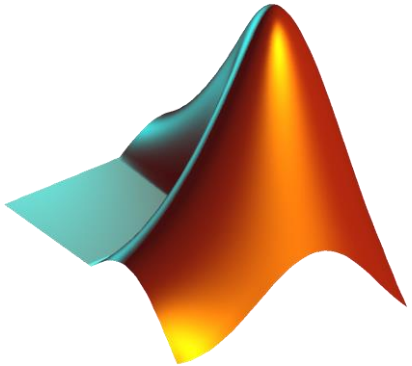
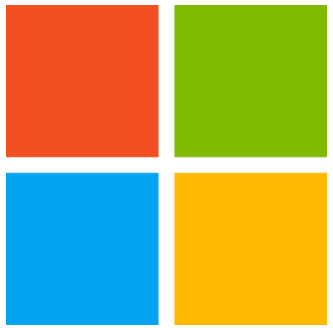
Making Libraries Consumable for Non-C++ Developers

AARON ROBINSON

Who am I?

Still at Microsoft, now on the .NET Core runtime team.

- https://github.com/dotnet/runtime

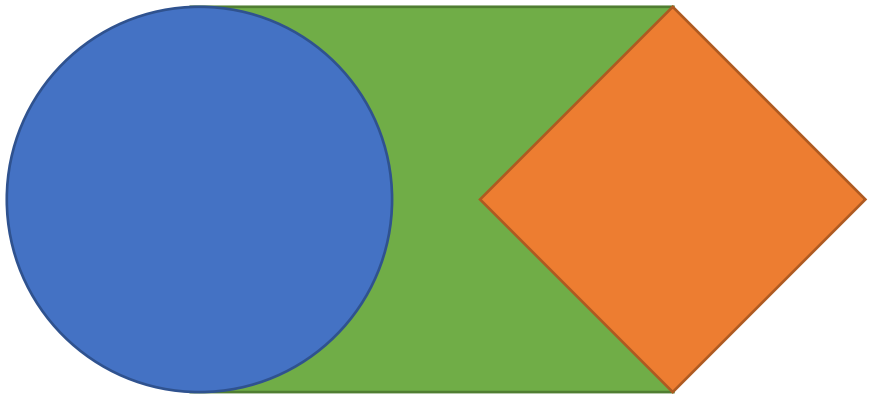
What is interoperability?

Enabling two or more disparate entities to work together.

Glue code?

Boiler plate?

A nightmare?



Don’t touch it!

What is interoperability?

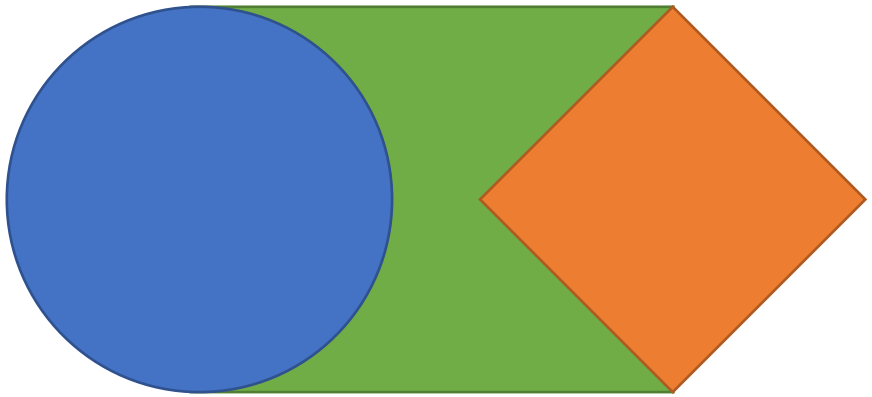
Enabling two or more disparate entities to work together.

Glue code?

Boiler plate?

A nightmare?

Don’t touch it!



Application binary interface (ABI)

Calling conventions

Marshalling

Was I supposed to free that?

Was I **not** supposed to free that?

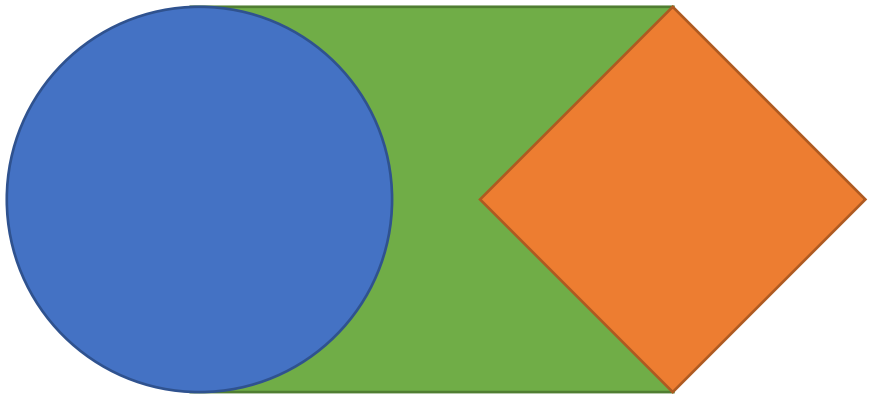
Why interoperability?

No language or platform is good for everything.

Fast inner loop

Makes UX easy

Has tooling for workload



High performance

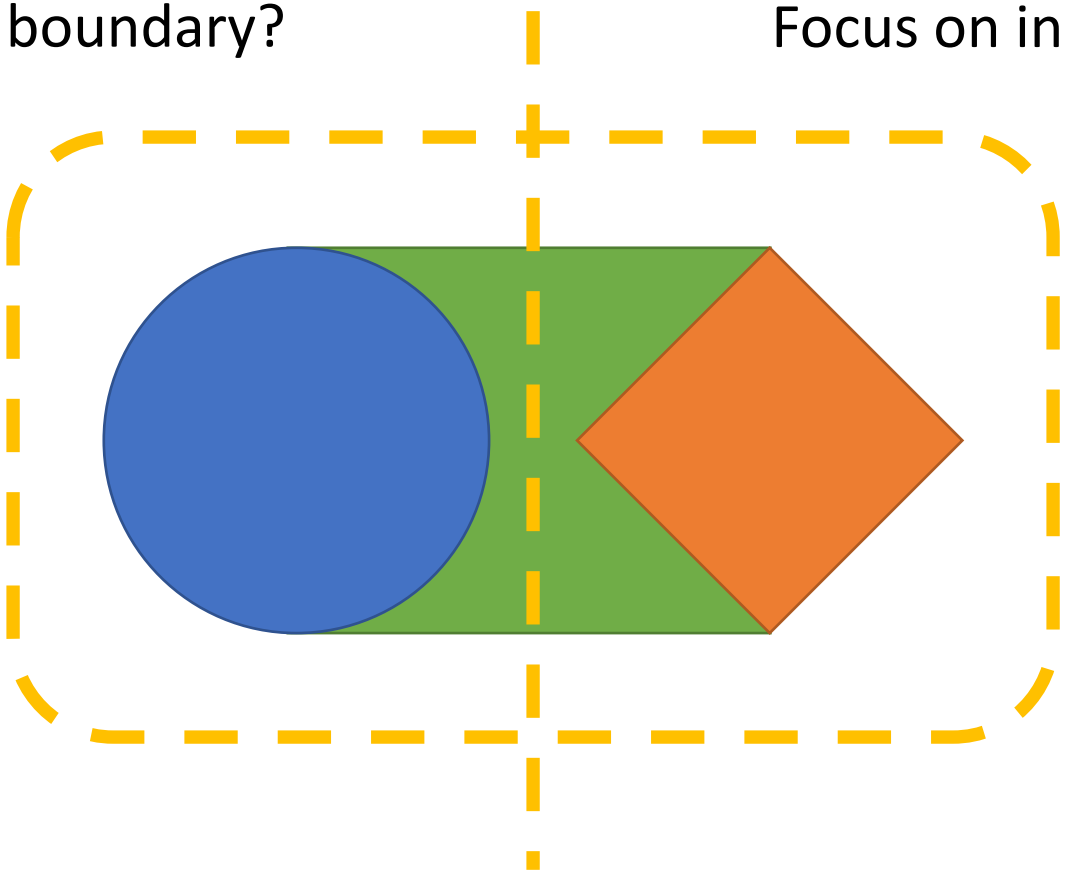
Avoids costly abstractions

What the vendor provides

A quick note on the details in this talk.

Where is the process boundary?

Focus on in-process interoperability.



Run down of **some** approaches

Just be like C? – post-1972

Common Object Model (COM) – 1993

Foreign function interface (libffi) – 1996

Simplified Wrapper and Interface Generator (SWIG) – 1996

JVM – Java Native Interface (JNI) – 1997

.NET – Platform Invoke (P/Invoke), COM interop, C++/CLI – 2002, 2005

JVM – Java Native Access (JNA) – 2007

Go – cgo – permit C in the `.go` source file – 2009

Swift – share a runtime and be like C – 2014

There is no one approach.

Make it suck less by recognizing assumptions.

What assumptions are being made?

```
/* Opens the device with name ‘dev’.
   On failure to open, returns SIZE_MAX. */
size_t open_device(char const* dev);
size_t open_device(std::wstring_view const dev);
```

The types `char` and `wchar_t` do not indicate encoding.

The size of `wchar_t`:

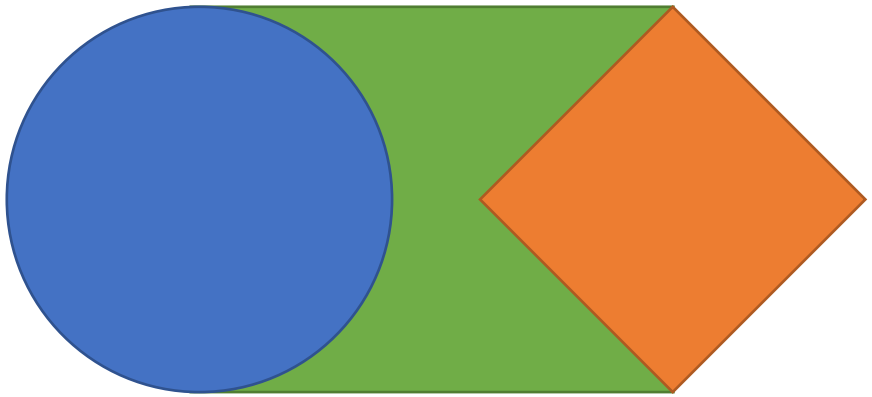
- Windows, `sizeof(wchar_t) == 2`
- Non-Windows, `sizeof(wchar_t) == 4`

`std::basic_string<CharT>` has memory implications.
More on that later.

What assumptions are being made?

```
void get_size(size_t dev, long* size);
```

Non-C/C++ language.
Caller of `get_size()`.



C/C++ binary.
Provides `get_size()`.

gcc and clang, `sizeof(long) == sizeof(size_t)`
MSVC, `sizeof(long) == 4`
Cygwin compile of gcc, `sizeof(long) == sizeof(size_t)`
MSYS2 compile of gcc, `sizeof(long) == 4`

You can make interop suck less by…

Explicitly state/document argument content.

- Instead of `long` or `int`, use `int64_t` or `int32_t`.
- String encoding is not the same as “width”.

What isn’t being declared?

```
struct data_t {
    int a; int b;
};
/* Get data from device ‘dev’. */
data_t get_data_from(size_t dev);
```

```
data_t d = get_data_from(dev);
return d.a + d.b;
```

What defines how dev is passed or data_t is returned?

Calling conventions… sigh.

Which one is being used here?

Caller cleanup (cdecl)

```
push    ...
call    data_t get_data_from(unsigned int)
add     esp, 4
add     eax, edx
```

Callee cleanup (stdcall)

```
push    ...
call    data_t get_data_from(unsigned int)
add     eax, edx
```

What isn’t being declared?

```
struct data_t {
    int a; int b;
};
/* Get data from device ‘dev’. */
data_t get_data_from(size_t dev);
```

```
data_t d = get_data_from(dev);
return d.a + d.b;
```

```
class dev_t {
public:
/* Get data from this device. */
virtual data_t get_data_from() = 0;
};
```

```
data_t d = dev->get_data_from();
return d.a + d.b;
```

Assuming callee cleanup and focusing on `data_t`, is its return location consistent?

What isn't being declared?

```
data_t d = get_data_from(dev);
return d.a + d.b;
```

```
push    [esp-4]
call    data_t get_data_from(unsigned int)
add     eax, edx
```

The `get_data_from()` function returns the struct in registers, but the `get_data_from()` member function returns in caller provided memory.

This is often unexpected but occurs using the MSVC compiler for x86 with `stdcall` (callee cleanup) or `cdecl` (caller cleanup).

For non-MSVC, `data_t` is always returned in a caller provided memory.

```
data_t d = dev->get_data_from();
return d.a + d.b;
```

```
sub     esp, 8
mov     eax, [esp+4]
lea     edx, [esp+8]
push    edx
push    eax
mov     ecx, [eax]
call    [ecx]
mov     eax, [esp+12]
add     eax, [esp+8]
```

What **else** isn't being declared?

```
struct data_t {
    int a; int b;
};
/* Get data from device 'dev'. */
data_t get_data_from(size_t dev);
```

How does this function fail?

What compiler flags (clang) were used by the library? By the library consumer?

-fsjlj-exceptions?
-fignore-exceptions?
-fdwarf-exceptions?
-fseh-exceptions?
etc.

C++ exceptions have no universal binary contract.
Meaning the consumer may not be prepared for a C++ exception – of any sort.
The typical result is ... undefined.

What is being declared?

```
OBJC_EXPORT id objc_msgSend(id self, SEL op, ...);
```

What is being declared?

```
OBJC_EXPORT id objc_msgSend(id self, SEL op, ...);
```

Indicates variadic arguments… but does it in this case?

```
// Incorrect usage – not really variadic argument signature.
objc_msgSend(_id, _op, 10, 1.f);

// Correct signature and usage.
((void(*)(id,SEL,int,float))objc_msgSend)(_id, _op, 10, 1.f);
```

2018: objc_msgSend() doc – `id objc_msgSend(id self, SEL op, ...);`

2019+: objc_msgSend() doc – `void objc_msgSend(void);`

You can make interop suck less by…

Explicitly state/document argument content.

- Instead of `long` or `int`, use `int64_t` or `int32_t`.
- String encoding is not the same as “width”.

Explicitly state/document/reference function conventions.

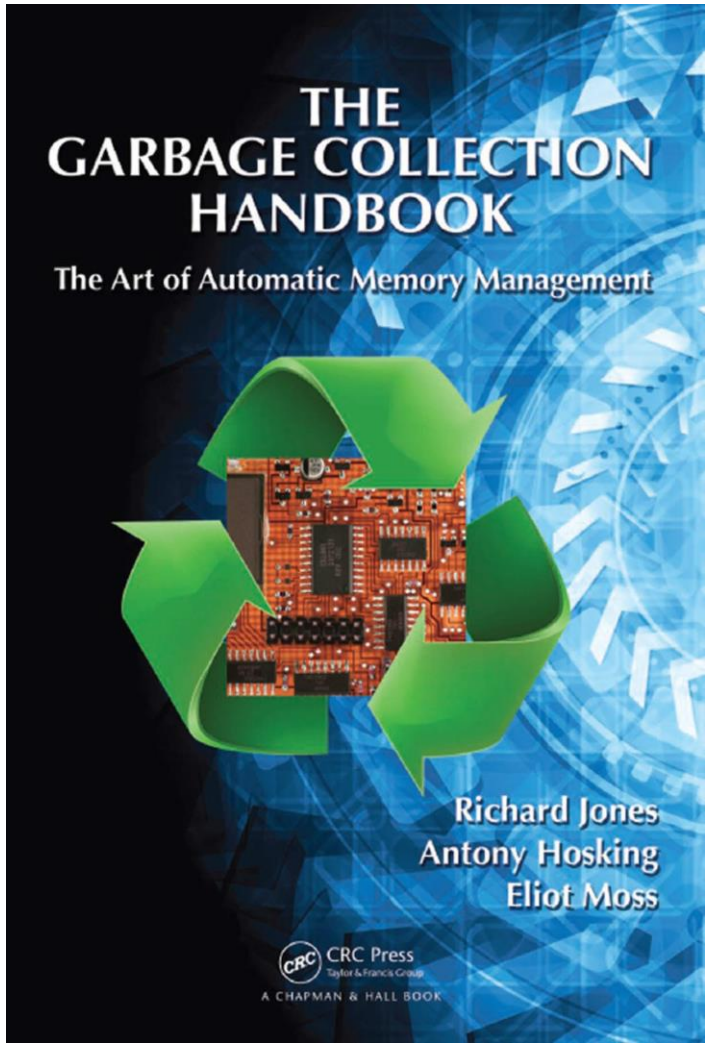
- Defining a macro for calling conventions is a great start. For example, `MYLIB_CCONV`.
- Reference: llvm - CallingConv.h
- Don’t throw exceptions across the boundary.

Memory model

Manual memory management is **rarely** advocated for anymore.

Garbage collection is really “automatic memory management”.

- Reference counted
  - C++ – `std::shared_ptr<T>`
  - Python
  - Objective-C (manual or automatic – see ARC)
  - Swift
  - COM – `AddRef()`/`Release()`
- Non-Reference counted
  - .NET
  - JVM
  - JavaScript

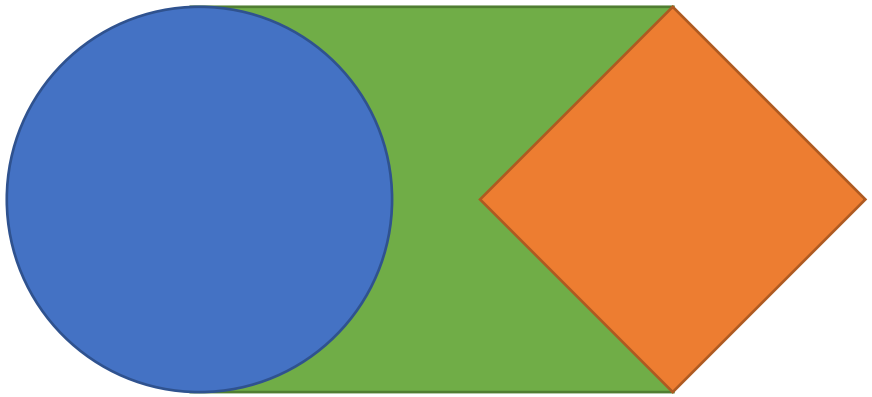


https://gchandbook.org/

Memory model – Manual

Memory is allocated and deallocated **directly**.
Allocation locations are static.

```
ptr = new int[10]; // 0x2600
```



Could have been deleted here, if allocator was known.

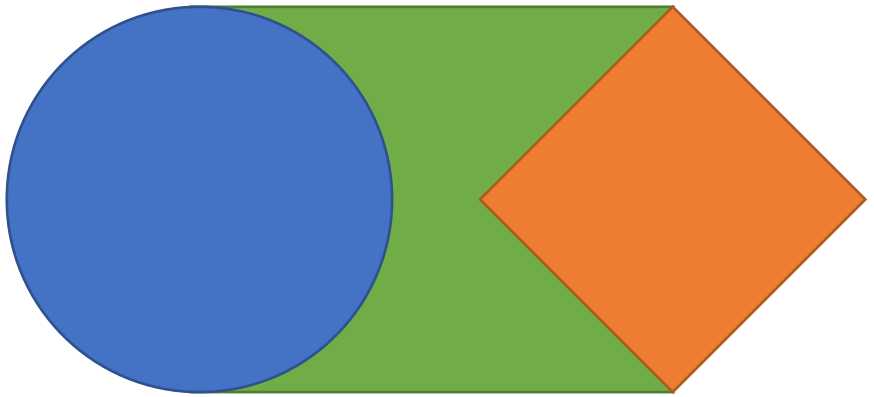
```
delete[]    ; // 0x2600
```

Memory model – Reference Counted

Memory lifetime is tracked **explicitly** through reference counting.
This **typically** means allocation locations are static.

```
// 0x5200, ref 1
obj = create();
```

```
// 0x5200, ref 2
[obj AddRef];
```



```
// 0x5200, ref 3
obj->add_ref();
```

```
// 0x5200, ref 2
[obj Release];
```

```
// 0x5200, ref 1
obj->rel_ref();
// 0x5200, ref 0
obj->rel_ref();
```

Memory model – Non-Reference Counted

Memory lifetime is tracked **implicitly** based on the type itself.
Allocated memory can be moved and must be indirectly accessed during interop scenarios.

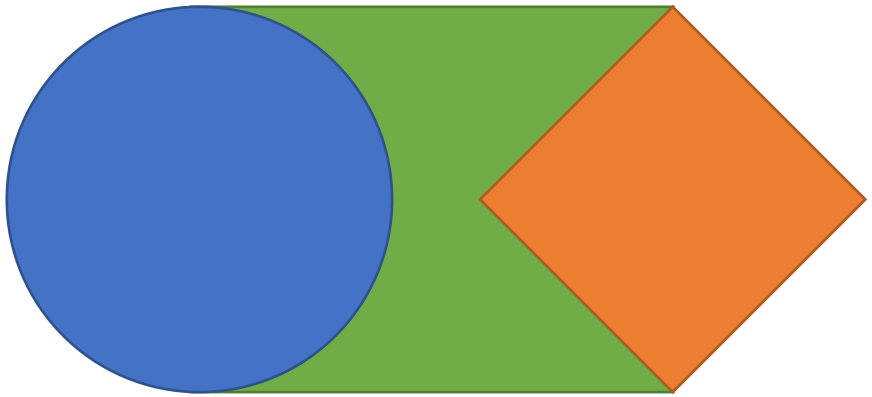
```
// 0x7800
obj = new();
// 0x9200
hnd = Handle.New(obj);

// Collection occurs.
// Copy, compacting, etc.
// obj now at 0x6800.

// Collection occurs.
// obj now at 0x7200.

Handle.Free(   );

// Collection occurs.
// obj now “free”.
```



This is only one possible sequence.

```
// 0x9200 -> 0x7800
Sys_Query(   );

// 0x9200 -> 0x6800
Sys_Query(hnd);

// 0x9200 –> 0x????
Sys_Query(hnd);

// 0x9200 -> 0x7200
Sys_Done(hnd);
```

Memory model – Non-Reference Counted

Common ways of facilitating interop scenarios with a Garbage Collector (GC).

1. Handles – Level of indirection.
   - Usually requires a “platform” API to use the memory.
     - .NET has `GCHandle`.
     - JVM, through JNI, exposes most memory as a handle – `jobject`, `jstring`, `jintArray`, etc.

2. Pinning – Tell GC to not move object.
   - The platform needs to provide a mechanism.
     - .NET has `GCHandle` and C# has `fixed` keyword.
     - Conforming JVM implementations have the option.

Memory model

Control of “shared” memory needs to be documented and/or agreed upon.

GCs make this far more complicated since they are typically non-deterministic – possible even if Reference Counting is used.

Consider accepting alloc/dealloc callbacks.

You can make interop suck less by…

Explicitly state/document argument content.

- Instead of `long` or `int`, use `int64_t` or `int32_t`.
- String encoding is not the same as “width”.

Explicitly state/document/reference function conventions.

- Defining a macro for calling conventions is a great start. For example, `MYLIB_CCONV`.
- Reference: [llvm - CallingConv.h]
- Don’t throw exceptions across the boundary.

Explicitly state/document memory ownership rules.

- Consider accepting memory alloc/dealloc callbacks – recall previous recommendation.
- Limit implicit models that force memory to have thread affinity at interop boundaries.
- Consider how the consumer’s tools work with your library’s memory model.

Conclusion

- Document what you want and assume.
- Understand assumptions and be precise when possible.
  - C++ now has many types that express precisely what is meant – integer sizes, string encoding, etc.
- Interop scenarios often aren't using a C++ compiler to read the header, humans are.
  - Kate Gregory's "What Do We Mean When We Say Nothing At All?"

Thank you.

Email: arobins@microsoft.com

GitHub: @AaronRobinsonMSFT, https://github.com/AaronRobinsonMSFT

Recent Interop ABI fun

```
struct blub_t {
    size_t a; int b;
};
size_t DoTheThing(blub_t b);
```

Switch .NET call from

```
nint DoTheThing(BlubT b);
```

to

```
nint DoTheThing(in BlubT b);
```

Result:

Windows – everything passed.

Linux – everything failed.

Why?

Making Libraries Consumable for Non-C++ Developers

Aaron R Robinson

arobins@microsoft.com

https://github.com/AaronRobinsonMSFT