C++ interop

About Me

- I like C/D/C++
- SAoC 2023
- Researching into network bufferbloat

Targets for this talk

- Inspired from my experience when I first joined.
- Basic D programmer
  - Provide an easy path for the beginner D programmer to interoperate with C++
  - To preach the most basic level of understanding with interoperating with C++
  - Knowing your environments
  - Knowing your use cases

- You Experts
  - Reveal the state of using the STL in D
  - Stumbling blocks
  - Way forward

Truth About C++ interoperability

- C++ interoperability is a global topic.

Languages actively interoperating with C++

- Rust
- Swift
- Zig
- Python
- etc.

Their Approach ????

- Write wrappers and some other confusing stuff.

D’s approach??

- matching C++ name mangling conventions
- matching C++ function calling conventions
- matching C++ virtual function table layout for single inheritance

Walkthrough of the execution of your program.

- Compiler compiles your code
  - Stack and your heap allocated. (if there be need)
  - Stack very important for this work
- Generates an assembly instruction based on your hardware instruction set architecture
- Instructions go through the fetch decode execute cycle

Basic Rule

- THINK D!

Backbone of interop

- Semantics analysis
- Any (C++/D) interoperable routine must semantically agree

C++ libraries

- Majority of the C++ libraries are implemented with classes
- C++ Classes are value types
- D classes are reference types
- D structs are value types
- Use your structs

Thenn..

- KNOW YOUR D STRUCT!

- @disable this()
  - MSVC allocates on default initialization in debug mode
  - Just avoid this especially on windows
- Constructors
  - Can call C++ copy constructors
- Operator overloads
- destructors

Move constructors????

- If there be a need for an internal move operations, can use the phobos move function.

- But you cannot move ctors and functions.

C++ class – me.cpp

```
class A {
public:
    int * b;
    A(int a);
    A(int a, int b);
    ~A();
};

A::A(int a) {
    b = new int(a);
    std::cout <<"allocating memory";
}

A::A(int a, int b) {
    std::cout << "just run!";
}

A::~A() {
    delete(b);
    std::cout <<"deallocated from heap" << std::endl;
}
```

D counterpart struct – you.d

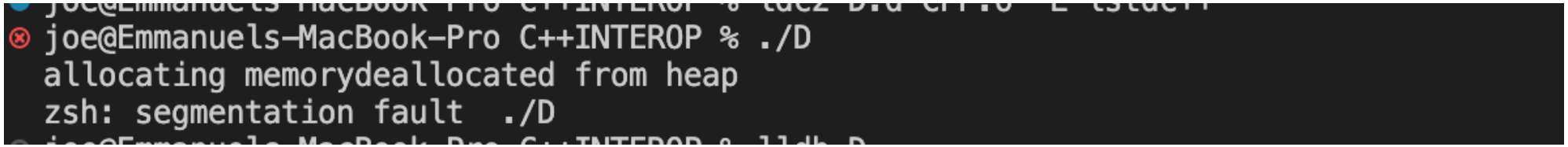
```d
extern(C++) struct A {
    this(int a);
    this(int a, int b);
    ~this();
}


void main()
{
    auto obj = A(5);
}
```

Let’s link and execute

- g++ -c me.cpp
- ldc2 you.d me.o –L-lstdc++

- ./you

Oooopsss… Segmentation fault



Stack frame for D's main – x86_64 ISA

```
__Dmain:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    -1(%rbp), %rdi
    xorl    %esi, %esi
    movl    $1, %edx
    callq   _memset
    leaq    -1(%rbp), %rdi
    movl    $5, %esi
    callq   __ZN1AC1Ei
    leaq    -1(%rbp), %rdi
    callq   __ZN1AD1Ev
    xorl    %eax, %eax
    addq    $16, %rsp
    popq    %rbp
```

C++ this(int) call stack

```
__ZN1AC2Ei:                             ## @_ZN1AC2Ei
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movq    %rdi, -8(%rbp)
    movl    %esi, -12(%rbp)
    movq    -8(%rbp), %rax
    movq    %rax, -24(%rbp)             ## 8-byte Spill
    movl    $4, %edi
    callq   __Znwm
    movq    %rax, %rcx
    movq    -24(%rbp), %rax             ## 8-byte Reload
    movl    -12(%rbp), %edx
    movl    %edx, (%rcx)
    movq    %rcx, (%rax)
    movq    __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
```

C++: movq -8(%rbp), %rax

D : leaq -1(%rbp), %rdi

Trace your stack

```
joe@Emmanuels-MacBook-Pro C++INTEROP % lldb D
(lldb) target create "D"
Current executable set to '/Users/joe/Desktop/C/C++INTEROP/D' (x86_64).
(lldb) run
Process 8829 launched: '/Users/joe/Desktop/C/C++INTEROP/D' (x86_64)
allocating memorydeallocated from heap
Process 8829 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1,
    frame #0: 0x000000010004c817 D`_d_run_main2 + 487
D`:
->  0x10004c817 <+487>: movl   -0x54(%rbp), %eax
    0x10004c81a <+490>: leaq   -0x28(%rbp), %rsp
    0x10004c81e <+494>: popq   %rbx
    0x10004c81f <+495>: popq   %r12
Target 0: (D) stopped.
(lldb)
```

Let’s call 2nd constructor

```
extern(C++) struct A {
    this(int a);
    this(int a, int b);
    ~this();
}


void main()
{
    auto obj = A(5, 9);
}
```

Let’s link and execute

- ldc2 you.d me.o –L-lstdc++
- ./you

- Execution success
- Just run!

No seg fault this time?

Consider C++ this(int, int) stack frame

```
__ZN1AC2Eii:                                ## @_ZN1AC2Eii
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movq    %rdi, -8(%rbp)
    movl    %esi, -12(%rbp)
    movl    %edx, -16(%rbp)
    movq    __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
    leaq    L_.str(%rip), %rsi
    callq   __ZNSt3__1lsB8ue170006INS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EE
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc
```

Let’s make this constructor call

```
void main()
{
    auto obj = A(5, 9);
}
```

D this(int, int) stack frame

```
callq   _memset
leaq    -1(%rbp), %rdi
movl    $5, %esi
movl    $9, %edx
callq   __ZN1AC1Eii
```

C++ is not moving any value from the address offset the base pointer.

```
## %bb.0:
__ZN1AC2Eii:                            ## @_ZN1AC2Eii
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movq    %rdi, -8(%rbp)
    movl    %esi, -12(%rbp)
    movl    %edx, -16(%rbp)
    movq    __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
    leaq    L_.str.1(%rip), %rsi
    callq   __ZNSt3__1lsB8ue170006INS_11char_traitsIcEEEERNS_13basic_ostreamIcT
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc
                                        ## -- End function
```

Takeaways

- This is why….

1. C++ can pass two member pointers
2. D can pass 4 integers
3. And still get interoperable results

It’s mainly a memory passing routine and that should be most factored

Another takewawy

- So always make room for whatever C++ wants to come and do in the call stack.
- Treat C++ like a visitor coming to your house and wants to sit down. Give your visitor the chair.

It comes with a tradeoff

Safety

RAII in D when C++ allocated heap is involed?

- Some C++ libraries do not emit their destructors in their symbol table.
- Some of these libraries allocates on initialization
- RAII takes care of destroying heap allocated resources in C++ when object is constructed in the stack frame of the C++ main.
- What about D??
- Let’s find out!

C++ class ： No destructor

```
class A {
public:
    int * b;
    A(int a);
    A(int a, int b);
    //~A();
};

A::A(int a) {
    b = new int(a);
    std::cout <<"allocating memory";
}

A::A(int a, int b) {
    std::cout << "just run!";
}
```

D struct

```d
extern(C++) struct A {
    int* p;
    this(int a);
    this(int a, int b);
}


void main()
{
    auto obj = A(5, 9);
```

NB: for this test, Linux is most reliable

- ldc2 D.d CPP.o –-fsanitize=address –L-lstdc++
- ./D
- ..... Running.

Obviously.....

- Summary: AddressSanitizer: 4 byte(s) leaked in 1 allocations

Nowww, we call destroy

```
extern(C++) struct A {
    this(int a);
    this(int a, int b);
}


void main()
{
    auto obj = A(5, 9);
    destroy!false(obj);
}
```

Sadly

- Summary: AddressSanitizer: 4 byte(s) leaked in 1 allocations

THE C++ STL

Pragmatic tradeoffs

- Reimplement ??

- Interface it and use from D?

Runtimes

- Linux is your best friend
  - Especially when linking with gcc compiled binaries
  - You get most of the symbols you desire.
  - Symbols are linkable
  - Interfacing is quite easy due to the simplified CXX-itanium ABI

Runtimes

- Linux might betray you a little
  - When compiling with clang compiled binaries.
  - libc++ libraries do not like to handle deallocations with the base class' destructor then default the base destructor
  - Rather like to do their destructions in the abstract classes.
  - If you do not keep track of those, can leave your code vulnerable to leaks.

Runtimes

- Windows is that cool friend who decides when to be good to you.
  - Visual C++ mangling scheme is very complex
  - Little bit hard to debug when finding it difficult to pick your symbols.
  - Cannot walk through the visual C++ mangling now, we will run out of time

  - Few trace points when finding it hard to debug your symbol interfacing
    - Look out for your access modifiers
      - Public and private fields are mangled differently
      - Classes and structs are mangled differently
      - Uses a rather systemic numbering system for types when working with templates but you can easily miss

Runtimes

- macOS is your worst enemy
- macOS-12
  - Emit some functions as local text symbols (t)
  - Hence not linkable
- macOS-13
  - Those functions emitted as local text symbols in 12 are emmited as global(T)
  - linkable
  - Then some other symbols emitted as global text symbols in 12 are then not emitted.
- macOS-14
  - Actually doesn’t care about you. Just emits what it wants and what it doesn’t

Work done

- Moved from druntime.
- Current dir : https://github.com/dlang/stdcpp
- Currently Managed in a dub package
- Containers worked on
  - Vectors
  - List
  - Set only works on linux
  - String

  Runtimes: Linux , Macos, Windows

Primarily C++ compiler targets

- clang++/MSVC on windows
- clang++ on macOS
- clang++/ g++ on linux

```cpp
#include <list>
#include <string>
#include <vector>
#include <set>

namespace stdcpp {
    namespace test {
        template<typename T>
        std::size_t cppSizeOf() {
            return sizeof(T);
        }

        /// Returns the result of `std::string` capacity with the provided string
        std::size_t stringCapacity (char const* str) {
            std::string s(str);
            return s.capacity();
        }
    };
};
```

Instantiating templates for our classes

```
template std::size_t stdcpp::test::cppSizeOf<std::string>();

template class std::list<int>;
template std::size_t stdcpp::test::cppSizeOf<std::list<int> >();

template class std::vector<int>;
template std::size_t stdcpp::test::cppSizeOf<std::vector<int> >();

template class std::set<int>;
template std::size_t stdcpp::test::cppSizeOf<std::set<int> >();
```

Templates instantiations

- We can instantiate our template classes for all fundamental types.
- So we can use it for chars, doubles, floats etc.

```
template std::size_t stdcpp::test::cppSizeOf<std::string>();

template class std::list<int>;
template std::size_t stdcpp::test::cppSizeOf<std::list<int> >();

template class std::vector<int>;
template std::size_t stdcpp::test::cppSizeOf<std::vector<int> >();

template class std::set<int>;
template std::size_t stdcpp::test::cppSizeOf<std::set<int> >();
```

Types are not only fundamental

```
1   import stdcpp.vector;
2
3   struct A {
4       int a;
5       int b;
6   }
7
8   void main()
9   {
10      A obj = { 3, 4};
11      auto A = vector!A(obj);
12  }
```

In our C++ file for symbols

- Definitely, we will need our

```
struct A;
template class std::vector<A>;
```

Should we do this for every aggregate?

- Aggregates name can be anything
- Infinite number of possible names for your aggregates.

A simple demo

Std::String test

```
}

unittest
{
    auto a = std_string("hello");
    a.push_back('a');
    assert(a.size() == 6);
    // verifying small string optimization, this is 15 on GCC, 22-23 on clang
    assert(a.capacity == stringCapacity("helloa"));
    assert(a.front() == 'h');
    assert(a.back() == 'a');
    a.resize(4); // shrinks a to "hell"
    assert(a.size() == 4);
    immutable LongStr = "Hi, this is a test for string capacity growth for a length r
    auto b = std_string(LongStr);
    assert(b.capacity == stringCapacity(LongStr.ptr));
    a.swap(b); // a and b swaps
    assert(a.capacity == stringCapacity(LongStr.ptr));
    assert(b.capacity == stringCapacity("hell")); // a was shrinked to hell so b cont
    b.pop_back();
    assert(b.size() == 3);
    assert(b[0] == 'h');
    assert(b[1] == 'e');
    assert(a.empty == 0);
    a.clear();
    assert(a.empty == 1);
}
```

list test

```
unittest
{
    auto p = list!int(5);
    p.push_back(5);
    assert(p.size() == 6);
    assert(p.front() == 0);
    assert(p.back() == 5);
    p.push_front(7);
    assert(p.front() == 7);
    p.clear();
    assert(p.size() == 0);
    p.assign(5,5);
    assert(p.size == 5);
    p.pop_front();
    assert(p.size == 4);
    p.resize(3);
    assert(p.size == 3);
```

Vector test

```
unittest
{
    auto vec = vector!int(4);
    vec.push_back(42);
    assert(vec.length == 5);
    assert(vec[4] == 42);
    assert(vec.at(3) == 0);
    vec.pop_back();
    assert(vec.length == 4);
    vec.clear();
    assert(vec.empty == 1);
    vec.push_back(7);
    vector!int new_vec = vec;
    auto it = new_vec.begin();
    assert(*(it) == 7);
}
```

Set test

```d
import stdcpp.allocator;
allocator!int alloc_instance = allocator!(int).init;
less!int a;
auto p = set!int(a);
p.insert(5);
assert(p.size == 1);
assert(p.empty == 0);
p.erase(5);
p.insert(6);
p.clear;
assert(p.size == 0);
assert(p.empty == 1);
set!int q = a;
q.swap(p);
q.insert(4);
q.insert(4);
q.insert(4);
assert(q.size == 1);
assert(q.count(4) == 1);//count for set only results in 0 for not pr
assert(q.count(5) == 0);
assert(q.contains(4) == 1); // q contains 4 evaluates to true
auto iter = q.find(4);
set!int w = q; //copy constructor
assert(w.size() == 1);
```