Easy Binary Compatible C++ Interfaces Across Compilers

John R. Bandela, MD

Google

old new thing

Web | Images | Maps | Shopping | More | Search tools

About 1,700,000,000 results (0.20 seconds)

The Old New Thing - Site Home - MSDN Blogs
blogs.msdn.com/b/**oldnewthing**/
Weblog on technical nitty-gritty, with a Microsoft focus.

- **If the shell is written in C++, why**
  If the shell is written in C++, why not just export its base ...
- **How to insert a large number of**
  How to insert a large number of items into a treeview ...
- **WinMain is just the ...**
  WinMain is just the conventional name for the Win32 process ...
- **Don't try to allocate memory ...**
  Don't try to allocate memory until there is only x% free ...
- **Why was Pinball removed**
  [Pinball was licensed from another company, so you'll have to ask ...
- **How did my hard drive turn ...**
  If a network drive wants to report that it is a TARDIS, then it's a ...

More results from msdn.com »

What is the problem

- No easy way to share a C++ class compiled with compiler A with compiler B
- Often have to rebuild the component if changing compilers and/or standard library
- “The biggest fault in C++, for my money, is the lack of a standard for the ABI. As a direct consequence C++ code compiled with GCC won't work with C++ code compiled with MSVC unless you provide a 'C' interface (using extern “C” linkage). This forces all C++ providers looking to provide generic, reusable code to use 'C' linkage at some point.” – Unnamed friend of Matthew Wilson, ex-friend of C++ from *Imperfect C++*

What are we going to cover in this talk

- Why calling C++ code across compilers is hard
- How we currently can call C++ code across compilers
- How to make it easier to define, implement, and use interfaces that work across compilers
- What are some of the library features and how are they used and implemented
- What is my vision for the future in terms of these techniques

- Code available at https://github.com/jbandela/cross_compiler_call

- Note: This is an interactive talk, so please feel free to interrupt and ask questions or questions answers
- A lot the background comes from *Imperfect C++* by Matthew Wilson chapters 7-8 and from *Inside Ole* by Kraig Brockschmidt

Why is it hard

- Common to C and C++
  - Calling conventions
  - Structure packing
- C++
  - Name mangling
  - Virtual function implementation
  - RTTI
  - Exception handling
  - Standard library implementation

How we share with C

- Calling conventions
  - Specifies how arguments and return values are handled, who cleans up stack, what registers are used for what
  - Can often be handled with a platform specific #define, for example on Windows
    - `#define CALLING_CONVENTION __stdcall`
    - `HKVStore CALLING_CONVENTION Create_KVStore();`

- Structure packing
  - Compiler is allowed to insert padding in structures
  - Can use compilers specific pragma's and keywords to control the packing

How do we share C++

- “Extern C”
- Compiler generated vtable
- Programmer generated vtable

A simple motivating example

```
struct KVStore{
    KVStore();
    ~KVStore();
    void Put(const std::string& key, const std::string& value);
    bool Get(const std::string& key, std::string* value);
    bool Delete(const std::string& key);
};
```

Extern C

- Use `extern "C"` to avoid C++ name mangling
- Then unpack each of our public member functions into global functions that take an opaque pointer.

Extern C interface

```
struct KVStore;
typedef KVStore* HKVStore;
using std::int32_t;
typedef std::int32_t error_code;
#define CALLING_CONVENTION __stdcall
extern "C"{
    HKVStore CALLING_CONVENTION Create_KVStore();

    void CALLING_CONVENTION Destroy_KVStore(HKVStore h);

    error_code CALLING_CONVENTION Put (HKVStore h,const char* key, int32_t
      key_count,const char* value, int32_t value_count);

    error_code CALLING_CONVENTION Get(HKVStore h, const char* key, int32_t
      key_count,const char** pvalue,int32_t* pvalue_count,char* breturn);

    error_code CALLING_CONVENTION Delete(HKVStore h, const char* key, int32_t
      key_count,char* breturn);
}
```

Extern C implementation

```
// Extern C
struct KVStore{
    std::map<std::string,std::string> m_;
};


extern "C"{
    HKVStore CALLING_CONVENTION Create_KVStore(){
        try{
            return new KVStore;
        }
        catch(std::exception&){
            return nullptr;
        }
    }

    void CALLING_CONVENTION Destroy_KVStore(HKVStore h){
        delete h;
    }
```

Extern C Implementation Continued

```
error_code CALLING_CONVENTION Put (HKVStore h,const char* key, int32_t
      key_count,const char* value, int32_t value_count){
        try{
            std::string key(key,key_count);
            std::string value(value,value_count);
            h->m_[key] = value;
            return 0;
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Extern C Implementation Continued

```
error_code CALLING_CONVENTION Get(HKVStore h, const char* key, int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn){
        try{
            std::string key(key,key_count);
            auto iter = h->m_.find(key);
            if(iter == h->m_.end()){
                *breturn = 0;
                return 0;
            }
            else{
                std::string value = iter->second;
                auto pc = new char[value.size()];
                std::copy(value.begin(),value.end(),pc);
                *pvalue_count = value.size();
                *pvalue = pc;
                *breturn = 1;
                return 0;
            }
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Extern C Implementation Final

```
error_code CALLING_CONVENTION Delete(HKVStore h, const char* key, int32_t
        key_count,char* breturn){
        try{
            std::string key(key,key_count);
            auto iter = h->m_.find(key);
            if(iter == h->m_.end()){
                *breturn = 0;
                return 0;
            }
            else{
                h->m_.erase(iter);
                *breturn = 1;
                return 0;
            }
        }
        catch(std::exception&){
            return -1;
        }
    }
}
```

Extern C usage

```
auto kv = Create_KVStore();
std::string key = "key";
std::string value = "value";
Put(kv,key.data(),key.size(),value.data(),value.size());

const char* pvalue = nullptr;
int32_t count = 0;
char b = 0;

Get(kv,key.data(),key.size(),&pvalue,&count,&b);

std::cout << "Value is " << std::string(pvalue,count) << "\n";

Delete(kv,key.data(),key.size(),&b);

Destroy_KVStore(kv);
```

Review of extern “C”

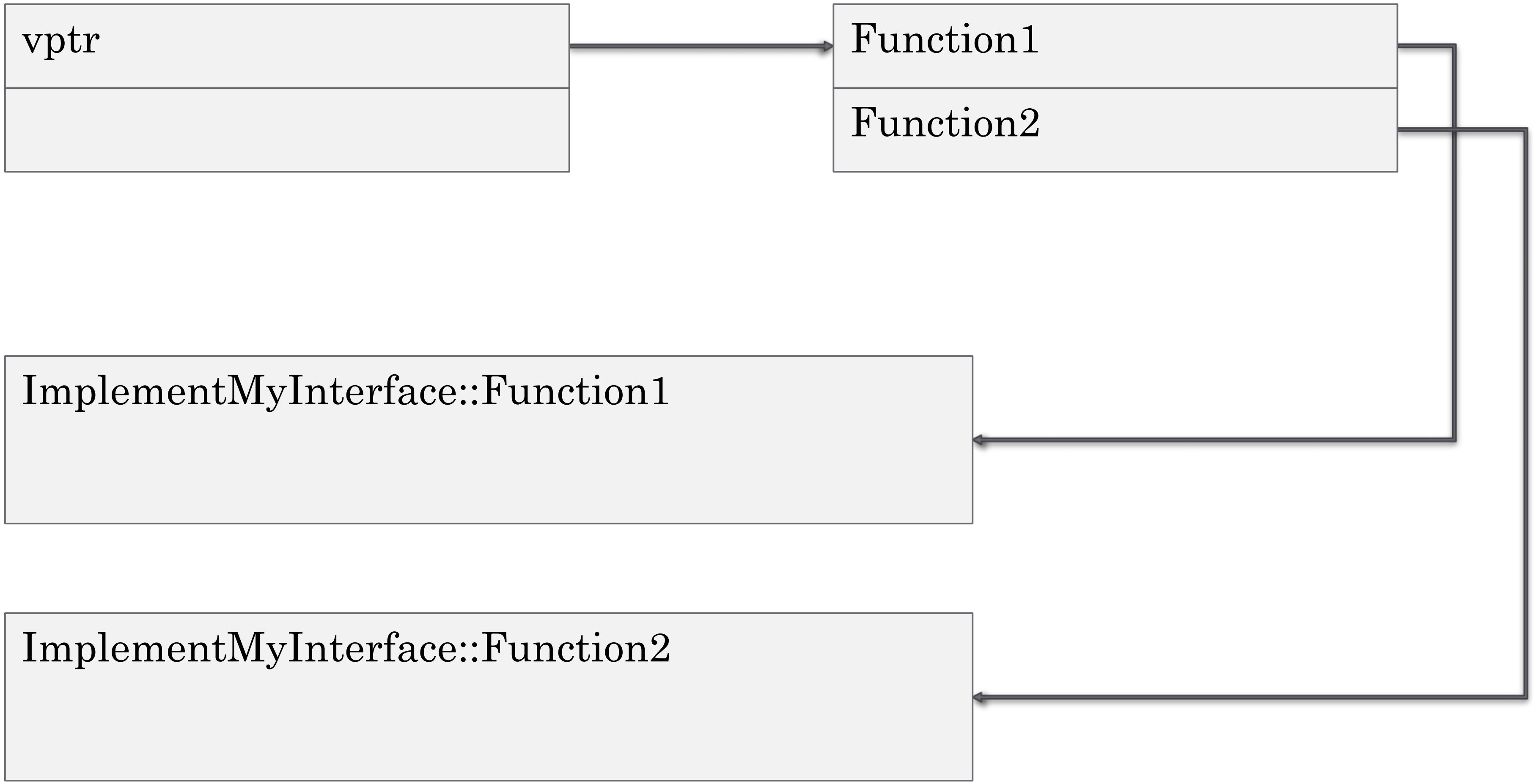
- Can be used on multiple C++ compilers
- Can even be called from C
- Biggest problem is that you lose polymorphism
  - For example if you implemented hierarchical storage on top of our key-value store, how would you be able to use multiple implementations?
  - For that we need some type of object

Compiler generated vtable

- Takes advantage that many compilers transform the following to …

```cpp
struct IMyInterface{
    virtual void Function1() = 0;
    virtual void Function2() = 0;
};

class ImplementMyInterface:public IMyInterface{
    void Function1(){
        // Implementation
    }

    void Function2(){
        // Implementation

    }

};
```

Compiler generated vtable



Compiler generated vtable interface

```cpp
struct IKVStore{
    virtual error_code CALLING_CONVENTION Put (const char* key, int32_t key_count,const char* value, int32_t value_count) = 0;
    virtual error_code CALLING_CONVENTION Get(const char* key, int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn) = 0;
    virtual error_code CALLING_CONVENTION Delete(const char* key, int32_t key_count,char* breturn)=0;
    virtual void CALLING_CONVENTION Destroy() = 0;
};
```

Implementation

```
struct KVStoreImplementation:public IKVStore{
    std::map<std::string,std::string> m_;

    virtual error_code CALLING_CONVENTION Put (const char* key, int32_t key_count,const char* value, int32_t value_count) override{
        try{
            std::string key(key,key_count);
            std::string value(value,value_count);
            m_[key] = value;
            return 0;
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Implementation continued

```
virtual error_code CALLING_CONVENTION Get(const char* key, int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn) override{
        try{
            std::string key(key,key_count);
            auto iter = m_.find(key);
            if(iter == m_.end()){
                *breturn = 0;
                return 0;
            }
            else{
                std::string value = iter->second;
                auto pc = new char[value.size()];
                std::copy(value.begin(),value.end(),pc);
                *pvalue_count = value.size();
                *pvalue = pc;
                *breturn = 1;
                return 0;
            }
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Implementation Final

```
 virtual error_code CALLING_CONVENTION Delete(const char* key, int32_t key_count,char* breturn)override{
        try{
            std::string key(key,key_count);
            auto iter = m_.find(key);
            if(iter == m_.end()){
                *breturn = 0;
                return 0;
            }
            else{
                m_.erase(iter);
                *breturn = 1;
                return 0;
            }
        }
        catch(std::exception&){
            return -1;
        }
    }

    virtual void CALLING_CONVENTION Destroy() override {
        delete this;
    }
};
```

Getting the interface

```
extern "C"{
    IKVStore* CALLING_CONVENTION Create_KVStoreImplementation(){
        try{
            return new KVStoreImplementation;
        }
        catch(std::exception&){
            return nullptr;
        }
    }
}
```

Usage

```
auto ikv = Create_KVStoreImplementation();
std::string key = "key";
std::string value = "value";
ikv->Put(key.data(),key.size(),value.data(),value.size());

const char* pvalue = nullptr;
int32_t count = 0;
char b = 0;

ikv->Get(key.data(),key.size(),&pvalue,&count,&b);

std::cout << "Value is " << std::string(pvalue,count) << "\n";

ikv->Delete(key.data(),key.size(),&b);

ikv->Destroy();
```

Programmer generated vtable

- The compiler generated vtable has polymorphism – You can pass an interface from one dll to another dll that expects that interface
- The weakness of the above technique is that you are depending on a compiler transformation
- The solution to this is to manually specify the vtable as a struct containing function pointers instead of relying on the compiler
- This technique is described in *Inside Ole* by Kraig Brockschmidt as a technique to define interfaces in C, and in *Imperfect C++* by Matthew Wilson as a technique to get around depending on the C++ compiler to generate the same structure as another compiler

Interface

```
struct IKVStore2;

struct IKVStoreVtable{
    error_code (CALLING_CONVENTION * Put) (IKVStore2* ikv, const char* key,
      int32_t key_count,const char* value, int32_t value_count);

    error_code (CALLING_CONVENTION *Get)(IKVStore2* ikv, const char* key,
      int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn);

    error_code (CALLING_CONVENTION *Delete)(IKVStore2* ikv, const char* key,
      int32_t key_count,char* breturn);

    void (CALLING_CONVENTION *Destroy)(IKVStore2* ikv);
};

struct IKVStore2{
    IKVStoreVtable* vtable;
};
```

Implementation

```
struct KVStore2Implementation:public IKVStore2{
    std::map<std::string,std::string> m_;
    IKVStoreVtable vt;

    KVStore2Implementation(){
        vtable = &vt;
        vtable->Put = &Put_;
        vtable->Get = &Get_;
        vtable->Delete = &Delete_;
        vtable->Destroy = &Destroy_;
    }


    static void CALLING_CONVENTION Destroy_(IKVStore2* ikv ){
        delete static_cast<KVStore2Implementation*>(ikv);
    }
```

Implementation

```
 static error_code CALLING_CONVENTION Put_ (IKVStore2* ikv, const char* key, int32_t key_count,const char* value, int32_t value_count){
        try{
            std::string key(key,key_count);
            std::string value(value,value_count);
            static_cast<KVStore2Implementation*>(ikv)->m_[key] = value;
            return 0;

        }
        catch(std::exception&){
            return -1;
        }
    }
```

Implementation

```
 static error_code CALLING_CONVENTION Get_(IKVStore2* ikv,  const char* key, int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn){
        try{
            std::string key(key,key_count);
            auto iter = static_cast<KVStore2Implementation*>(ikv)->m_.find(key);
            if(iter == static_cast<KVStore2Implementation*>(ikv)->m_.end()){
                *breturn = 0;
                return 0;
            }
            else{
                std::string value = iter->second;
                auto pc = new char[value.size()];
                std::copy(value.begin(),value.end(),pc);
                *pvalue_count = value.size();
                *pvalue = pc;
                *breturn = 1;
                return 0;
            }

        }
        catch(std::exception&){
            return -1;
        }

    }
```

Implementation

```
  static error_code CALLING_CONVENTION Delete_(IKVStore2* ikv,  const char* key, int32_t key_count,char* breturn){
         try{
             std::string key(key,key_count);
             auto iter = static_cast<KVStore2Implementation*>(ikv)->m_.find(key);
             if(iter == static_cast<KVStore2Implementation*>(ikv)->m_.end()){
                 *breturn = 0;
                 return 0;
             }
             else{
                 static_cast<KVStore2Implementation*>(ikv)->m_.erase(iter);
                 *breturn = 1;
                 return 0;
             }

         }
         catch(std::exception&){s
             return -1;
         }
     }


};
```

Getting the interface

```
extern "C"{
    IKVStore2* CALLING_CONVENTION Create_KVStore2Implementation(){
        try{
            return new KVStore2Implementation;
        }
        catch(std::exception&){
            return nullptr;
        }

    }

}
```

Usage

```
auto ikv = Create_KVStore2Implementation();
std::string key = "key";
std::string value = "value";
ikv->vtable->Put(ikv,key.data(),key.size(),value.data(),value.size());

const char* pvalue = nullptr;
int32_t count = 0;
char b = 0;

ikv->vtable->Get(ikv,key.data(),key.size(),&pvalue,&count,&b);

std::cout << "Value is " << std::string(pvalue,count) << "\n";

ikv->vtable->Delete(ikv,key.data(),key.size(),&b);

ikv->vtable->Destroy(ikv);
```

Vtable approaches and COM

- The vtable approach whether compiler generated or programmer generated is essentially the binary interface of COM
- If you search the web for solutions to the problem of cross-compiler interfaces, you end up with a lot of articles that either recommend COM explicitly or end up "reinventing" COM (sometimes you even see comments saying "you are reinventing COM")
- While COM works, is not *easy* from C++
- It is helpful to take a look how COM signatures look like
  - `HRESULT __stdcall FunctionName(ParameterType1 p1, ParameterType2 p2, ReturnType* pResult)`
  - No exceptions, return type is not "logical" return type, low-level types for parameters.
- By the way, anybody see the memory leak in the previous code?

What can we do to make it easier

- Hand write wrappers
  - People often write wrappers for a COM interface to make it easier to **use**
  - Not as many people write wrappers to make an interface easier to **implement**
  - Writing 2 wrappers for every interface would probably get old fast
- Macros
  - Limited, hard to use, fragile
- Compiler extensions
  - Visual C++ has had #import for a while which will take a COM type library and write a wrapper to make it easier to use. It will generate RAII wrapper types/typedefs and have logical return values and use exceptions for errors
- Custom code generators
  - Comet tlb2h (http://lambdasoft.dk/comet/) (appears to be from 2004)
- Language extensions

Jim Springfield on Why C++/CX

We actually did develop a new C++ template library for Windows 8 called WRL (Windows Runtime Library) that does support targeting Windows 8 without language extensions. WRL is quite good and it can be illuminating to take a look at it and see how all of the low-level details are implemented. It is used internally by many Windows teams, although it does suffer from many of same problems that ATL does in its support of classic COM.

1. Authoring of components is still very difficult. You have to know a lot of the low-level rules about interfaces.
2. You need a separate tool (MIDL) to author interfaces/types.
3. There is no way to automatically map interfaces from low-level to a higher level (modern) form that throws exceptions and has real return values.
4. There is no unification of authoring and consumption patterns.

http://blogs.msdn.com/b/vcblog/archive/2011/10/20/10228473.aspx

Martial arts movies and C++11

- Martial arts movies often have this plot outline
  - Hero meets villain and gets beaten up
  - Hero meets master and learns
  - Hero meets villain again and beats up villain
- C++11 enables us to make things easier which have been hard for C++ in the past

Jackie Chan from *Drunken Master*
http://snakeandeagle.wordpress.com/movies/drunken-master/

Goals

- No external tools
- Header only
- Define an interface once and use it for both implementation and usage
- Make interfaces easy to implement and use once defined
- Support std::string, vector, and pair in the interface and allow the user to add support for custom types
- Use real return types
- Use exceptions in both usage and implementation
- Support interface inheritance
- Support implementation inheritance
- Binary compatible with COM
- Support multiple platforms (ie not just tied to 1 platform)

Non-goals

- Make easier to use from different languages
  - Part of what makes COM complicated is it has a goal of cross-language compatibility
  - Our focus is on C++11 to C++11
- No compromise machine efficiency
  - Cross-compiler code will not be as fast as say a template library where the compiler is able to see everything and optimize accordingly
  - Willing to trade some efficiency if can get significant usability benefit
  - Try to be "as efficient as possible" and maintain usability benefit

Preview – our KVStore example

```
using cross_compiler_interface::cross_function;

template<class T>
struct InterfaceKVStore
    :public cross_compiler_interface::define_interface<T>
{
    cross_function<InterfaceKVStore,0,void(std::string,std::string)> Put;

    cross_function<InterfaceKVStore,1,
        bool(std::string,cross_compiler_interface::out<std::string>)> Get;

    cross_function<InterfaceKVStore,2, bool(std::string)> Delete;

    cross_function<InterfaceKVStore,3,void()> Destroy;

    InterfaceKVStore():Put(this),Get(this),Delete(this),Destroy(this){}
};
```

Implementation

```
struct ImplementKVStore{
    cross_compiler_interface::implement_interface<InterfaceKVStore> imp_;

    std::map<std::string,std::string> m_;

    ImplementKVStore(){

        imp_.Put = [this](std::string key, std::string value){
            m_[key] = value;
        };

        imp_.Get = [this](std::string key, cross_compiler_interface::out<std::string> value)
               ->bool{
            auto iter = m_.find(key);
            if(iter==m_.end()) return false;
            value.set(iter->second);
            return true;
        };
```

Implementation continued

```
 imp_.Delete = [this](std::string key)->bool{
            auto iter = m_.find(key);
            if(iter==m_.end())return false;
            m_.erase(iter);
            return true;
        };

        imp_.Destroy = [this](){
            delete this;
        };
    }
};
```

Getting the interface

```
extern "C"{
    cross_compiler_interface::portable_base* CALLING_CONVENTION Create_ImplementKVStore(){
        try{
            auto p = new ImplementKVStore;
            return p->imp_.get_portable_base();
        }
        catch(std::exception&){
            return nullptr;
        }

    }

}
```

Usage

```
auto ikv = cross_compiler_interface::
  create<InterfaceKVStore>(m,"Create_ImplementKVStore");

std::string key = "key";
std::string value = "value";
ikv.Put(key,value);

std::string value2;
ikv.Get(key,&value2);

std::cout << "Value is " << value2 << "\n";

ikv.Delete(key);

ikv.Destroy();
```

Key steps

1. Use function objects instead of member functions in the interface
2. Use array of function pointers instead of named function pointers
3. Hold an array of `void*` so the function object can store information for the vtable function
4. Make the function template take an additional `int` parameter so it can use that to find the appropriate vtable function
5. Use a template class to define the interface. Make the function template take another parameter and partial specialize on that to determine if it is for usage or implementation
6. Use a template cross_function that converts from non-trivial types to trivial types and back again. Use cross_function to define the vtable function

Key steps

1. **Use function objects instead of member functions in the interface**
2. Use array of function pointers instead of named function pointers
3. Hold an array of void pointers so the function object can store information for the vtable function
4. Make the function template take an additional int parameter so it can use that to find the appropriate vtable function
5. Use a template class to define the interface. Make the function template take another parameter and partial specialize on that to determine if it is for usage or implementation
6. Use a template cross_function that converts from non-trivial types to trivial types and back again. Use cross_function to define the vtable function

Function objects

- Std::function pretty much enables you to call any callable entity with the same syntax
- Has built in polymorphism
- Handles all the hard work of getting you from call to implementation
- Imagine an interface like this

```
struct FunctionInterface{
    std::function<std::string()> SayHello;
    std::function<std::string(int)> SayMultipleHellos;
};
```

Imagining cross_function

- Takes a signature like std::function
- When used for implementation provides a static function for the vtable that is “low-level” – ie returns error codes, takes “real” return value by pointer and assigns to it. The vtable function will access an std::function with the same signature as the cross_function and convert to “high-level” parameters and catch exceptions thrown by the function and convert them to error codes.
- When used for usage provides an operator() that takes high-level parameters and converts to them to low-level and calls the vtable function. It turns the vtable function returned error code into an exception and returns the “real” return code.

What do we mean by high level or non-trivial and low-level or trivial parameters

- By low-level we mean types that are trivially copyable and standard layout
- By high-level we mean everything else

Trivially copyable class(9 #6)

- No non-trivial copy constructor
- No non-trivial move constructor
- No non-trivial copy assignment operators
- No non-trivial move assignment operators
- Has a trivial destructor

Trivial copy/move constructor (12.8 #13)

- A copy/move constructor for class X is trivial if it is neither user-provided nor deleted and if
- Class X has no virtual functions and no virtual base classes and
- The constructor selected to copy/move each direct base class subobject is trivial and
- For each non-static data member of X that is of class type (or array thereof), the constructor selected to copy/move that member is trivial

Trivial copy/move assignment operator (12.8 #26)

- A copy/move assignment operator for class X is trivial if it is neither user-provided nor deleted and if
- Class X has no virtual functions and no virtual base classes and
- The assignment operator selected to copy/move each direct base subobject is trivial, and
- For each non-static data member of X that is class type(or array thereof), the assignment operator selected to copy/move that member is trivial

Trivial destructor (12.4 #4)

- A destructor is trivial if it is neither user-provided nor deleted and if
- The destructor is not virtual
- All of the direct base classes of its class have trivial destructors, and
- For all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor

Standard layout class(9 #7)

- Has no non-static data members of type non-standard layout class (or array of such types) or reference,
- Has no virtual functions and no virtual base classes
- Has the same access control for all non-static data members
- Has no non-standard layout base classes
- Either has no non-static data members in the most derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and
- Has no base classes of the same type as the first non-static data members
- Note: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.2

Creating a simple cross_function – review of programmer generated vtable

```
struct IKVStoreVtable{
    error_code (CALLING_CONVENTION * Put) (IKVStore2* ikv, const char* key,
      int32_t key_count,const char* value, int32_t value_count);

    error_code (CALLING_CONVENTION *Get)(IKVStore2* ikv, const char* key,
      int32_t key_count,const char** pvalue,int32_t* pvalue_count,char* breturn);

    error_code (CALLING_CONVENTION *Delete)(IKVStore2* ikv, const char* key,
      int32_t key_count,char* breturn);

    void (CALLING_CONVENTION *Destroy)(IKVStore2* ikv);
};
```

Simple cross_function hard wired for Put

```
struct simple_cross_function1_usage{
    IKVStore2* ikv;

    void operator()(std::string key, std::string value){
        auto ret = ikv->vtable->Put(
            ikv,key.data(),key.size(),value.data(),value.size());
        if(ret){
            throw std::runtime_error("Error in Put");
        }
    };

    simple_cross_function1_usage(IKVStore2* i):ikv(i){}
};

struct IKVStore2UsageWrapper{
    simple_cross_function1_usage Put;

    IKVStore2UsageWrapper(IKVStore2* ikv):Put(ikv){}
};
```

Simple cross_function

```cpp
struct IKVStore2Derived:public IKVStore2{
    void* pput;

};

struct simple_cross_function1_implementation{
    std::function<void(std::string,std::string)> put;

    static error_code CALLING_CONVENTION Put_ (IKVStore2* ikv, const char* key,
        int32_t key_count,const char* value, int32_t value_count){
        try{
            std::string key(key,key_count);
            std::string value(value,value_count);
            auto ikvd = static_cast<IKVStore2Derived*>(ikv);
            auto& f = *static_cast<std::function<void(std::string,
                std::string)>*>(ikvd->pput);
            f(key,value);
            return 0;
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Simple cross_function

```
 template<class F>
    void operator=(F f){
        put = f;
    }

    simple_cross_function1_implementation(IKVStore2Derived* ikvd){
        ikvd->pput = &put;
        ikvd->vtable->Put = &Put_;
    }
};

struct IKV2DerivedImplementationBase:public IKVStore2Derived{
    IKVStoreVtable vt;
    IKV2DerivedImplementationBase(){
        vtable = &vt;
    }
};
```

Define wrappers using simple cross_function

```
struct IKVStore2UsageWrapper{
    simple_cross_function1_usage Put;

    IKVStore2UsageWrapper(IKVStore2* ikv):Put(ikv){}
};

struct IKVStore2DerivedImplementation:public IKV2DerivedImplementationBase{
    simple_cross_function1_implementation Put;

    IKVStore2DerivedImplementation():Put(this){}
};
```

Implementing the interface

```cpp
struct KVStore2Implementation2{
    std::map<std::string,std::string> m_;

    IKVStore2DerivedImplementation imp_;

    KVStore2Implementation2(){
        imp_.Put = [this](std::string key, std::string value){
            m_[key] = value;
        };
    }
};
```

Getting the interface

```
extern "C"{
    IKVStore2* CALLING_CONVENTION Create_KVStore2Implementation2(){
        try{
           auto p =  new KVStore2Implementation2;
           return &p->imp_;
        }
        catch(std::exception&){
            return nullptr;
        }

    }

}
```

Using the interface

```
IKVStore2UsageWrapper ikv(Create_KVStore2Implementation2());
ikv.Put("key","value");
```

Critique of simple cross_function

- Makes implementation and usage easier
- Need to make it more general

Key steps

1. Use function objects instead of member functions in the interface
2. **Use array of function pointers instead of named function pointers**
3. **Hold an array of void pointers so the function object can store information for the vtable function**
4. **Make the function template take an additional int parameter so it can use that to find the appropriate vtable function**
5. Use a template class to define the interface. Make the function template take another parameter and partial specialize on that to determine if it is for usage or implementation
6. Use a template cross_function that converts from non-trivial types to trivial types and back again. Use cross_function to define the vtable function

Array of function pointers

- Having named function pointers is not flexible
- Use an array of function pointers
- What type do we use for the array
  - As long as it's a function pointer type it does not matter
- 5.2.10 #6
    - A pointer to a function can be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that is not the same as the type is in the definition of the function is undefined. Except that converting a prvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.

What our binary interface looks like (Actual library code)

```
namespace detail{
    // Calling convention defined in platform specific header
    typedef  void(CROSS_CALL_CALLING_CONVENTION *ptr_fun_void_t)();
}

struct  portable_base{
    detail::ptr_fun_void_t* vfptr;
};
```

A size independent base class for vtable

```
// base class for vtable_n
    struct vtable_n_base:public portable_base{
        void** pdata;
        portable_base* runtime_parent_;
        vtable_n_base(void** p):pdata(p),runtime_parent_(0){}
        template<int n,class T>
        T* get_data()const{
            return static_cast<T*>(pdata[n]);
        }

        void set_data(int n,void* d){
            pdata[n] = d;
        }

```

Continued

```
        template<class R, class... Parms>
        void update(int n,R(CROSS_CALL_CALLING_CONVENTION *pfun)(Parms...)){
            vfptr[n] = reinterpret_cast<detail::ptr_fun_void_t>(pfun);
        }

        template<class R, class... Parms>
        void add(int n,R(CROSS_CALL_CALLING_CONVENTION *pfun)(Parms...)){
            // If you have an assertion here, you have a duplicated number in you interface
            assert(vfptr[n] == nullptr);
            update(n,pfun);
        }
    };
```

The vtable

```cpp
// Our "vtable" definition
    template<int N>
    struct vtable_n:public vtable_n_base
    {
    protected:
        detail::ptr_fun_void_t table_n[N];
        void* data[N];
        enum {sz = N};
        vtable_n():vtable_n_base(data),table_n(),data(){
            vfptr = &table_n[0];
        }

    public:
        portable_base* get_portable_base(){return this;}
        const portable_base* get_portable_base()const{return this;}

    };
```

Simple cross_function_usage

```
template<int n>
struct simple_cross_function2_usage{

    typedef error_code (CALLING_CONVENTION
      *fun_ptr_t)(cross_compiler_interface::portable_base*, const char*,int32_t,const char*,
      int32_t);
    cross_compiler_interface::portable_base* pb_;
    void operator()(std::string key, std::string value){
        auto ret = reinterpret_cast<fun_ptr_t>(pb_->vfptr[n])
               (pb_,key.data(),key.size(),value.data(),value.size());
        if(ret){
            throw std::runtime_error("Error in simple cross_function2");
        }
    }

    simple_cross_function2_usage(cross_compiler_interface::portable_base* p):pb_(p){}
};
```

Simple cross_function_implementation

```
template<int n>
struct simple_cross_function2_implementation{
    std::function<void(std::string,std::string)> f_;
    static error_code CALLING_CONVENTION Function_(
      cross_compiler_interface::portable_base* pb, const char* key,
      int32_t key_count,const char* value, int32_t value_count){
        try{
            std::string key(key,key_count);
            std::string value(value,value_count);
            auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
            auto& f = *static_cast<std::function<void(std::string,
                std::string)>*>(vnb->pdata[n]);
            f(key,value);
            return 0;
        }
        catch(std::exception&){
            return -1;
        }
    }
```

Continued

```
template<class F>
    void operator=(F f){
        f_ = f;
    }

    simple_cross_function2_implementation(cross_compiler_interface::portable_base* pb){
        auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
        vnb->vfptr[n] =
                reinterpret_cast<cross_compiler_interface::detail::ptr_fun_void_t>(&Function_);
        vnb->pdata[n] = &f_;
    }
};
```

Interface based on simple cross_function

```
struct IKVStore2UsageWrapper2{
    simple_cross_function2_usage<0> Put;

    IKVStore2UsageWrapper2(cross_compiler_interface::portable_base* p):Put(p){}
};
struct IKVStore2DerivedImplementation2
   :public cross_compiler_interface::vtable_n<4>{

    simple_cross_function2_implementation<0> Put;

    IKVStore2DerivedImplementation2():Put(this){}
};
```

Critique

- More general, we do not rely on the name, but a position
- However, defining the interface twice(once for usage and once for implementation) is not ideal

Key steps

1. Use function objects instead of member functions in the interface
2. Use array of function pointers instead of named function pointers
3. Hold an array of void pointers so the function object can store information for the vtable function
4. Make the function template take an additional int parameter so it can use that to find the appropriate vtable function
5. **Use a template class to define the interface. Make the function template take another parameter and partial specialize on that to determine if it is for usage or implementation**
6. Use a template cross_function that converts from non-trivial types to trivial types and back again. Use cross_function to define the vtable function

Define use and implement interface

```
template<template <class> class Iface>
struct use_interface:public Iface<use_interface<Iface>>{ // Usage
    explicit use_interface(cross_compiler_interface::portable_base* p):Iface<use_interface<Iface>>(p){}
};

template<template <class> class Iface>
struct implement_interface:
    private cross_compiler_interface::vtable_n<4>,
    public Iface<implement_interface<Iface>>
{
    implement_interface():Iface<implement_interface<Iface>>( this->get_portable_base()){}

    using cross_compiler_interface::vtable_n<4>::get_portable_base;
};
```

Define simple cross_function for usage

```
template<class T, int n>
struct simple_cross_function3{ // usage

    typedef error_code (CALLING_CONVENTION *fun_ptr_t)
        (cross_compiler_interface::portable_base*, const char*, int32_t,const char*, int32_t);
    cross_compiler_interface::portable_base* pb_;
    void operator()(std::string key, std::string value){
        auto ret = reinterpret_cast<fun_ptr_t>(pb_->vfptr[n])
            (pb_,key.data(),key.size(),value.data(),value.size());
        if(ret){
            throw std::runtime_error("Error in simple cross_function3");
        }
    }

    simple_cross_function3(cross_compiler_interface::portable_base* p):pb_(p){}
};
```

Specialize simple cross_function for implementation

```
template<template<class> class Iface,int n>
struct simple_cross_function3<Iface<implement_interface<Iface>>,n>{ // implementation

    std::function<void(std::string,std::string)> f_;

    static error_code CALLING_CONVENTION Function_(cross_compiler_interface::portable_base* pb,
      const char* key,int32_t key_count,const char* value, int32_t value_count){
            try{
                std::string key(key,key_count);
                std::string value(value,value_count);
                auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
                auto& f = *static_cast<std::function<void(std::string,
                    std::string)>*>(vnb->pdata[n]);
                f(key,value);
                return 0;
            }
            catch(std::exception&){
                return -1;
            }
    }
```

Continued

```
 template<class F>
    void operator=(F f){
        f_ = f;
    }

    simple_cross_function3(cross_compiler_interface::portable_base* pb){
        auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
        vnb->vfptr[n] =
            reinterpret_cast<cross_compiler_interface::detail::ptr_fun_void_t>(&Function_);
        vnb->pdata[n] = &f_;
    }
};
```

Defining the interface

```
template<class T>
struct IKV_simple_cross_function3{
    simple_cross_function3<IKV_simple_cross_function3,0> Put;

    IKV_simple_cross_function3(cross_compiler_interface::portable_base* p):Put(p){}

};
```

Implementing the interface

```
struct IKV_simple_cross_function3_implementation{
    std::map<std::string,std::string> m_;

    implement_interface<IKV_simple_cross_function3> imp_;

    IKV_simple_cross_function3_implementation(){
        imp_.Put = [this](std::string key, std::string value){
            m_[key] = value;
        };
    }
};
```

Using the interface

1.
```
use_interface<IKV_simple_cross_function3>
    ikv(Create_IKV_simple_cross_function3_implementation());
```
2.
```
ikv.Put("key","value");
```

Key steps

1. Use function objects instead of member functions in the interface
2. Use array of function pointers instead of named function pointers
3. Hold an array of void pointers so the function object can store information for the vtable function
4. Make the function template take an additional int parameter so it can use that to find the appropriate vtable function
5. Use a template class to define the interface. Make the function template take another parameter and partial specialize on that to determine if it is for usage or implementation
6. **Use a template cross_function that converts from non-standard layout/trivial copy types to standard layout/trivial copy types and back again. Use cross_function to define the vtable function**

Cross_conversion

- Converts to and from a trivial type (standard layout/trivially copyable)
- May be specialized
- If a type is already standard layout/trivially copyable, a class `trivial_conversion` is provided
- Trivial conversions provided for char, (u)int8/16/32/16, float, double, void*
- Specialization provided for bool, std::string, std::vector, std::pair
- No specialization provided for long double

Trivial conversion

```
template<class T>
   struct trivial_conversion{
       typedef T converted_type;
       typedef T original_type;
       static converted_type to_converted_type(original_type i){return i;};
       static original_type to_original_type(converted_type c){return c;}
   };

// Allow support for void* and const void*
   template<>
   struct cross_conversion<void*>:public trivial_conversion<void*>{};
   template<>
   struct cross_conversion<const void*>:public trivial_conversion<const void*>{};
```

A trivial type to represent a string

```
struct cross_string{
        const char* begin;
        const char* end;
    }CROSS_COMPILER_INTERFACE_PACK;
```

The cross_conversion specialization

```
template<>
    struct cross_conversion<std::string>{
        typedef std::string original_type;
        typedef cross_string converted_type;
        static converted_type to_converted_type(const original_type& s){
            cross_string ret;
            ret.begin = s.data();
            ret.end = s.data() + s.size();
            return ret;
        }
        static  std::string to_original_type(converted_type& c){
            return std::string(c.begin,c.end);
        }
    };
```

Using cross_conversion for simple_cross_function (usage)

```
template<class T, int n, class F> struct simple_cross_function4{};

template<class T, int n, class Parm1, class Parm2>
struct simple_cross_function4<T,n,void(Parm1,Parm2)>{ // usage

    typedef error_code (CALLING_CONVENTION
        *fun_ptr_t)(cross_compiler_interface::portable_base*,
        typename cross_compiler_interface::cross_conversion<Parm1>::converted_type,
        typename cross_compiler_interface::cross_conversion<Parm2>::converted_type);

    cross_compiler_interface::portable_base* pb_;

    void operator()(Parm1 p1, Parm2 p2){
        auto ret = reinterpret_cast<fun_ptr_t>(pb_->vfptr[n])(pb_,
            cross_compiler_interface::cross_conversion<Parm1>::to_converted_type(p1),
            cross_compiler_interface::cross_conversion<Parm2>::to_converted_type(p2));
        if(ret){
            throw std::runtime_error("Error in simple cross_function2");
        }
    }

    simple_cross_function4(cross_compiler_interface::portable_base* p):pb_(p){}
};
```

Simple_cross_function (implementation)

```
template<template<class> class Iface,int n,class Parm1, class Parm2>
struct simple_cross_function4<Iface<implement_interface<Iface>>,n,void(Parm1,Parm2)>{
// implementation

    std::function<void(Parm1, Parm2)> f_;
    // Without these msvc has compiler error
    typedef cross_compiler_interface::cross_conversion<Parm1> cc1;
    typedef cross_compiler_interface::cross_conversion<Parm2> cc2;
```

Continued

```
static error_code CALLING_CONVENTION Function_ (cross_compiler_interface::portable_base* pb,
        typename cross_compiler_interface::cross_conversion<Parm1>::converted_type p1,
        typename cross_compiler_interface::cross_conversion<Parm2>::converted_type p2){
            try{
                using namespace std;
                using namespace cross_compiler_interface;
                auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
                auto& f = *static_cast<std::function<void(Parm1, Parm2)>*>(vnb->pdata[n]);
                f(cc1::to_original_type(p1),cc2::to_original_type(p2));
                return 0;
            }
            catch(std::exception&){
                return -1;
            }
    }
```

Continued

```
template<class F>
    void operator=(F f){
        f_ = f;
    }

    simple_cross_function4(cross_compiler_interface::portable_base* pb){
        auto vnb = static_cast<cross_compiler_interface::vtable_n_base*>(pb);
        vnb->vfptr[n] =
            reinterpret_cast<cross_compiler_interface::detail::ptr_fun_void_t>(&Function_);
        vnb->pdata[n] = &f_;
    }
};
```

Simple cross_function review

- We can now handle any function that takes 2 parameters and has a void return
- We can define an interface once and use it via use_interface and implement_interface for both client usage and implementation
- To generalize, we use variadic templates
- With this background, we will review how to do various things with the library

Defining an interface

```
template<class T>
struct InterfaceKVStore
    :public cross_compiler_interface::define_interface<T>
{
    cross_function<InterfaceKVStore,0,void(std::string,std::string)> Put;

    cross_function<InterfaceKVStore,1,
        bool(std::string,cross_compiler_interface::out<std::string>)> Get;

    cross_function<InterfaceKVStore,2,
        bool(std::string)> Delete;
    cross_function<InterfaceKVStore,3,void()> Destroy;

    InterfaceKVStore()
        :Put(this),Get(this),Delete(this),Destroy(this)
    {}
};
```

Shorter way (does not work with MSVC currently)

```
template<class T>
struct InterfaceKVStore
    :public cross_compiler_interface::define_interface<T>
{
    template<int Id, class F>
    using cf = cross_function<InterfaceKVStore,Id,F>;

    cf<0, void(std::string,std::string)> Put = this;

    cf<1, bool(std::string, cross_compiler_interface::out<std::string>)> Get = this;

    cf<2, bool(std::string)> Delete = this;

    cf<3, void()> Destroy = this;

    InterfaceKVStore(){}
};
```

Calculating vtable size and catching misnumbering errors

```
template<class T>
struct InterfaceKVStore
:public cross_compiler_interface::define_interface<T>
{
cross_function<InterfaceKVStore,0,void(std::string,std::string)>
Put;

cross_function<InterfaceKVStore,1,
bool(std::string,cross_compiler_interface::out<std::string>)> Get;

cross_function<InterfaceKVStore,2,
bool(std::string)> Delete;
cross_function<InterfaceKVStore,2,void()> Destroy;

InterfaceKVStore()
:Put(this),Get(this),Delete(this),Destroy(this)
{}
};
```

Calculating vtable size and catching misnumbering errors

- 4>ClCompile:
- 4> simple_demo_dll.cpp
- 3>c:\users\jrb\source\repos\cross_compiler_call\cross_compiler_interface\cross_compiler_interface.hpp(606): error C2338: The Id's for a cross_function need to be ascending order from 0, you have possibly repeated a number
- 3>   c:\users\jrb\source\repos\cross_compiler_call\simple_demo.cpp(126) : see reference to class template instantiation 'cross_compiler_interface::use_interface<InterfaceKVStore>' being compiled

Calculating vtable size and catching misnumbering errors

```
struct size_only{};
struct checksum_only{};

   // size only
   template<template<class> class Iface,int Id,class F>
   struct cross_function<Iface<size_only>,Id,F>{
        char a[1024];
       template<class T>
       cross_function(T t){}
   };
   // checksum only
   template<template<class> class Iface,int Id,class F>
   struct cross_function<Iface<checksum_only>,Id,F>{
       char a[1024*(Id+1+Iface<checksum_only>::base_sz)
              *(Id+1+Iface<checksum_only>::base_sz)];
       template<class T>
       cross_function(T t){}
   };
```

Calculating the vtable size and catching numbering errors continued

```
enum{num_functions = sizeof(Iface<size_only>)/sizeof(cross_function<Iface<size_only>,0,void()>)};
private:

// Simple checksum that takes advantage that sum of squares can be calculated with formula (n(n+1)(2n+1)/6
enum{checksum = sizeof(Iface<checksum_only>)/sizeof(cross_function<InterfaceBase<checksum_only>,0,void()>)};

// Simple check to catch simple errors where the Id is misnumbered uses sum of squares
static_assert(checksum==(num_functions * (num_functions +1)*(2*num_functions + 1 ))/6,"The Id's for a cross_function need to be ascending order from 0, you have possibly repeated a number");
```

Types supported as parameters and returns – Items in blue are trivial

- char
- (u)int8/16/32/64_t
- float, double
- all (const) * and (const) & of the above
- (const) void*
- bool
- std::string,vector,pair
- cr_string (an adaptation of ref_string from boost 1.53, to allow us to pass references to strings without copying)
- use_interface<Interface>, use_unknown<Interface>
- out<T> (allows for out parameters, you pass in a parameter by taking the address, and in the implementation you call outvar.set(value) to set the value.

Inheriting an interface

```cpp
struct InterfaceKVStore
    :public cross_compiler_interface::define_interface<T>
```

```cpp
struct InterfaceKVStore
    :public cross_compiler_interface::define_interface<T,BaseInterface>
```

- You do not have to change anything else in the interface
- The integer provided to the `cross_function` template does not need to change. `cross_function` will calculate the correct vtable offset by adding the template parameter to the number of functions in the base class

Using an interface

```cpp
compiler_interface::module m("simple_demo_dll");
auto ikv =
cross_compiler_interface::create<InterfaceKVStore>(m,"Create_ImplementKVStore");

std::string key = "key";
std::string value = "value";
ikv.Put(key,value);

std::string value2;
ikv.Get(key,&value2);

std::cout << "Value is " << value2 << "\n";

ikv.Delete(key);

ikv.Destroy();
```

Implementing an interface

```
struct ImplementKVStore{
    cross_compiler_interface::implement_interface<InterfaceKVStore> imp_;

    std::map<std::string,std::string> m_;

    ImplementKVStore(){

        imp_.Put = [this](std::string key, std::string value){
            m_[key] = value;
        };

        imp_.Get = [this](string key, out<string> value)->bool{
            auto iter = m_.find(key);
            if(iter==m_.end()) return false;
            value.set(iter->second);
            return true;
        };
        // Other functions
    }
};
```

Use a member function instead of a lambda

- `imp_.Put.set_mem_fn<ImplementKVStore,&ImplementKVStore::Put>(this);`
- Faster as it avoids a second indirect function call due to std::function. It also avoids checking to make sure function was assigned a lambda.

Reusing interfaces and implementations

- Can inherit an interface as above (single inheritance only)
- Reuse implementation of interface via source reuse
  - If you have an interface that is used a lot, you can define a class to implement that interface and use containment
- Reuse implementation of interface via binary reuse

Source reuse – example interface

```cpp
template<class T>
struct PropertyInterface:public cross_compiler_interface::define_interface<T>{

    cross_function<PropertyInterface,0,void(std::string /*key*/ ,std::string /*value*/)> SetProperty;

    cross_function<PropertyInterface,1,std::string(std::string /*key*/,std::string /*default*/)> GetProperty;

    PropertyInterface():SetProperty(this),GetProperty(this){}

};
```

Source reuse – reusable implementation

```
struct PropertyInterfaceImplementationHelper{

    std::map<std::string, std::string> m_;

PropertyInterfaceImplementationHelper(cross_compiler_interface::implement_interface<PropertyInterface>& imp){
        imp.SetProperty = [this](std::string key, std::string value){
            m_[key] = value;
        };

        imp.GetProperty = [this](std::string key, std::string default_value){
            auto iter = m_.find(key);
            if(iter==m_.end()) return default_value;
            return iter->second;
        };

    };
};
```

Source reuse – using the implementation

```
struct ImplementPropertyInterface{

    cross_compiler_interface::implement_interface<PropertyInterface> imp_;
    PropertyInterfaceImplementationHelper helper_;
    ImplementPropertyInterface():helper_(imp_){}
};
```

Binary reuse

- Suppose you want to implement an interface in terms of another implementation of that interface
- That implementation could be in another dll, maybe one that was even compiled with another compiler
- To use that interface, you can implement the interface methods and manually forward them to the other implementation
  - That is tedious
- You could use something like COM aggregation
  - Complicated and the component has to support it
- You could use `set_runtime_parent`
  - If an interface method does not have a lamda assigned, any call on that interface will be forwarded to the runtime parent

Using set_rutime_parent

```
struct ImplementPropertyInterfaceBinary{

    cross_compiler_interface::module m_;

    cross_compiler_interface::use_interface<PropertyInterface> other_;

    cross_compiler_interface::implement_interface<PropertyInterface> imp_;

    ImplementPropertyInterfaceBinary()
        :m_("AwesomeDll")
    {
        other_ = cross_compiler_interface::create<PropertyInterface>
               (m_,"CreatePropertyManager");

        imp_.set_runtime_parent(other_);
    }
};
```

How does set_runtime_parent work

```
// base class for vtable_n
    struct vtable_n_base:public portable_base{
        void** pdata;
        portable_base* runtime_parent_;
        vtable_n_base(void** p):pdata(p),runtime_parent_(0){}
        template<int n,class T>
        T* get_data()const{
            return static_cast<T*>(pdata[n]);
        }

        void set_data(int n,void* d){
            pdata[n] = d;
        }

```

Runtime_parent inside the vtable function

```
auto& f = detail::get_function<N,fun_t>(v);
if(!f){
    // See if runtime inheritance present with parent
    const vtable_n_base* vt = static_cast<const vtable_n_base*>(v);
    if(vt->runtime_parent_){
        return reinterpret_cast<vt_entry_func>(vt->runtime_parent_->
            vfptr[N])(vt->runtime_parent_, r,detail::dummy_conversion<
            typename cross_conversion<Parms>::converted_type>(p)...);
     }
     else{
        return error_not_implemented::ec;
      }
 }
```

Lifetime Management and Multiple Interfaces

- So far, we have considered single interfaces with a destroy function
- What if we want to support multiple interfaces and have automated lifetime management
- For multiple interfaces, we need a way to go from one interface to another as we cannot use dynamic_cast
- One way we could do automatic lifetime management would be with reference counting
- We need an interface that can handle lifetime management and interface discovery – any suggestions?

IUnknown and nsISupports

- QueryInterface
- AddRef
- Release

Defining an interface that supports IUnknown

- Use `define_unknown_interface`
- Same parameters as `define_interface`, except the second parameter takes a `uuid`
- The repository includes source code for a simple program based on boost.uuid to generate the uuid and class outline
  - `Create_unknown_interface_with_uuid` InterfaceName [BaseInterface]
    - BaseInterface is optional

Defining an interface that supports IUnknown

```
template<class T>
struct InterfaceKVStore2
:public cross_compiler_interface::define_unknown_interface<T,
// {B781B4FF-995D-4122-842C-E14A4C0348CC}
cross_compiler_interface::uuid<
0xB781B4FF,0x995D,0x4122,0x84,0x2C,0xE1,0x4A,0x4C,0x03,0x48,0xCC
>
>
{
    typedef cross_compiler_interface::cr_string cr_string;

    cross_function<InterfaceKVStore2,0,void(cr_string,cr_string)> Put;
    cross_function<InterfaceKVStore2,1,bool(cr_string,
      cross_compiler_interface::out<std::string>)> Get;
    cross_function<InterfaceKVStore2,2,bool(cr_string)> Delete;

    InterfaceKVStore2()
     :Put(this),Get(this),Delete(this){}
};
```

Implementing an interface that supports IUnknown

```
struct ImplementKVStore2
    :public implement_unknown_interfaces<ImplementKVStore2,InterfaceKVStore2>

{

    std::map<std::string,std::string> m_;

    ImplementKVStore2(){
        using cross_compiler_interface::cr_string;
        auto imp = get_implementation<InterfaceKVStore2>();

        imp->Put = [this](cr_string key, cr_string value){
            m_[key.to_string()] = value.to_string();
        };
```

Using an interface that supports IUnknown

```
use_unknown<InterfaceUnknown> iunk = create_unknown(m,"Create_ImplementKVStore2");

auto ikv = iunk.QueryInterface<InterfaceKVStore2>();

std::string key = "key";
std::string value = "value";
ikv.Put(key,value);

std::string value2;
ikv.Get(key,&value2);

std::cout << "Value is " << value2 << "\n";

ikv.Delete(key);
```

Error handling

- HRESULT
- All the vtable functions return a 32-bit signed integer
- A 0 is success
- A negative value is an error
- Has function to turn exceptions to error_codes and error_codes to exceptions
- Supports so far 15 error codes with own classes, other error codes get turned into a generic exception (cross_compiler_interface_error_base) that has a get_error_code function

Custom cross functions

- Most of the time the automated conversions provided by cross_function will suffice
- Sometimes, however, you may want to define the signature of the vtable function and how the conversions occur
- One time you might do this is where you want to be binary compatible with an already specified interface
- For example, in writing IUnknown support , custom functions were used because we wanted to be binary compatible with IUnknown

Using custom_cross_function

- `template<class Iface, int Id,class F1, class F2,class Derived,class FuncType = std::function<F1>> struct custom_cross_function`
- F1 is the signature visible to users/implementers of the interface
- F2 is the signature of the vtable function (don't forget to include a portable_base* as your first parameter)
- Derived is the name of the class deriving from `custom_cross_function`
- Currently `custom_cross_function` is geared toward vtable functions that return integer error codes
- Custom_cross_function will handle `set_runtime_parent` as well as `set_mem_fn`

Example of custom_cross_function usage

- In implementing IUnknown support, we needed to define an interface that would be binary compatible with IUnknown
- Unfortunately, if we used cross_function the vtable functions would have the wrong signatures
- To get around this, we implemented the IUnknown methods as custom_cross_functions.
- AddRef has a vtable signature like this - `uint32_t (portable_base*)`
- If we used custom cross_function with a signature of `uint32_t ()` the vtable function would have been `error_code f(portable_base*,uint32_t*)`
- We will go step by step to see how we use custom_cross_function to achieve the right signature

Step 1 – Derive from custom_cross_function

```
template<class Iface, int Id>
struct addref_release_cross_function
:public custom_cross_function<Iface, Id,   std::uint32_t(),std::uint32_t(portable_base*),
     addref_release_cross_function<Iface,Id>>
{
```

Step 2 – Write call_vtable_function and vtable_function

```
std::uint32_t call_vtable_function()const{
    return this->get_vtable_fn()(this->get_portable_base());
}

template<class F>
static std::uint32_t vtable_function(F f, portable_base* v){
    try{
         return f();
    } catch(std::exception& ){
        return 0;
    }
}
```

Step 3 – Write constructor and operator=

```
template<class F>
void operator=(F f){
    this->set_function(f);
}

template<class T>
addref_release_cross_function(T t)
    :addref_release_cross_function::base_t(t){}

};
```

Using your custom cross function

```
//IUnknown
typedef uuid<0x00000000,0x0000,0x0000,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46>
      Unknown_uuid_t;
template<class T>
struct InterfaceUnknown:public define_interface<T>{

      query_interface_cross_function<InterfaceUnknown,0> QueryInterfaceRaw;

      addref_release_cross_function<InterfaceUnknown,1> AddRef;

      addref_release_cross_function<InterfaceUnknown,2> Release;


      typedef Unknown_uuid_t uuid;


      InterfaceUnknown()
          :QueryInterfaceRaw(this),AddRef(this),Release(this){}

};
```

Return values

```cpp
template<class T>
struct cross_conversion_return{
    typedef cross_conversion<T> cc;
    typedef typename cc::original_type return_type;
    typedef typename cc::converted_type converted_type;

    static void initialize_return(return_type&, converted_type&){
        // do nothing
    }

    static void do_return(const return_type& r,converted_type& c){
        typedef cross_conversion<T> cc;
        c = cc::to_converted_type(r);
    }
    static void finalize_return(return_type& r,converted_type& c){
        r = cc::to_original_type(c);
    }

};
```

Vtable_caller

```
template<template<class> class Iface, int N>
struct call_adaptor{

    template<class R,class... Parms>
    struct vtable_caller{
        static R call_vtable_func(const detail::ptr_fun_void_t pFun,const portable_base* v,typename arg<Parms>::type... p){
            using namespace std; typedef cross_conversion_return<R> ccr;
            typedef typename ccr::converted_type cret_t;
            typename ccr::return_type r;
            cret_t cret;
            ccr::initialize_return(r,cret);
            auto ret =  detail::call<error_code,const portable_base*, cret_t*, typename cross_conversion<Parms>::converted_type...>(pFun, v,&cret,conversion_helper::to_converted<Parms>(p)...);
           if(ret < 0){
               error_mapper<Iface>::mapper::exception_from_error_code(ret);
           }
           ccr::finalize_return(r,cret);
           return r;
        }
    };
```

Vtable_entry

```
template<class R,class... Parms>
        struct vtable_entry{
            typedef std::function<R(Parms...)> fun_t;
            typedef cross_conversion_return<R> ccr;
            typedef error_code (CROSS_CALL_CALLING_CONVENTION * vt_entry_func)(const portable_base*, typename ccr::converted_type*,typename cross_conversion<Parms>::converted_type...);

            static error_code CROSS_CALL_CALLING_CONVENTION func(const portable_base* v, typename ccr::converted_type* r,typename cross_conversion<Parms>::converted_type... p){
                using namespace std;
```

Vtable_entry continued

```
            try{
                auto& f = detail::get_function<N,fun_t>(v);
                if(!f){
                    // See if runtime inheritance present with parent
                    const vtable_n_base* vt = static_cast<const
                        vtable_n_base*>(v);
                    if(vt->runtime_parent_){
                        return reinterpret_cast<vt_entry_func>(vt->runtime_parent_->
                        vfptr[N])(vt->runtime_parent_, r,detail::dummy_conversion<
                        typename cross_conversion<Parms>::converted_type>(p)...);
                    }
                    else{
                        return error_not_implemented::ec;
                    }
                }
              ccr::do_return(f(conversion_helper::to_original<Parms>(p)... ),*r);
                return 0;
            } catch(std::exception& e){
                return error_mapper<Iface>::mapper::error_code_from_exception(e);
            }
        }
    };
```

Specializing cross_conversion_return

```
template<>
   struct cross_conversion_return<std::string>{
       typedef std::string return_type;
       typedef cross_string_return converted_type;

       static error_code CROSS_CALL_CALLING_CONVENTION do_transfer_string(void* str,const char* begin, const char* end){
           try{
               auto& s = *static_cast<std::string*>(str);
               s.assign(begin,end);
               return 0;
           }
           catch(std::exception& e){
               return general_error_mapper::error_code_from_exception(e);
           }

       }
   };
```

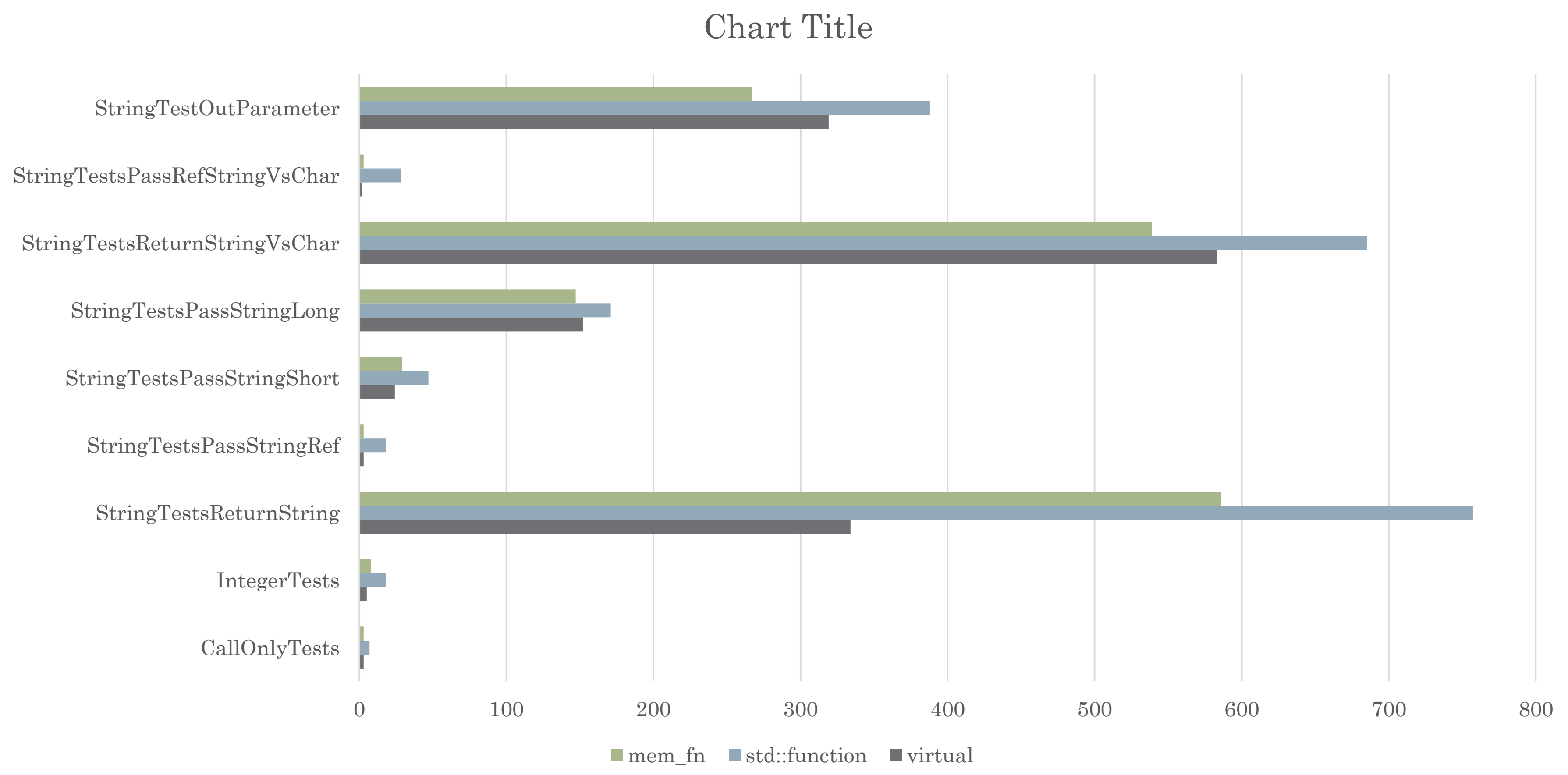
Specializing cross_conversion_return

```
static void initialize_return(return_type& r, converted_type& c){
            c.retstr = &r;
            c.transfer_string = &do_transfer_string;
        }

        static void do_return(const return_type& r,converted_type& c){
            auto ec = c.transfer_string(c.retstr,r.data(),r.data() + r.size());
            if(ec < 0){
                general_error_mapper::exception_from_error_code(ec);
            }
        }
        static void finalize_return(return_type& r,converted_type& c){
            // do nothing
        }


    };
```

Performance

- How much are we paying for this convenience?
- Cross_compiler_interface does the following to try to make performance acceptable
  - Provides set_mem_fn to avoid the extra indirect function call we would get with std::function
  - Does not allocate memory on its own
  - Use function pointers to assign return values to strings/vectors/pairs across boundaries
- The following chart shows the results of running a simple benchmark comparing a regular virtual interface with cross compiler interfaces implemented with lambda's and member functions.
- Test compiled with MSVC 2012 32-bit with full optimizations and run on i5-2300 running Windows 8 64-bit
- 1 million function calls were made and then averaged
- The string return tests and and string tests marked long are string that are 4K in size

Performance



Can we make it easier?

```
template<class T>
struct InterfaceKVStore2
:public cross_compiler_interface::define_unknown_interface<T,
// {B781B4FF-995D-4122-842C-E14A4C0348CC}
cross_compiler_interface::uuid<
0xB781B4FF,0x995D,0x4122,0x84,0x2C,0xE1,0x4A,0x4C,0x03,0x48,0xCC
>
>
{
    typedef cross_compiler_interface::cr_string cr_string;

    cross_function<InterfaceKVStore2,0,void(cr_string,cr_string)> Put;
    cross_function<InterfaceKVStore2,1,bool(cr_string,
      cross_compiler_interface::out<std::string>)> Get;
    cross_function<InterfaceKVStore2,2,bool(cr_string)> Delete;

    InterfaceKVStore2()
     :Put(this),Get(this),Delete(this){}
};
```

Can we make it easier?

- No
- Unless…
- We use Macros

Interface Definition

```
struct KVStoreFinal{

    typedef
        cross_compiler_interface::uuid<
        0x8B651383,0x8852,0x4DF7,0x81,0x1A,0xBF,0xAE,0xD8,0x7D,0x02,0xE9
        > uuid;

    typedef cross_compiler_interface::cr_string cr_string;

    void Put(cr_string key, cr_string value);
    bool Get(cr_string key, cross_compiler_interface::out<std::string> pvalue);
    bool Delete(cr_string key);

    CROSS_COMPILER_INTERFACE_CONSTRUCT_UNKNOWN_INTERFACE(KVStoreFinal,Put,Get,Delete);
};
```

Interface Implementation

```
struct ImplementKVStoreFinal
    :public implement_unknown_interfaces<ImplementKVStoreFinal,KVStoreFinal::Interface>{

            typedef cross_compiler_interface::cr_string cr_string;

            std::map<std::string,std::string> m_;

            void Put(cr_string key, cr_string value){
                m_[key.to_string()] = value.to_string();

            }

            bool Get(cr_string key, cross_compiler_interface::out<std::string> pvalue){
                auto iter = m_.find(key.to_string());
                if(iter==m_.end()) return false;
                pvalue.set(iter->second);
                return true;
            }
```

Interface Implementation

```
            bool Delete(cr_string key){
                auto iter = m_.find(key.to_string());
                if(iter==m_.end())return false;
                m_.erase(iter);
                return true;
            }

            ImplementKVStoreFinal(){
                get_implementation<KVStoreFinal::Interface>()
                    ->map_to_member_functions_no_prefix(this);
            }
};
```

Creating the implementation

```
extern "C"{
    cross_compiler_interface::portable_base* CALLING_CONVENTION Create_ImplementKVStoreFinal(){
        try{
            auto p = ImplementKVStoreFinal::create();
            return p.get_portable_base_addref();
        }
        catch(std::exception&){
            return nullptr;
        }

    }

}
```

Using the interface

```cpp
using namespace cross_compiler_interface;

auto ikv = create_unknown(m,"Create_ImplementKVStoreFinal")
    .QueryInterface<KVStoreFinal::Interface>();

std::string key = "key";
std::string value = "value";
ikv.Put(key,value);

std::string value2;
ikv.Get(key,&value2);

std::cout << "Value is " << value2 << "\n";

ikv.Delete(key);
```

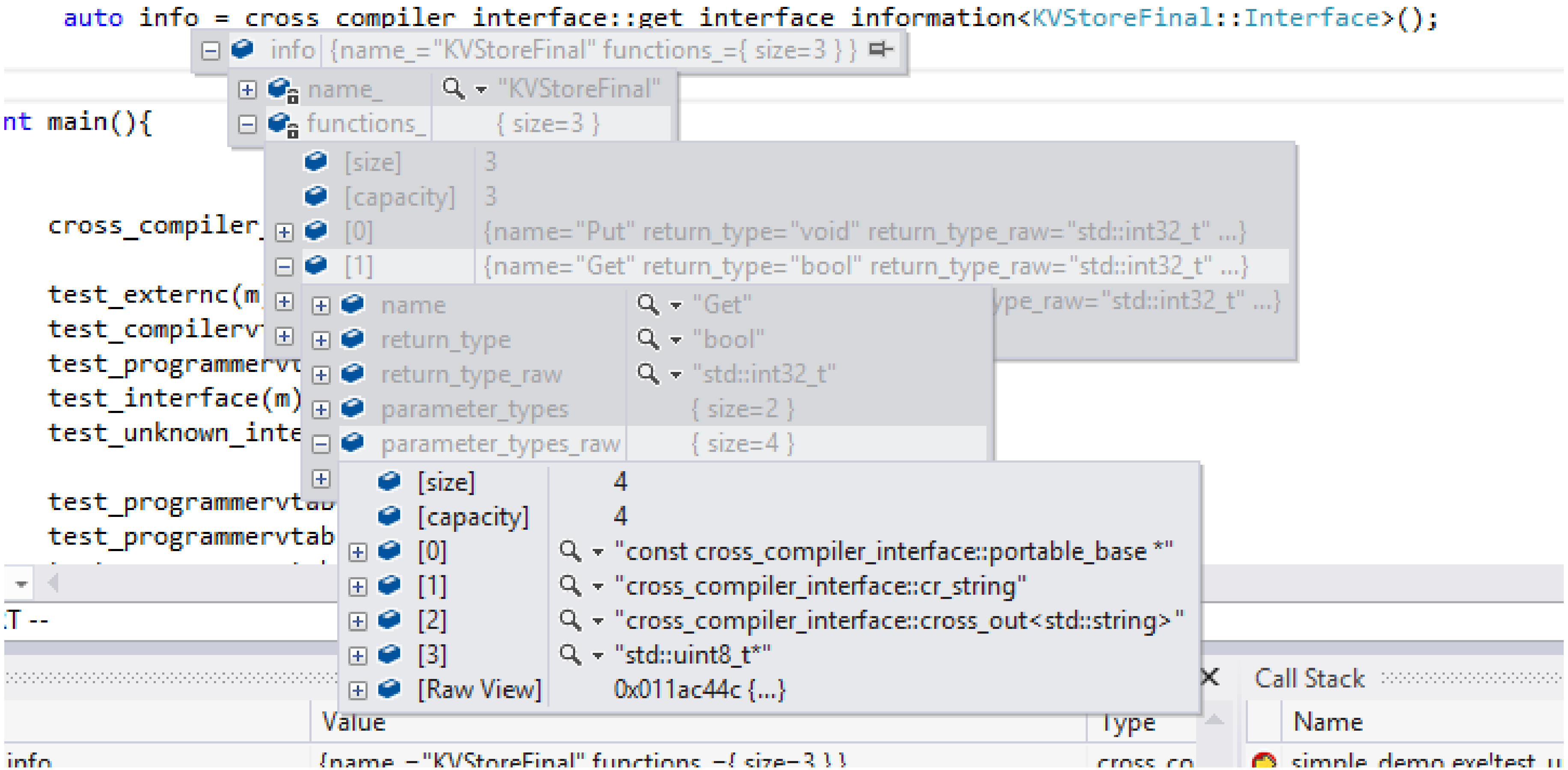
How it works

```
#define CROSS_COMPILER_INTERFACE_HELPER_CONSTRUCT_INTERFACE(T,B,...)   \
    template<class Type> struct Interface:public B{ \
CROSS_COMPILER_INTERFACE_SEMICOLON_APPLY(T,CROSS_COMPILER_INTERFACE_DECLARE_CROSS_FUNCTION_EACH,__VA_ARGS__)\
Interface():CROSS_COMPILER_INTERFACE_APPLY(T,CROSS_COMPILER_INTERFACE_DECLARE_CONSTRUCTOR,__VA_ARGS__){}\
    template<class Derived>\
    void map_to_member_functions_no_prefix(Derived* pthis){CROSS_COMPILER_INTERFACE_SEMICOLON_APPLY(T,CROSS_COMPILER_INTERFACE_DECLARE_MAP_TO_MEMBER_FUNCTIONS_NO_PREFIX_EACH,__VA_ARGS__);}\
    template<class Derived>\
    void map_to_member_functions(Derived* pthis){CROSS_COMPILER_INTERFACE_SEMICOLON_APPLY(T,CROSS_COMPILER_INTERFACE_DECLARE_MAP_TO_MEMBER_FUNCTIONS_EACH,__VA_ARGS__);} \
// Other stuff for introspection
};

#define CROSS_COMPILER_INTERFACE_CONSTRUCT_UNKNOWN_INTERFACE(T,...)  \
    CROSS_COMPILER_INTERFACE_HELPER_CONSTRUCT_INTERFACE(T, cross_compiler_interface::define_unknown_interface<Type CROSS_COMPILER_INTERFACE_COMMA T::uuid>, __VA_ARGS__)
```

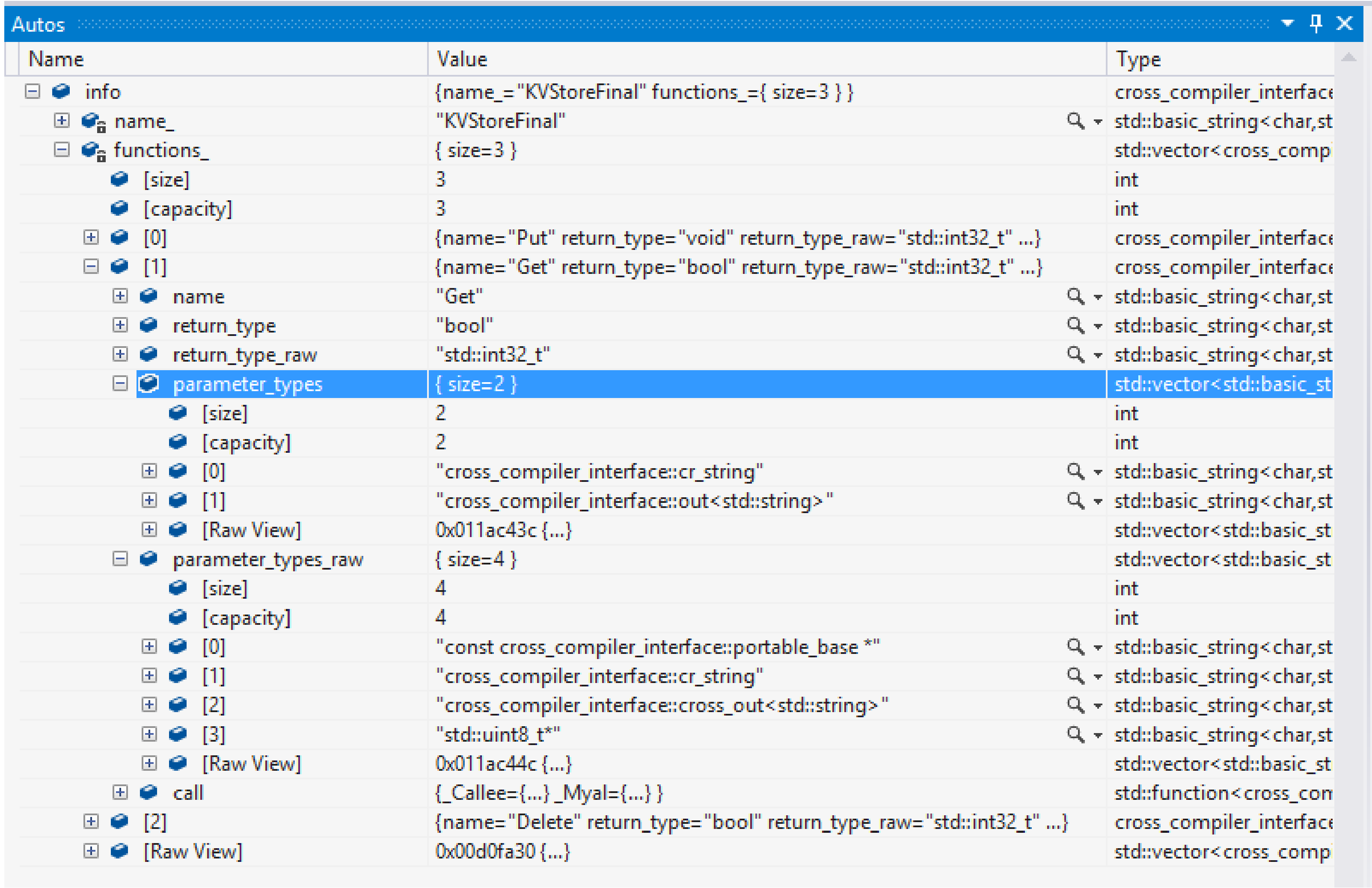
How it works

```
 template<class Interface,int Id,class T, class R, class... P>
    cross_function<Interface,Id,R(P...)> cf_from_member_function(R (T::*)(P...) );

    }

#define CROSS_COMPILER_INTERFACE_DECLARE_CROSS_FUNCTION_EACH(T,i,x) decltype(cross_compiler_interface::detail::cf_from_member_function<Interface,i-1>(&T::x)) x
#define CROSS_COMPILER_INTERFACE_DECLARE_MAP_TO_MEMBER_FUNCTIONS_NO_PREFIX_EACH(T,i,x) x.template set_mem_fn<Derived,&Derived::x>(pthis)
#define CROSS_COMPILER_INTERFACE_DECLARE_MAP_TO_MEMBER_FUNCTIONS_EACH(T,i,x) x.template set_mem_fn<Derived,&Derived::CROSS_COMPILER_INTERFACE_CAT(CROSS_COMPILER_INTERFACE_CAT(T,_),x)>(pthis)
#define CROSS_COMPILER_INTERFACE_DECLARE_CONSTRUCTOR(T,i,x) x(this)
```

Introspection



Introspection



Autos

Name	Value	Type
info	{name_="KVStoreFinal" functions_={ size=3 } }	cross_compiler_interface
name_	"KVStoreFinal"	std::basic_string<char,st
functions_	{ size=3 }	std::vector<cross_comp
[size]	3	int
[capacity]	3	int
[0]	{name="Put" return_type="void" return_type_raw="std::int32_t" ...}	cross_compiler_interface
[1]	{name="Get" return_type="bool" return_type_raw="std::int32_t" ...}	cross_compiler_interface
name	"Get"	std::basic_string<char,st
return_type	"bool"	std::basic_string<char,st
return_type_raw	"std::int32_t"	std::basic_string<char,st
parameter_types	{ size=2 }	std::vector<std::basic_st
[size]	2	int
[capacity]	2	int
[0]	"cross_compiler_interface::cr_string"	std::basic_string<char,st
[1]	"cross_compiler_interface::out<std::string>"	std::basic_string<char,st
[Raw View]	0x011ac43c {...}	std::vector<std::basic_st
parameter_types_raw	{ size=4 }	std::vector<std::basic_st
[size]	4	int
[capacity]	4	int
[0]	"const cross_compiler_interface::portable_base *"	std::basic_string<char,st
[1]	"cross_compiler_interface::cr_string"	std::basic_string<char,st
[2]	"cross_compiler_interface::cross_out<std::string>"	std::basic_string<char,st
[3]	"std::uint8_t*"	std::basic_string<char,st
[Raw View]	0x011ac44c {...}	std::vector<std::basic_st
call	{_Callee={...} _Myal={...} }	std::function<cross_com
[2]	{name="Delete" return_type="bool" return_type_raw="std::int32_t" ...}	cross_compiler_interface
[Raw View]	0x00d0fa30 {...}	std::vector<cross_comp

Current and planned projects based on cross_compiler_interface

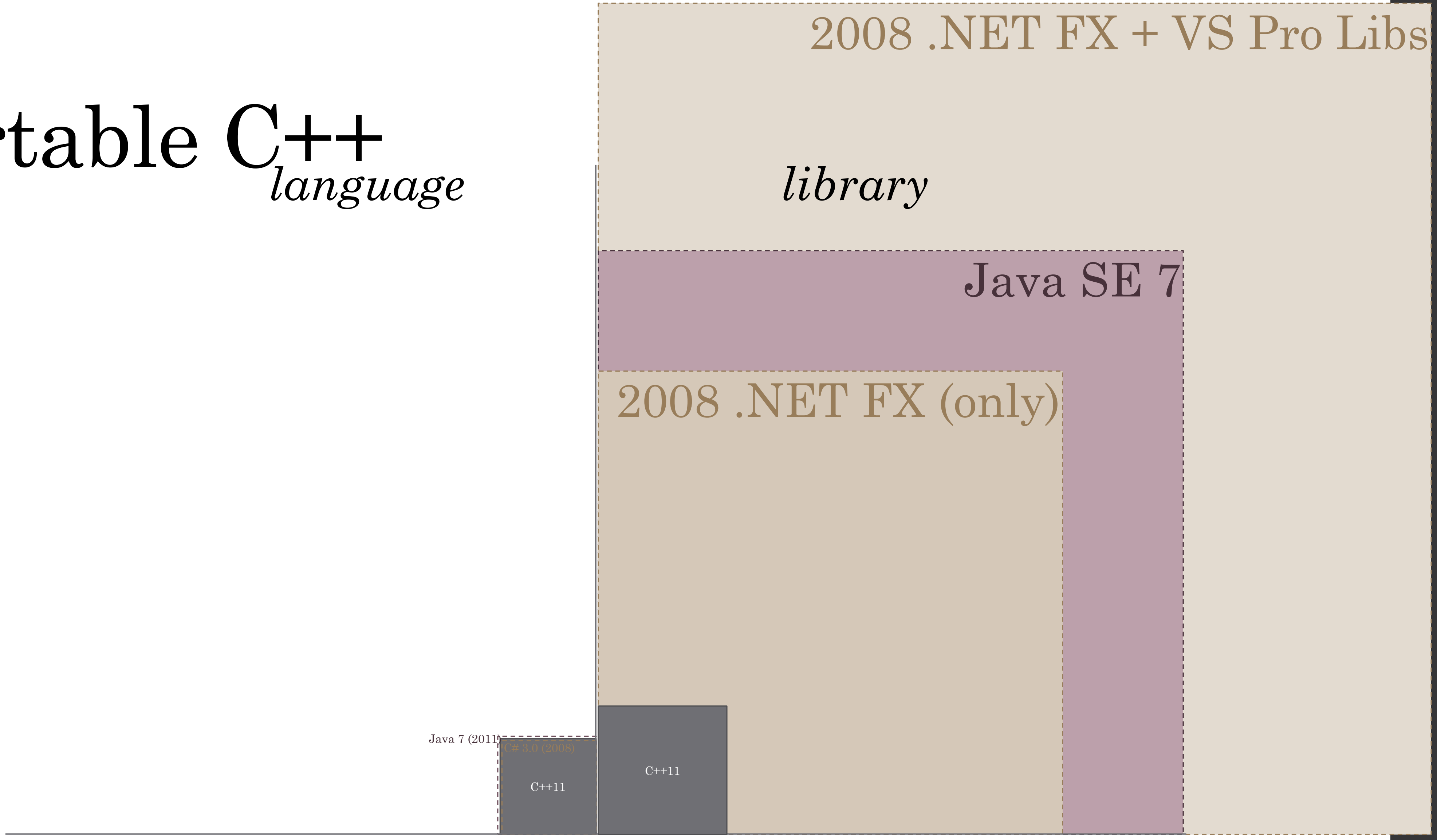
- Very early attempt at integrating with google mock.
  - https://github.com/jbandela/gmock_cross_compiler_interface
  - You can mock the implementation of an interface. Uses the interface definition without requiring MOCK_METHOD, etc.
- Cross-compiler leveldb wrapper for windows – proof of concept
  - https://github.com/jbandela/leveldb_cross_compiler
  - https://code.google.com/p/jrb-windows-builds/downloads/list
  - LevelDB is painful to build on Windows
  - Provides a dll built with gcc that can be used from MSVC or gcc
  - Currently needs rebuilt with latest cross_compiler_interface
- Use cross_compiler_interface for COM components
  - Unknown Interfaces are already COM components – they support IUnknown
  - Library has hooks that could be used to support Idispatch
- Use cross_compiler_interface for WinRT
  - Preliminary (very alpha quality) code at
  - https://github.com/jbandela/cc_winrt
- Library for writing http/https servers
  - Got distracted and wrote https://github.com/jbandela/cpp_async_await

Benefits

- Modularity
- Upgrade/change compilers/libraries without breaking compatibility
- Allow plugins to be written easily with any compliant compiler

- Make it easy to create prebuilt components that work with multiple compilers

Portable C++



From Herb Sutter's presentation *C++11, VC++11, and Beyond*

Questions/Comments

WinRT

```
import "inspectable.idl";
#define COMPONENT_VERSION 1.0
namespace WRLWidgetComponent
{
    runtimeclass Widget;
    [exclusiveto(Widget)]
    [uuid(ada06666-5abd-4691-8a44-56703e020d64)]
    [version(1.0)]
    interface IWidget : IInspectable
    {
        HRESULT GetNumber([out] [retval] int* number);
    }
    [exclusiveto(Widget)]
    [uuid(5b197688-2f57-4d01-92cd-a888f10dcd90)]
    [version(1.0)]
    interface IWidgetFactory : IInspectable
    {
        HRESULT CreateInstance1([in] int value,[out] [retval] Widget** widget);
        HRESULT CreateInstance2([in] int value,[in] int value2,[out] [retval] Widget** widget);
    }

    [activatable(1.0)]
    [activatable(IWidgetFactory, 1.0)]
    [version(1.0)]
    runtimeclass Widget
    {
        [default] interface IWidget;
    }
}
```

WinRT

```
// Define the Interface for the Widget
struct InterfaceWidget{
    // Every interface needs a unique uuid
    typedef
cc_winrt::uuid<0xada06666,0x5abd,0x4691,0x8a,0x44,0x56,0x70,0x3e,0x02,0x0d,0x64>
uuid;

    // Define the member functions of the interface
    std::int32_t GetNumber();

    // Defines the interface
    CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceWidget,GetNumber);
};
```

WinRT

```
// Define the Widget Factory
struct InterfaceWidgetFactory{

    // Every interface needs a unique uuid
    typedef cc_winrt::uuid<0x5b197688,0x2f57,0x4d01,0x92,0xcd,0xa8,0x88,0xf1,0x0d,0xcd,0x90> uuid;

    typedef cc_winrt::use_unknown<InterfaceWidget::Interface> IWidget;

    // Define the member functions of the interface
    IWidget CreateInstance1(std::int32_t);
    IWidget CreateInstance2(std::int32_t, std::int32_t);

    CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceWidgetFactory,CreateInstance1, CreateInstance2);
};
```

WinRT

```
// Tells what the RuntimeClassName is
inline cc_winrt::hstring WidgetRuntimeClassName(){return L"WRLWidgetComponent.Widget";}

// Define a runtime class
typedef cc_winrt::winrt_runtime_class<WidgetRuntimeClassName,InterfaceWidget::Interface, InterfaceWidgetFactory::Interface,cc_winrt::InterfaceInspectable> Widget_t;

// Define a typedef for use_winrt_runtime_class
typedef cc_winrt::use_winrt_runtime_class<Widget_t> Widget;
```

WinRT

```cpp
// To implement a widget derive from cc_winrt::implement_winrt_runtime_class
struct ImplementWidget :public cc_winrt::implement_winrt_runtime_class<ImplementWidget,Widget_t>
{

    int number_;

    // Implementation of the interface
    std::int32_t GetNumber(){
        return number_;
    }

    // cc_winrt will automatically map from factory interface to Constructors
    ImplementWidget():number_(0){ }
    ImplementWidget(std::int32_t i):number_(i){}
    ImplementWidget(std::int32_t i,std::int32_t j):number_(i+j){}

};
```

WinRT

```
// Default constructed
   Widget w;

   // Call Function - notice real return
   auto a = w.GetNumber();

   // Constructed with int parameter
   Widget w2(42);
   auto a2 = w2.GetNumber();

   // We have another constructor that takes 2 parameters
   Widget w3(42,7);
   auto a3 = w3.GetNumber();
```

Define_interface

```
 template<class b,template<class> class Base = InterfaceBase >
    struct define_interface:public Base<b>{

        enum{base_sz = 
sizeof(Base<size_only>)/sizeof(cross_function<Base<size_only>,0,void()>)};

        typedef define_interface base_t;
    };
```

Performance tests

```
struct VirtualInterface:public portable_base{
    virtual void f0() = 0;
    virtual int f1() = 0;
    virtual int f2(int) = 0;
    virtual std::string f3() = 0;
    virtual void f4(const std::string&) = 0;
    virtual void f5(std::string) = 0;
    virtual const char* f6(std::size_t* count)=0;
    virtual void f7(const char* pchar, std::size_t count) = 0;
    virtual void f8(std::string*) = 0;
};
```

Performance tests

```cpp
template<class T>
struct TestInterface1:public cross_compiler_interface::define_interface<T>{
    cross_function<TestInterface1,0,void()>f0;
    cross_function<TestInterface1,1,int()> f1;
    cross_function<TestInterface1,2,int(int)> f2;
    cross_function<TestInterface1,3,std::string()>f3;
    cross_function<TestInterface1,4,void(cr_string)>f4;
    cross_function<TestInterface1,5,void(std::string)> f5;
    cross_function<TestInterface1,6,void(out<std::string>)> f8;


TestInterface1():f0(this),f1(this),f2(this),
    f3(this),f4(this),f5(this),f8(this){}

};
```

```cpp
// wrl-consume-component.cpp
// compile with: runtimeobject.lib
#include <Windows.Foundation.h>
#include <wrl\wrappers\corewrappers.h>
#include <wrl\client.h>
#include <stdio.h>

using namespace ABI::Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}
```

```
int wmain()
{
    // Initialize the Windows Runtime.
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    // Get the activation factory for the IUriRuntimeClass interface.
    ComPtr<IUriRuntimeClassFactory> uriFactory;
    HRESULT hr = GetActivationFactory(HStringReference(RuntimeClass_Windows_Foundation_Uri).Get(), &uriFactory);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

```

```cpp
// Create a string that represents a URI.
HString uriHString;
hr = uriHString.Set(L"http://www.microsoft.com");
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}

// Create the IUriRuntimeClass object.
ComPtr<IUriRuntimeClass> uri;
hr = uriFactory->CreateUri(uriHString.Get(), &uri);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}

// Get the domain part of the URI.
HString domainName;
hr = uri->get_Domain(domainName.GetAddressOf());
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

```
    // Print the domain name and return.
    wprintf_s(L"Domain name: %s\n", domainName.GetRawBuffer(nullptr));

    // All smart pointers and RAII objects go out of scope here.
}
/*
Output:
Domain name: microsoft.com
*/
```

```cpp
int main(){
    try{
        // Initialize/deinitialize WinRT
        cc_winrt::unique_ro_initialize init;

        CUri uri(L"http://www.microsoft.com");

        std::wcout << L"Domain name: " << uri.GetDomain().c_str() << std::endl;

        std::wcout << L"Absolute Canonical Uri: " << uri.AbsoluteCanonicalUri().c_str() << std::endl;

        std::wcout << uri.static_interface().EscapeComponent(L"http://www.test.com/this is a test").c_str();

    }

    catch(std::exception& e){
        std::cerr << "Error. " << e.what() << "\n";
    }
}
```

```
struct InterfaceUriRuntimeClass{

        // Define a typedef for hstring
        typedef cc_winrt::hstring hstring;

        // Declare Interface so we can use it in our class
        template<class T> struct Interface;

        // Define the UUID for the class
        typedef cc_winrt::uuid<0x9E365E57,0x48B2,0x4160,0x95,0x6F,0xC7,0x38,0x51,0x20,0xBB,0xFC> uuid;

        hstring GetAbsoluteUri();
        hstring GetDisplayUri();
        hstring GetDomain();
        hstring GetExtension();
        hstring GetFragment();
        hstring GetHost();
        hstring GetPassword();
```

```
hstring GetPath();
hstring GetQuery();

// Change so we don't have to define Iwwwformdecoder
cc_winrt::use_interface<cc_winrt::InterfaceInspectable> GetQueryParsed();

hstring GetRawUri();
hstring GetSchemeName();
hstring GetUserName();
hstring GetPort();
boolean GetSuspicious();
boolean Equals(cc_winrt::use_unknown<Interface>);
cc_winrt::use_unknown<Interface> CombineUri(hstring);
```

```
CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceUriRuntimeClass,GetAbsoluteUri,GetDisplayUri,GetDomain,GetExtension,GetFragment,GetHost,GetPassword,GetPath,
GetQuery,GetQueryParsed,GetRawUri,GetSchemeName,GetUserName,GetPort,GetSuspicious,Equals,CombineUri);

    };
```

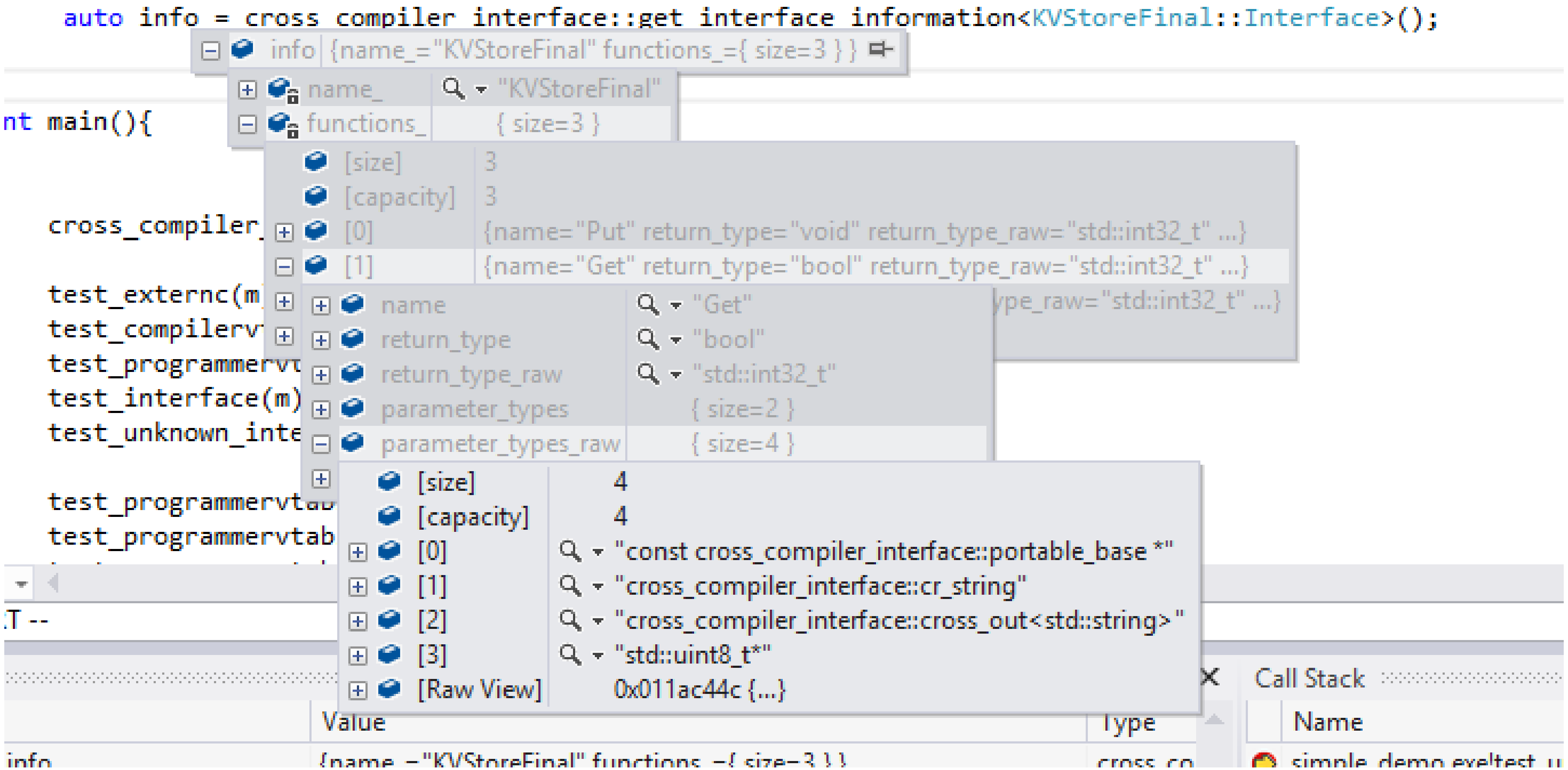
```
 //[uuid(758D9661-221C-480F-A339-50656673F46F)]
        //[version(0x06020000)]
        //[exclusiveto(Windows.Foundation.Uri)]
        //interface IUriRuntimeClassWithAbsoluteCanonicalUri : IInspectable
        //{
        //    [propget] HRESULT AbsoluteCanonicalUri([out] [retval] HSTRING* value);
        //    [propget] HRESULT DisplayIri([out] [retval] HSTRING* value);
        //}

    struct InterfaceUriRuntimeClassWithAbsoluteCanonicalUri{
        typedef cc_winrt::uuid<0x758D9661,0x221C,0x480F,0xA3,0x39,0x50,0x65,0x66,0x73,0xF4,0x6F> uuid;
        cc_winrt::hstring AbsoluteCanonicalUri();
        cc_winrt::hstring DisplayIri();

        CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceUriRuntimeClassWithAbsoluteCanonicalUri,AbsoluteCanonicalUri,DisplayIri);
    };
```

```
 //    [uuid(C1D432BA-C824-4452-A7FD-512BC3BBE9A1)]
        //[exclusiveto(Windows.Foundation.Uri)]
        //[version(0x06020000)]
        //interface IUriEscapeStatics : IInspectable
        //{
        //    HRESULT UnescapeComponent([in] HSTRING toUnescape, [out] [retval] HSTRING* value);
        //    HRESULT EscapeComponent([in] HSTRING toEscape, [out] [retval] HSTRING* value);
        //}

    struct InterfaceUriEscapeStatics{
        typedef cc_winrt::uuid<0xC1D432BA,0xC824,0x4452,0xA7,0xFD,0x51,0x2B,0xC3,0xBB,0xE9,0xA1> uuid;

        cc_winrt::hstring UnescapeComponent(cc_winrt::hstring toUnescape);
        cc_winrt::hstring EscapeComponent(cc_winrt::hstring toUnescape);

        CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceUriEscapeStatics,UnescapeComponent,EscapeComponent);
    };
```

```
struct InterfaceUriRuntimeClassFactory{
    typedef cc_winrt::hstring hstring;
    typedef cc_winrt::uuid<0x44A9796F,0x723E,0x4FDF,0xA2,0x18,0x03,0x3E,0x75,0xB0,0xC0,0x84> uuid;
    cc_winrt::use_unknown<InterfaceUriRuntimeClass::Interface> CreateUri(hstring);
    cc_winrt::use_unknown<InterfaceUriRuntimeClass::Interface> CreateWithRelativeUri(hstring,hstring);

    CC_WINRT_CONSTRUCT_INSPECTABLE_INTERFACE(InterfaceUriRuntimeClassFactory,CreateUri,CreateWithRelativeUri);
};
```

```
inline  cc_winrt::hstring FoundationUri(){return L"Windows.Foundation.Uri";}
typedef cc_winrt::winrt_runtime_class<FoundationUri,
InterfaceUriRuntimeClass::Interface,
InterfaceUriRuntimeClassFactory::Interface,
InterfaceUriEscapeStatics::Interface,
InterfaceUriRuntimeClassWithAbsoluteCanonicalUri::Interface> ClassUri_t;

typedef cc_winrt::use_winrt_runtime_class<ClassUri_t> CUri;
```

Introspection



Introspection

Autos

Name	Value	Type
info	{name_="KVStoreFinal" functions_={ size=3 } }	cross_compiler_interface
name_	"KVStoreFinal"	std::basic_string<char,st
functions_	{ size=3 }	std::vector<cross_comp
[size]	3	int
[capacity]	3	int
[0]	{name="Put" return_type="void" return_type_raw="std::int32_t" ...}	cross_compiler_interface
[1]	{name="Get" return_type="bool" return_type_raw="std::int32_t" ...}	cross_compiler_interface
name	"Get"	std::basic_string<char,st
return_type	"bool"	std::basic_string<char,st
return_type_raw	"std::int32_t"	std::basic_string<char,st
parameter_types	{ size=2 }	std::vector<std::basic_st
[size]	2	int
[capacity]	2	int
[0]	"cross_compiler_interface::cr_string"	std::basic_string<char,st
[1]	"cross_compiler_interface::out<std::string>"	std::basic_string<char,st
[Raw View]	0x011ac43c {...}	std::vector<std::basic_st
parameter_types_raw	{ size=4 }	std::vector<std::basic_st
[size]	4	int
[capacity]	4	int
[0]	"const cross_compiler_interface::portable_base *"	std::basic_string<char,st
[1]	"cross_compiler_interface::cr_string"	std::basic_string<char,st
[2]	"cross_compiler_interface::cross_out<std::string>"	std::basic_string<char,st
[3]	"std::uint8_t*"	std::basic_string<char,st
[Raw View]	0x011ac44c {...}	std::vector<std::basic_st
call	{_Callee={...} _Myal={...} }	std::function<cross_com
[2]	{name="Delete" return_type="bool" return_type_raw="std::int32_t" ...}	cross_compiler_interface
[Raw View]	0x00d0fa30 {...}	std::vector<cross_comp