result, but then was simply deleted without rewriting call through `operator->`. This can easily be fixed by letting apply2 template class monitor if the call was actually made and if not, make it inside apply2's destructor when returned temporary will be destroyed.

```
template <class T, class R, class A1, class A2>
class apply2 : public call2<T,R,A1,A2>
{
    typedef call2<T,R,A1,A2> base;
public:
    apply2(T* p, R (__stdcall T::*method)(A1,A2), A1 a1, A2 a2) :
        base(p,method), m_a1(a1), m_a2(a2), m_bApplied(false) {}
   ~apply2() {
        if (!m_bApplied && !std::uncaught_exception())
            (m_p->*m_method)(m_a1, m_a2);
    }
    operator R() const { m_bApplied = true; return (m_p->*m_method)(m_a1, m_a2);
}
private:
    A1 m_a1;
    A2 m_a2;
    mutable bool m_bApplied;
    using base::m_p;
    using base::m_method;
};
```

Another hidden problem that should be explicitly warned from is usage of the above machinery with interface methods that return long. Type HRESULT in Windows is just a typedef for long and since typdef declaration in C++ introduces synonym rather then a distinct type, while there is a semantic difference between these types there is no way for compiler to distinguish them. We took precaution to make sure that interface methods that don't return HRESULT wouldn't be allowed to be called via guarded assignment to CComResult, however this won't prevent developer from calling methods that return long, which is semantically different from HRESULT.

This problem can be solved partially for those cases when signature of the method returning long is different from signature of any other method in the same interface. In such cases author of the interface may non-intrusively define `operator->*` (which unlike `operator->`, `operator->*` can be declared outside of class in C++) that would return a non-callable object. The following will disable call through guarded assignment to IMyInterface::Method4, which wouldn't have been possible if IMyInterface::Method1 accepted short rather then int:

```
void operator->*(dispatcher<IMyInterface>&, long (__stdcall IMyInterface::*method)(short));
```

Returning a non-callable object (void here) is important to generate a compilation error and force user to rewrite that piece. Type void can be replaced with a dedicated type, name of which hints about improper usage. We refer to this here only as a potential use-case scenario for oncoming static_assert addition to the language.

Unfortunately there are many existing interfaces with methods returning long that don't fall into the mentioned category and they can potentially become a source of errors. It's also interesting to note that two most used interface methods in COM that do not return an HRESULT, namely IUnknown::AddRef and IUnknown::Release, return unsigned long and hence don't suffer from this problem.

You may have noticed that we've been passing all the parameters by value. This was done on purpose to avoid discussing here issues related to forwarding problem described in [7]. While this is certainly not acceptable for general case, we may afford it in this domain cause as we mentioned before complex data types in COM are passed by pointer, while basic ones are small enough to be passed by value. References are not supported by COM.

4.1 Pros and Cons

Pluses of the approach are now gaining points, lost by its simpler alternative that didn't use exceptions:

- No exception handling overhead and with proper inlining support the code is as efficient as it would have been with explicit flow redirection.
- Debugger steps via one statement at a time, though at the statement you are interested in you'll have to step in and out 3 intermediates before getting into the callee.
- Wrapping logic can be extended to do some more complicated manipulations with resulting codes. For example often times we would like to receive a warning (which are also success HRESULT codes in COM, with certain bit set up) if any of the chain operations returned it, rather then have S_OK code of the last operation in the chain. Another obvious application is debugging support mentioned before, though because no macros are involved, we wouldn't be able to provide as detailed information about the place where it happened as before. Breaking into debugger upon error should work equally well.

Obviously approach doesn't solve everything and has negative sides as well:

- Even though variables can be declared at the point of first use, their constructors will be executed even if one of the preceding calls on COM interface method failed.
- Non-COM calls has to be explicitly wrapped into if-statement. We can write a similar call wrapper for those, however this would probably make syntax more obscure then explicit if-statement.
- The logic behind assignments is less obvious and new user might become mislead by that. We believe this can be fixed by replacing `operator=` with some other operator that would underlain this difference. Some of the good candidates would be `operator<<` and `operator|`.
- Requires extra care of interface methods that return long that is semantically different from HRESULT.
- Changes in calling syntax.

5 Performance results

To test performance of different approaches, a small application was written using 3 different approaches, that was calling in a loop function `test<n>()` presented here from Dispatch approach.

```
template <int n>
HRESULT test()
{
    CComPtr<IMemAllocator> pMemAllocPtr;
    CComResult hr = CoCreateInstance(CLSID_MemoryAllocator,
                                     0,
                                     CLSCTX_INPROC_SERVER,
```

```
                                    IID_IMemAllocator,
                                    reinterpret_cast<void**>(&pMemAllocPtr));
    dispatcher<IMemAllocator> pMemAllocDsp(pMemAllocPtr);
    ALLOCATOR_PROPERTIES props = {2, 768*576*3, 4, 0};
    ALLOCATOR_PROPERTIES actual;

    hr = (pMemAllocDsp->*&IMemAllocator::SetProperties)(&props, &actual);
    hr = (pMemAllocDsp->*&IMemAllocator::GetProperties)(&props);
    hr = (pMemAllocDsp->*&IMemAllocator::Commit)();

    CComPtr<IMediaSample> pMediaSamplePtr;
    REFERENCE_TIME rtStart;
    REFERENCE_TIME rtEnd;

    if (hr) DoSomething();
    hr = (pMemAllocDsp->*&IMemAllocator::GetBuffer)(&pMediaSamplePtr, &rtStart, &rtEnd, 0);

    dispatcher<IMediaSample> pMediaSampleDsp(pMediaSamplePtr);

    hr = (pMediaSampleDsp->*&IMediaSample::IsSyncPoint)();
    long l = (pMediaSampleDsp->*&IMediaSample::GetActualDataLength)();
    l++;
    pMediaSamplePtr.Release();
    hr = (pMemAllocDsp->*&IMemAllocator::Decommit)();
    test<n-1>();
    return hr;
}
```

This is not a code taken from a real example, but instead it is a synthesized code that resembles typical usage of some standard COM objects used in DirectShow applications [8]. As can be seen it has a typical chain of COM calls discussed before.

Function was parameterized for int parameter to see the increase in code size in different approaches. It is this parameterization that caused some difficult to explain results at first, however later on it was clear that unusual behavior was due to a big number of its instantiations (around 1000). In particular with such amount of instantiations CheckError approach was several times faster then Dispatch and Succeeded approaches and it seemed that Microsoft provided some knowledge to the compiler about CheckError approach (word "native" in "Native COM support" was making this assumption stronger). To find out whether this was actually the case, a minimal set of classes from Native COM Support classes used by this application was ported to work with GCC 3.4.4, while actual COM objects used here were replaced with regular C++ objects that implement appropriate interfaces.

The following table shows increase in percentage relatively to the fastest time for different approaches and test cases.

	No errors CheckError	No errors Dispatch	No errors Succeeded	W/ errors CheckError	W/ errors Dispatch	W/ errors Succeeded
GCC 3.4.4/Linux		+0.21%	+1.12%	+3.31%		+0.46%
GCC 3.4.4/Windows (CygWin)		+0.07%		+2.97%	+0.22%	
MSVC 7.1 with code for GCC		+0.12%	+0.01%	+0.78%	+0.14%	
MSVC 7.1 with actual code	+7.99%	+0.91%		+201.47%		+163.27%

Table 2: Performance results

As can be seen CheckError approach was faster on synthesized code when no errors occurs, though it was slower with actual code. Dispatch approach was slower in most cases without errors, but not more then 1%.

In case when one of the operations in chain was failing, Dispatch approach was clearly faster with actual code and nearly as fast with synthesized one. The fact that Succeeded approach was more then 1.5 times slower then Dispatch in the last test case remains unclear and will have to be checked more thoroughly on whether test sets were the same in all the cases.

As to binary size, Succeeded approach was clearly always a winner, followed by CheckError approach (+5% for 100 instantiations and +9% for 200 instantiations) and then by Dispatch (+45% and +69% respectively).

6 Conclusions

In presented essay we tried to analyze alternatives to an approach of using exceptions as a goto operator in some COM applications. Presented solution clearly outperform exceptions approach in case when errors are expected, while maintaining almost optimal performance when errors are rare. Unfortunately because of platform dependence we couldn't perform exactly the same tests with different compilers and results therefore are quite different for modeled and actual code. This can possibly be improved by further tests should this essay have any rational in it.

References

[1] Microsoft Corporation. Midl data types. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_data_types.asp, 2002.

[2] Richard T. Grimes. *Professional ATL COM Programming*. Wrox Press, September 1998. ISBN 1-861001-40-1.

[3] Microsoft Corporation. Native com support. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_compiler_com_support.3a_.overview.asp, 2002.

[4] Inc Autodesk. Dynamic properties sample. http://www.csf.ru/ftp/autodesk/OARX/ObjectARX_2004_sdk/samples/editor/simpled 2002.

[5] Vishal Kochhar. How a c++ compiler implements exception handling. http://www.codeproject.com/cpp/exceptionhandler.asp, April 2002.

[6] Bjarne Stroustrup. Wrapping c++ member function calls. *C++ Report*, 12(6), June 2000.

[7] Peter Dimov, Howard Hinnant, and Dave Abrahams. The forwarding problem: Arguments. In *ANSI/ISO C++ Standard Committee Pre-Santa Cruz mailing*, number N1385=02-0043 in ANSI/ISO C++ Standard Committee Pre-Santa Cruz mailing, October 2002.

[8] Michael Linetsky. *Programming Microsoft DirectShow*. Wordware Publishing, October 2001. ISBN 1-556-22855-4.