Bringing Clang and LLVM to Visual C++ users

Reid Kleckner
Google

C++ devs demand a good toolchain

- Fast build times
- Powerful optimizations: LTO, etc
- Helpful diagnostics
- Static analyzers
- Dynamic instrumentation tools: the sanitizers
- New language features: C++11

LLVM has these on Mac/Linux, but not Windows

What does LLVM need for Windows?

- Need to support the existing platform
  - ABIs, external libraries, system libraries, etc.
- Indistinguishable for the users
  - Produces the *same* application
  - No wedges, shims, or layers for compatibility
- Need to support the existing development env
  - Drop-in compatible, deep integration the IDE

MSVC ABI compatibility is important

- Without ABI compat, must compile the world
  - Cannot use standard C++ libraries like ATL, MFC, or MSVC's STL
  - Cannot use third party C++ libs or dlls
  - Can only use extern "C" and COM interfaces
  - Impossible to wrap extensions like C++/CX
- Even if you recompile, you must port code
  - Must port to a new standard library
  - Must remove language extensions and inline asm
  - Must port third party code you don't own
  - No incremental migration path: all or nothing
- All before you can even try Clang/LLVM

Visual Studio is important

- Visual Studio is the gold standard for IDEs
  - Integration is a must for real users
  - Try asking users to run ‘make’
- Need to be able to use tools from VS
  - clang-cl provides cl.exe CLI compatibility
  - lld provides link.exe CLI compatibility
- Clang and LLVM: Integrated into your Development Environment

How do we get there?

Challenges to surmount

- C++ ABI is completely undocumented
- File formats are an unknown moving target
- Large language extensions employed throughout system headers
- ATL and MFC headers use invalid C++ templates
- LLVM linker was essentially non-existent

My focus has been the C++ ABI in clang

What’s in a C++ ABI?

Everything visible across a TU boundary:

- Name mangling: overloads and namespaces
- Record layout: vptrs, alignment, bitfields
- Vtable layout: destructors, overloads
- Calling conventions: __cdecl vs __thiscall
- C++ arcana: “initializers for static data members of class templates”

This all matters for compatibility!

How to test a C++ ABI

Write compiler A/B integration tests

```
struct S { int a; };
void foo(S s);

#ifdef COMPILER_A
void foo(S s) {      // TU1
  CHECK_EQ(1, s.a); // Verify we got the S data
}

#else // COMPILER_B
int main() {         // TU2
  S s;
  s.a = 1;
  foo(s);  // Pass S by value
}
#endif
```

MSVC compatibility affects all layers

- All layers: handle language extensions
  - delayed templates, declspec, __uuidof...
- AST: LLVM IR independent
  - Record layout: sizeof, __offsetof, __alignof
  - Name mangler
  - Vtable layout
- CodeGen: Generating LLVM IR
  - Virtual call lowering
  - Member pointers
  - Lowering pass-by-value
- Most work is in CodeGen

In every ABI, there are corner cases

- To analyze the ABI, we write tests for MSVC
- There are no docs, only tests, so we often uncover dark, untested ABI corners
- Sometimes MSVC crashes
  - Template instantiation with a null pointer to member function of a class that inherits virtually
- Sometimes MSVC produces invalid COFF
  - Two statics in inline functions with the same name
- Sometimes valid C++ is miscompiled
  - Passing pointer to member of an incomplete type
  - Casting to a pointer to member of a base class

Basic name mangling

```
namespace space { int foo(Bar *b); }
```

```
?foo@space@@YAHPAUBar@@@Z
```

```
_ZN5space3fooEP3Bar
```

Microsoft symbols are invalid C identifiers, ? prefix

Itanium symbols are reserved C identifiers, _Z prefix

Basic name mangling

```
namespace space { int foo(Bar *b); }
```

```
?foo@space@@YAHPAUBar@@@Z
```

```
_ZN5space3fooEP3Bar
```

Namespace first in Itanium

Basic name mangling

```
namespace space { int foo(Bar *b); }

?foo@space@@YAHPAUBar@@@Z

_ZN5space3fooEP3Bar
```

Function name first in Microsoft

Basic name mangling

```
namespace space { int foo(Bar *b); }

?foo@space@@YAHPAUBar@@@Z

_ZN5space3fooEP3Bar
```

Parameters last in both

All very reasonable

Names of static locals

- Static locals must be named and numbered:

```
inline void foo(bool a) {
  static int b = use(&b); // foo::2::b
  if (a)
    static int b = use(&b); // foo::4::b
  else
    static int b = use(&b); // foo::5::b
}
```

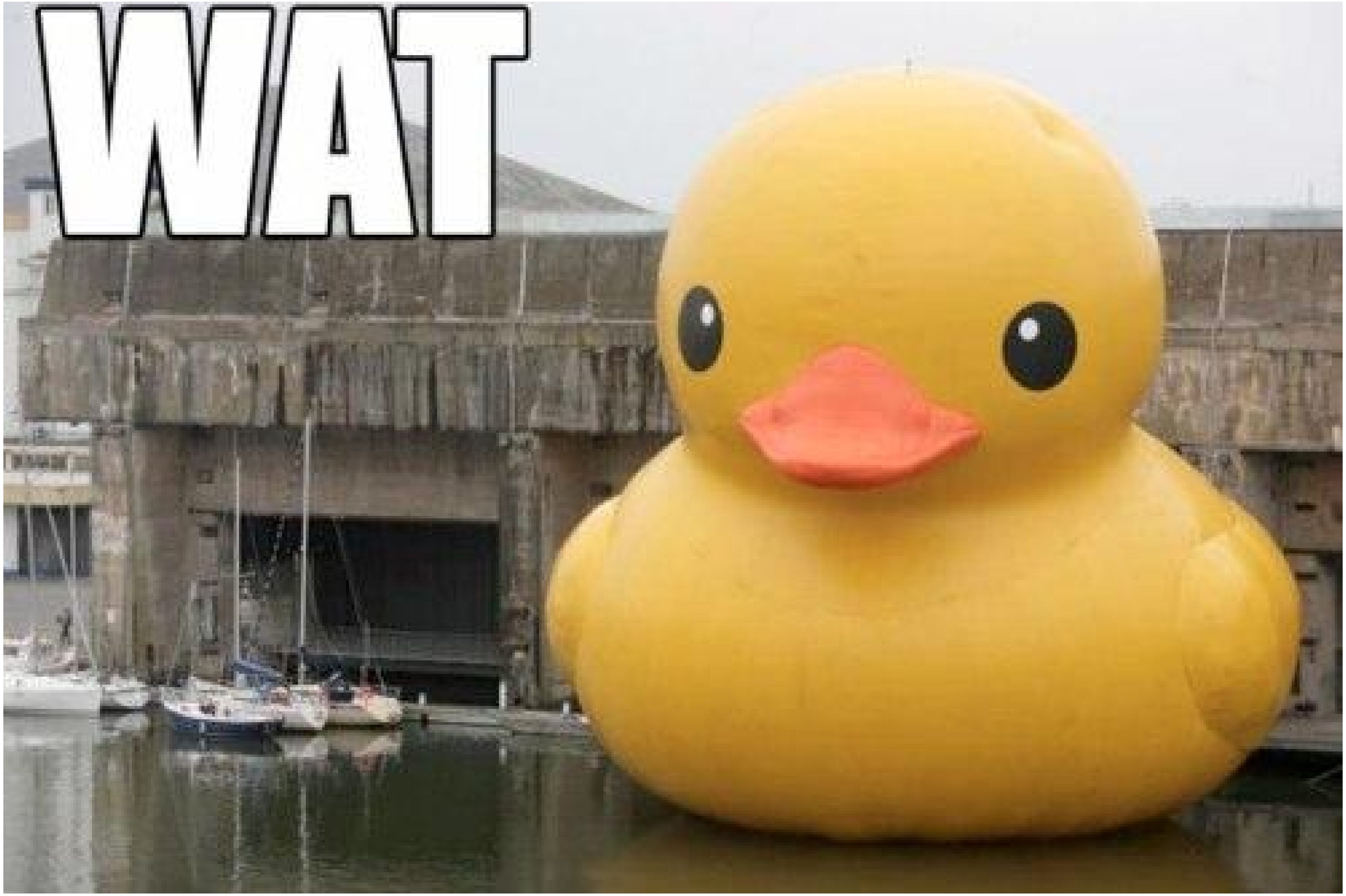
- The number appears to be the count of scopes entered at point of declaration

Names of static locals

- Variables can be declared without entering a scope

```
inline void foo(bool a) {
  if (a)
    static int b = use(&b); // foo::4::b
  static int b = use(&b);   // foo::4::b !!
}
```

- Compiles successfully
- Linker aborts due to invalid COFF, duplicate COMDAT group


WAT

Unnamed structs often need names

- MSVC appears to name <unnamed-tag>
- This code gives the diagnostic:

```
struct { void f() { this->g(); } };
```

'g' : is not a member of '<unnamed-tag>'

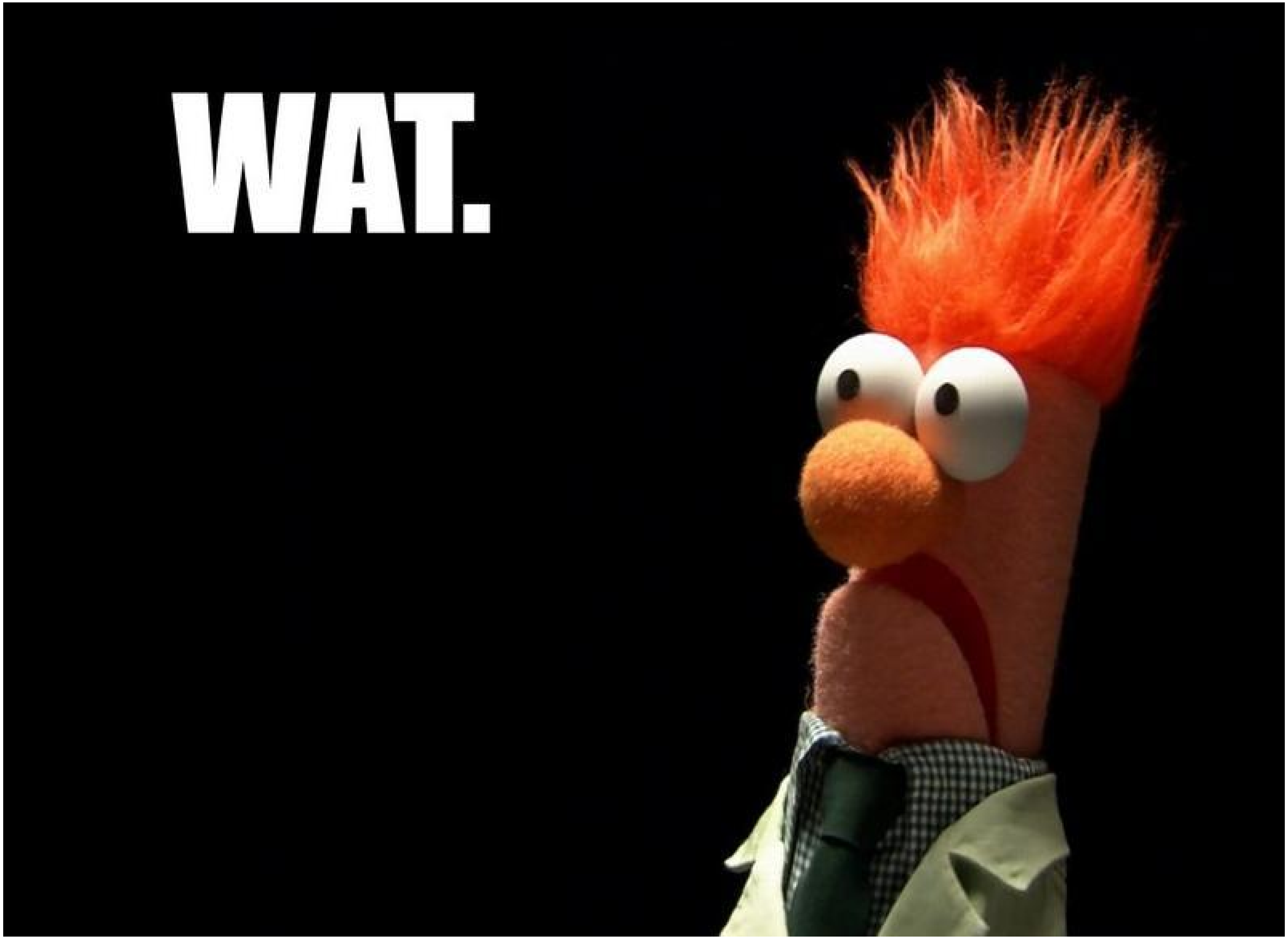
Unnamed struct mangling

The vftable of an unnamed struct is named:

`??_7<unnamed-tag>@@6B@`

This program prints ‘b’ twice:

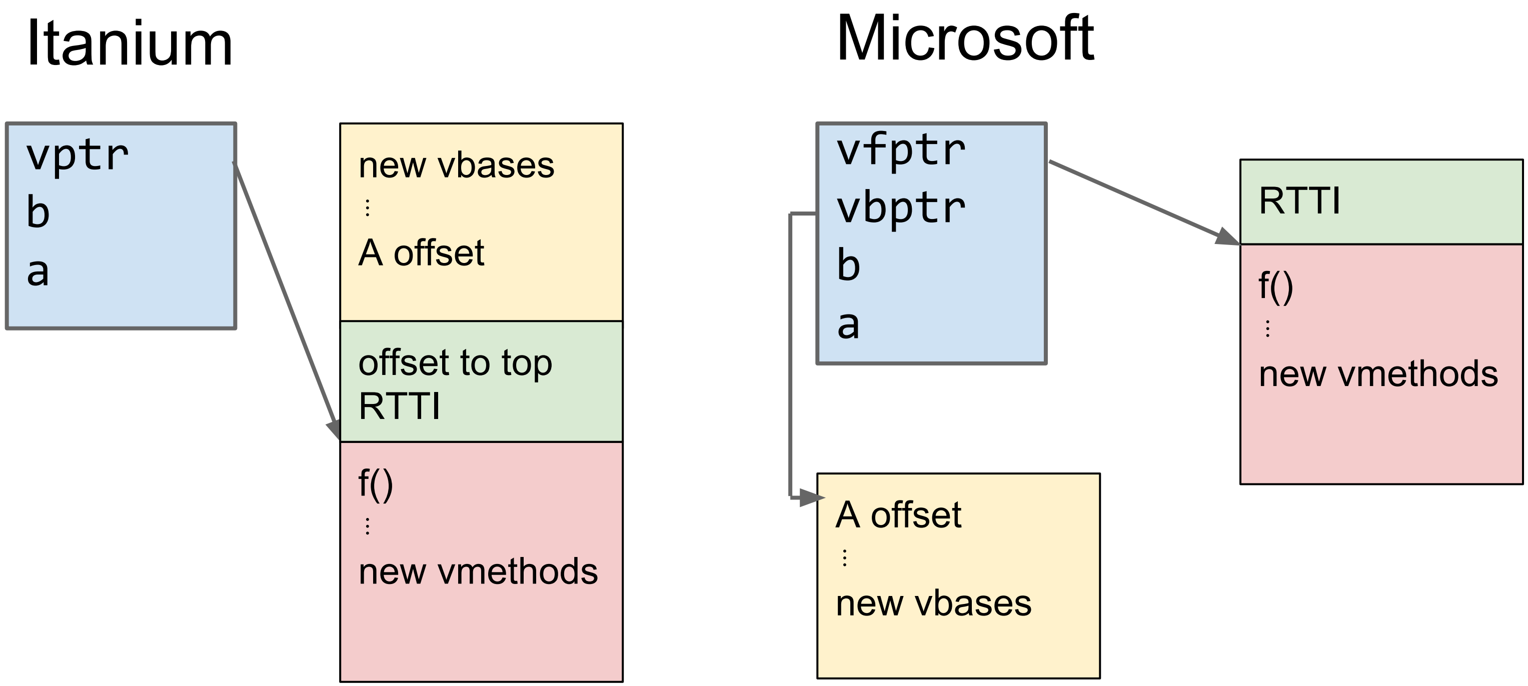
```
struct Foo { virtual void f() {} };
struct : Foo { void f() { puts("a"); } } a;
struct : Foo { void f() { puts("b"); } } b;
void call_foo(Foo *a) { a->f(); }
int main() {
  call_foo(&a);
  call_foo(&b);
}
```


WAT.

Virtual function and base tables

MSVC splits vtables into vftables and vbtables

```
struct A { int a; };
struct B : virtual A { virtual void f(); int b; };
```



Basic record layout

High-level rules are the same:

```
struct A { int a; };
struct B : virtual A { int b; };
struct C : virtual A { int c; };
struct D : B, C { int d; };
```

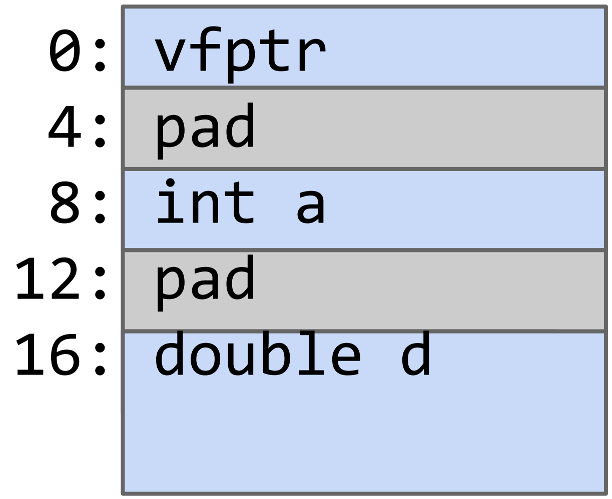
Gives D the layout:

B:	0	(B vbtable pointer)
	4	int b
C:	8	(C vbtable pointer)
	12	int c
D:	16	int d
A:	20	int a

Interesting alignment rules

```
struct A {
  virtual void f();
  int a;
  double d;
};

// Intuitively matches:
struct A {
  void *vfptr;
  struct _A_fields {
    int a;
    double d;
  };
};
```

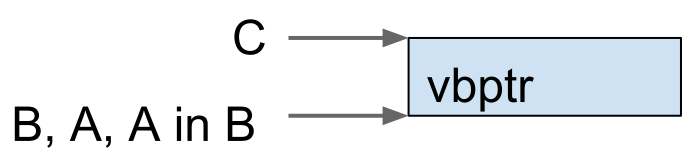


Again, presumably this is to make COM work for hand-rolled C inheritance

Zero-sized bases are interesting

- C++ says objects should not alias
- All bases are at offset 4:

```
struct A {  };
struct B : A { };
struct C : B, virtual A { };
sizeof(C) == 4
```



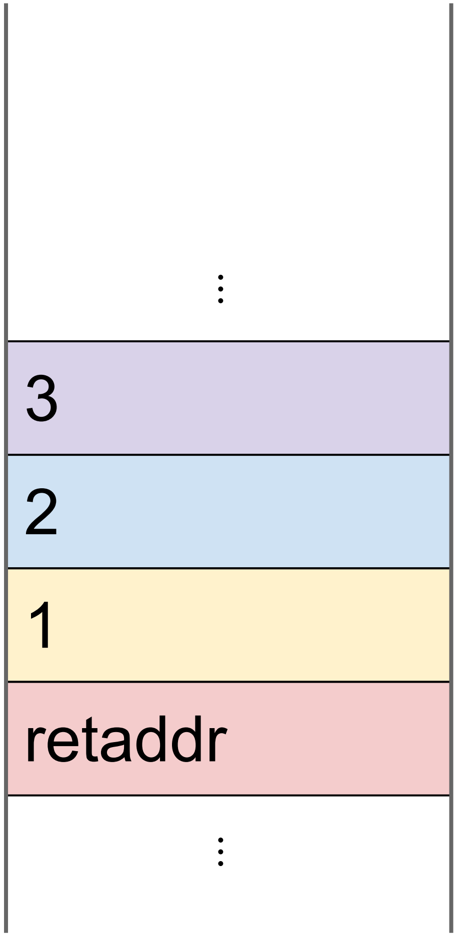

wat

Passing C++ objects by value

Pass by value in C

Corresponds to ‘byval’ in LLVM

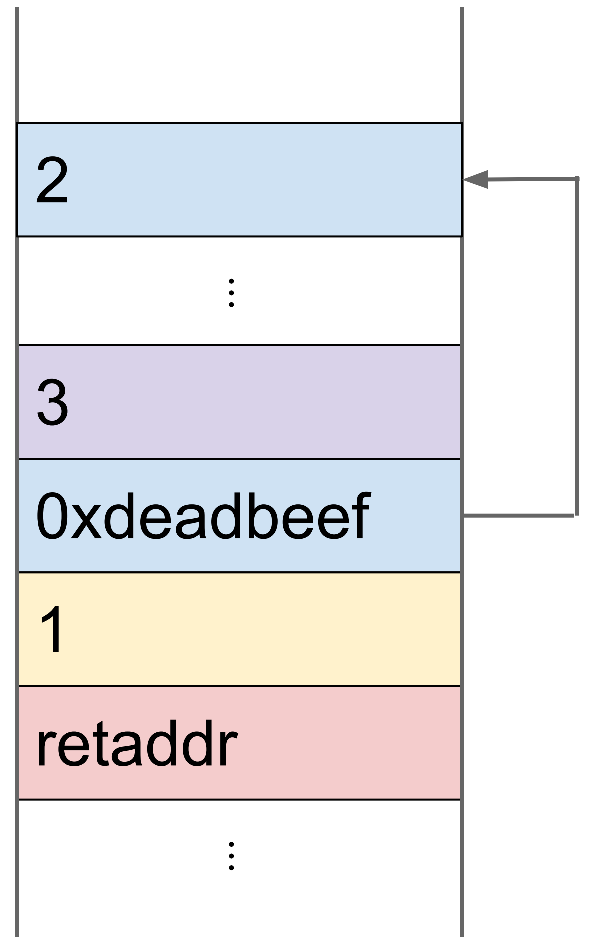
```
struct A {
  int a;
};
struct A a = {2}
foo(1, a, 3);
```



Pass by value in Itanium C++

Must call copy ctor

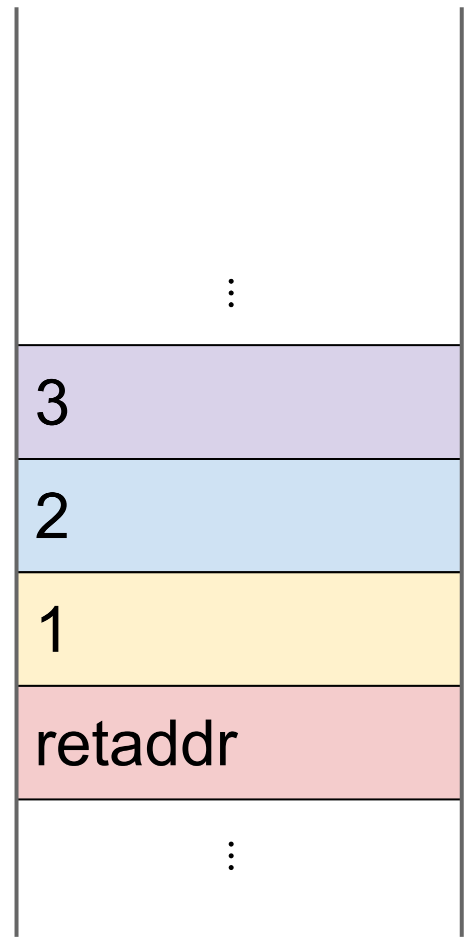
```
struct A {
  A(int a);
  A(const A &o);
  int a;
};
foo(1, A(2), 3);
```



Pass by value in Microsoft C++

- Constructed into arg slots
- Destroyed in callee

```
struct A {
  A(int a);
  A(const A &o);
  int a;
};
foo(1, A(2), 3);
```



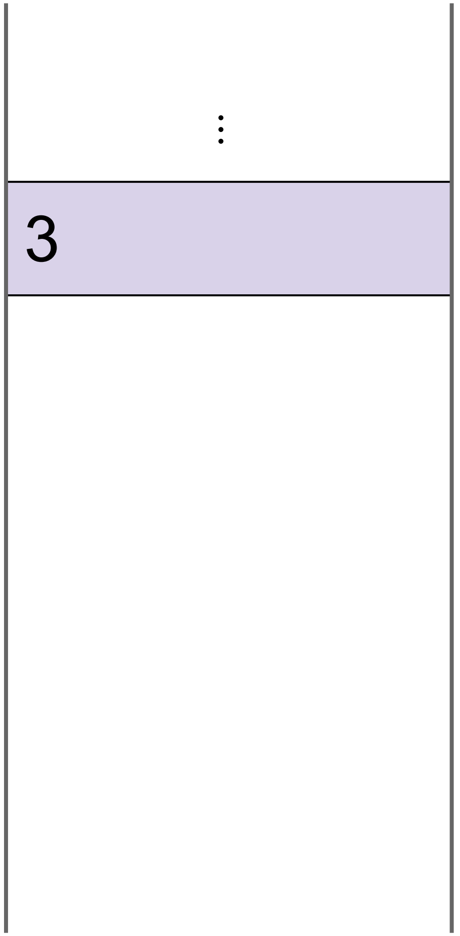
A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```

⋮

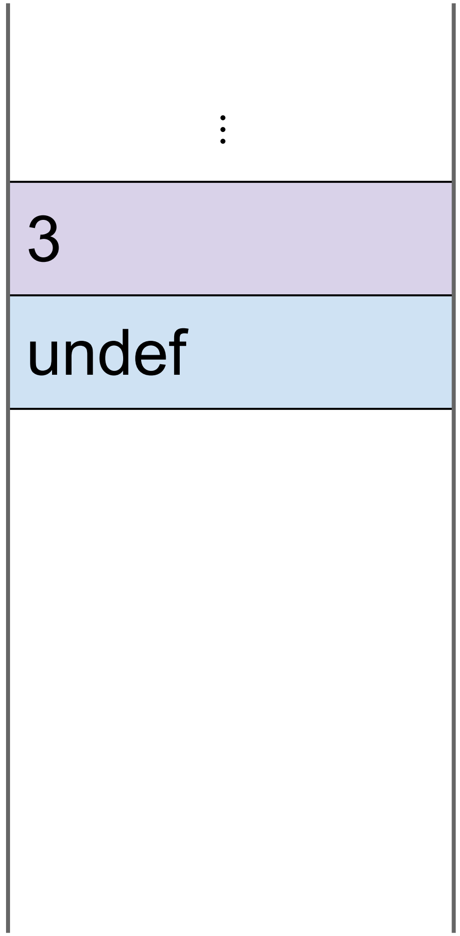
A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```



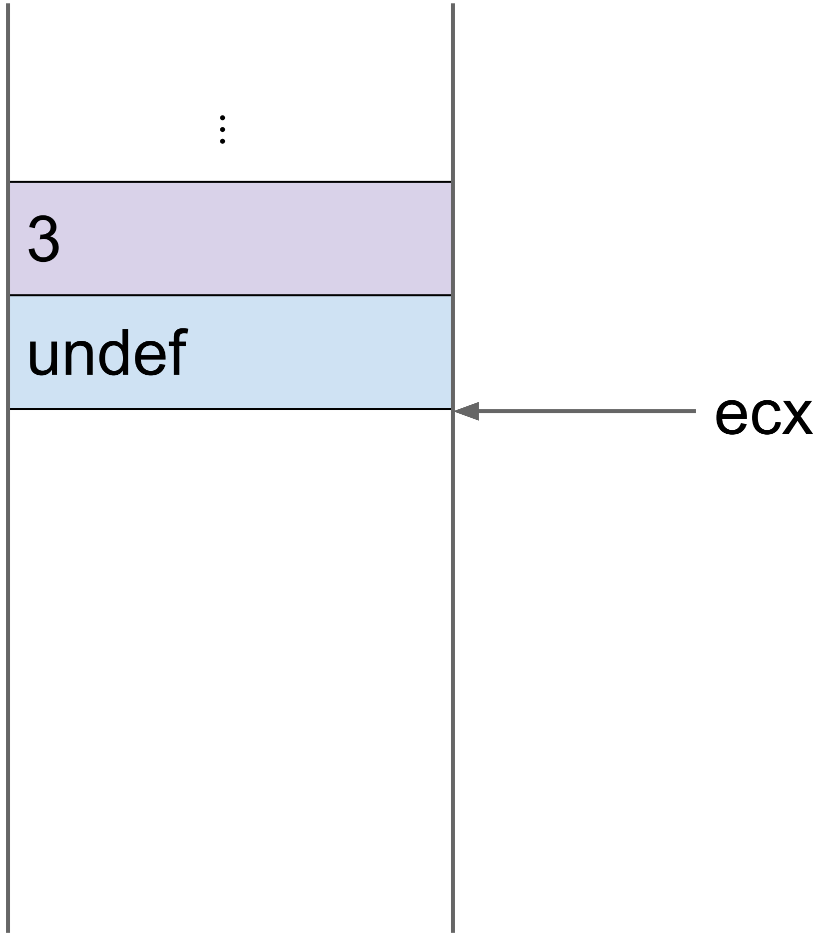
A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```



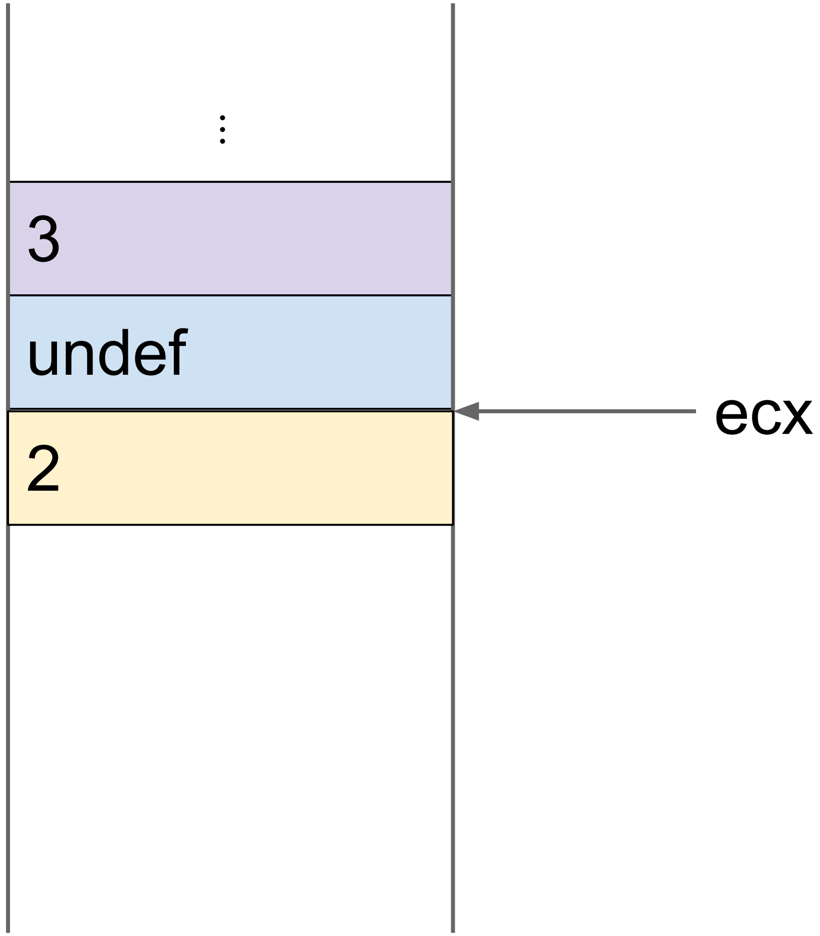
A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```



A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```

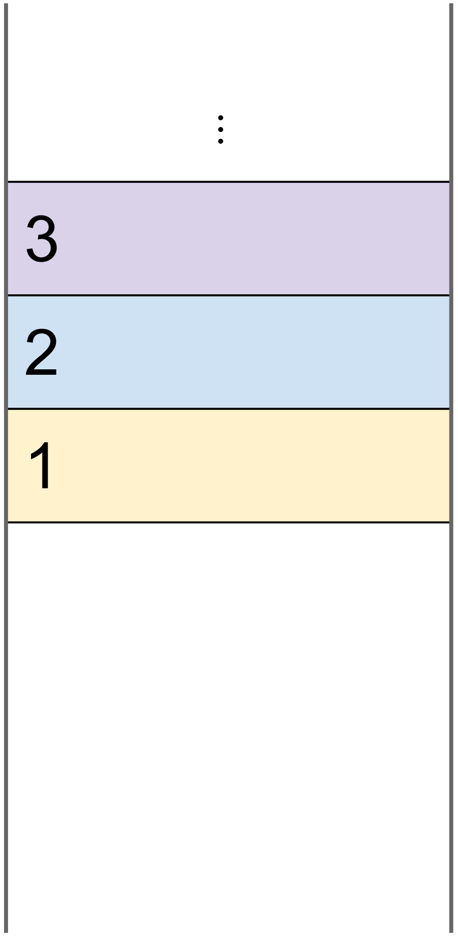


A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```

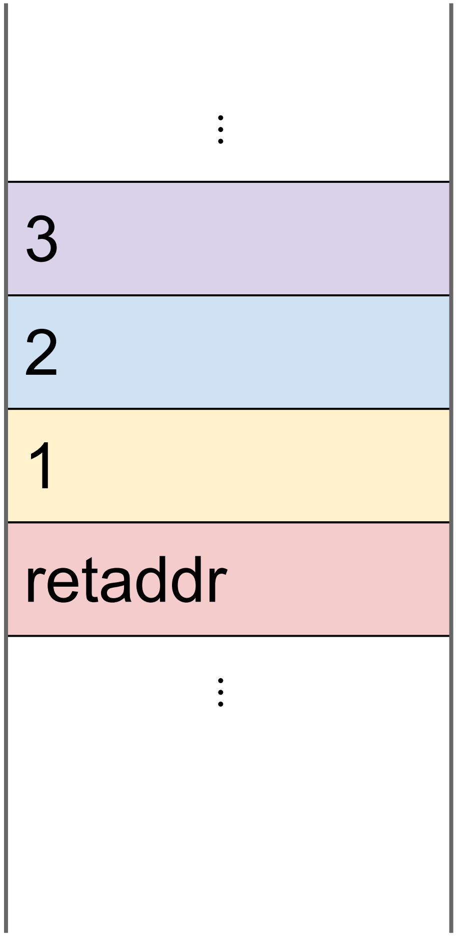
A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```

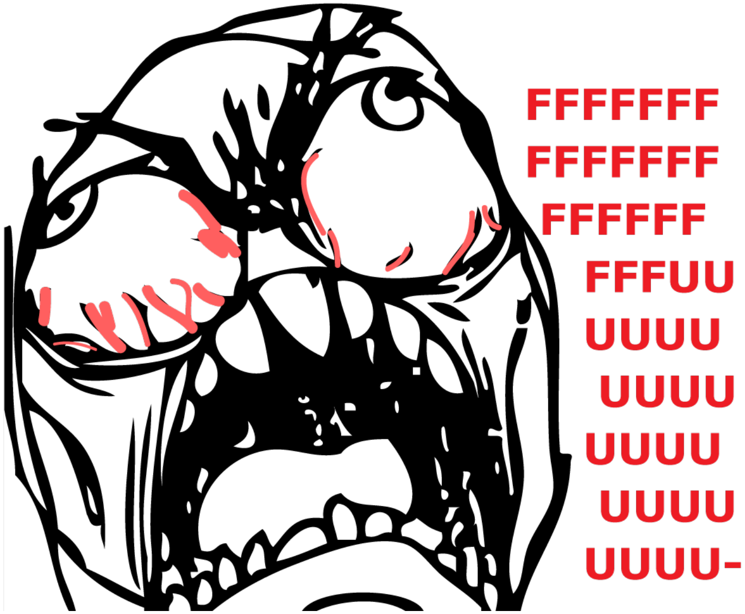


A hypothetical natural lowering

```
; foo(1, A(2), 3)
push 3
sub esp, 4
mov ecx, esp
push 2
call A_ctor
push 1
call foo
```



LLVM IR cannot represent this today!



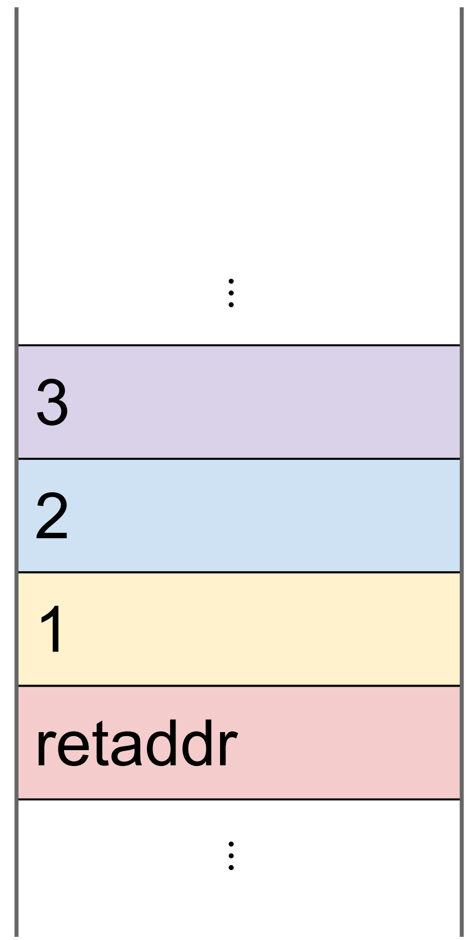
Pass by value in LLVM IR today

IR lowering today:

```
foo(1, A(2), 3);

call void @foo(
  i32 %1,
  %struct.A byval %2,
  i32 %3)
```

- byval implies a copy
- Where is the copy ctor?



How can we support this?

- Calls can be nested
  - foo(bar(A()), A())
  - Cannot reuse arg slot memory
  - Must adjust stack or copy
- Any call can throw exceptions
  - Even the copy ctor
  - Cannot tell LLVM how to copy
- Requirements
  - Need lifetime bounds respected by optimizers
  - Must be able to cleanup without calling
  - Allow an efficient future lowering (no frame pointer)

Proposal: inalloca

- The argument is passed… in the alloca
- An alloca used with inalloca takes the address of the outgoing argument

```
; Lowering for foo(A())
%b = call i8* @llvm.stacksave()
%a = alloca %struct.A
call void @ctor_A(%struct.A* %a)
call void @foo(%struct.A* inalloca %a)
call void @llvm.stackrestore(i8* %b)
```

Handles nested calls

```
; Lowering for foo(A(A()))
%b1 = call i8* @llvm.stacksave()
%a1 = alloca %struct.A
  %b2 = call i8* @llvm.stacksave()
  %a2 = alloca %struct.A
  call void @ctor_A(%struct.A* %a2)
  call void @ctor_A_A(%struct.A* %a1,
                      %struct.A* inalloca %a2)
  call void @llvm.stackrestore(i8* %b2)
call void @foo(%struct.A* inalloca %a1)
call void @llvm.stackrestore(i8* %b1)
```

Handles cleanup on unwind

```
; Lowering for foo(A())
%b = call i8* @llvm.stacksave()
%a = alloca %struct.A
invoke void @ctor_A(%struct.A* %a)
  to label %ft unwind label %lp

%ft: call void @foo(...)
     call void @llvm.stackrestore(i8* %b)

%lp: ; Destroy any temporaries
     call void @llvm.stackrestore(i8* %b)
```

Possible improvements

- Certain forms of save/restore can be optimized to constant adjustments
  - This shouldn’t require a frame pointer
  - Or, we could add an ‘afree’ instruction

Conclusion and questions

- Integrating Clang into Visual Studio
- Supporting the Visual C++ ABI in Clang
- Parsing Visual C++ extensions in Clang
- Visual C++ ABI corner cases
- Adding inalloca to LLVM for MSVC byval params