

Pointer Analysis for C/C++ with cclyzer



George Balatsouras <gbalats@di.uoa.gr>
University of Athens

cclyzer

Static Analysis Framework
for C/C++

- ❑ Analyzes C/C++ programs translated to LLVM Bitcode
- ❑ Declarative framework
 - ❑ Analyzes written in Datalog rules
- ❑ Uses the LogicBlox Datalog engine
 - ❑ Relations stored as database tables
 - ❑ Output relations computed using Datalog's least fixpoint model of the program

LLVM IR - Basic Instructions

- ❑ Stack allocations (1) `p = alloca [type]`
- ❑ Heap allocations (2) `p = malloc nbytes`
- ❑ Load from address (3) `v = load p`
- ❑ Store to address (4) `store v, p`
- ❑ Address-of-field (5) `poffset = &(p->f)`
- ❑ Address-of-array-index (6) `poffset = &(p[i])`
- ❑ Function call (7) `v = call fn (arg1, arg2, ...)`
- ❑ No-op cast (8) `v = bitcast p to [type]`

LLVM Bitcode vs Java Bytecode

I. Addresses of Fields

LLVM Bitcode

- ❑ As in C, an instance field can have its *address taken*
- ❑ ... and then *loaded* elsewhere.
- ❑ By elsewhere, we mean even in a different function
- ❑ Expression ‘p->f’ in fact translates to:
$$p_{\text{offset}} = \&(p \rightarrow f)$$
$$v = \text{load } p_{\text{offset}}$$

Java Bytecode*

- ❑ Impossible in Java
- ❑ May only allocate objects and then load from or store to some field
- ❑ Load/store instructions hence are ternary, containing an extra *field operand*

LLVM Bitcode vs Java Bytecode

II. Virtual registers

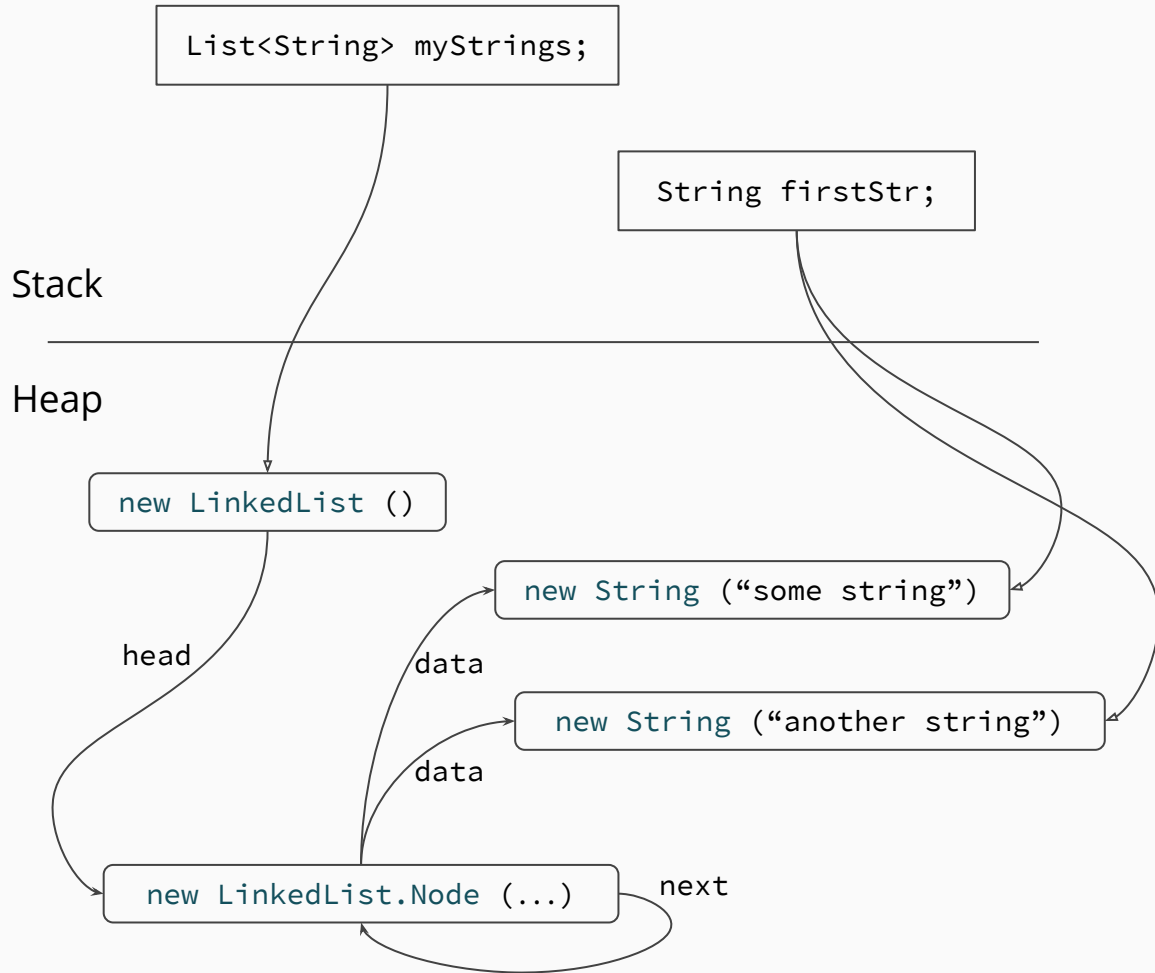
LLVM Bitcode

- ❑ All source-level variables become pointers ... unless optimized away
- ❑ E.g., `int p = 3;` becomes:
`%p = alloca i32`
`store i32 3, i32* %p`
- ❑ `&p` becomes just `%p`
- ❑ Subsequent assignments to `p` become store instructions to `%p`
- ❑ Additional temporary variables are introduced for intermediate expressions (e.g., `%1`, `%2`)
- ❑ Both `%p` and `%1`, `%2` are *virtual registers*.
- ❑ At register allocation:
 - some will be replaced by *physical registers*
 - some will be *spilled*.

Pointer Analysis on LLVM bitcode

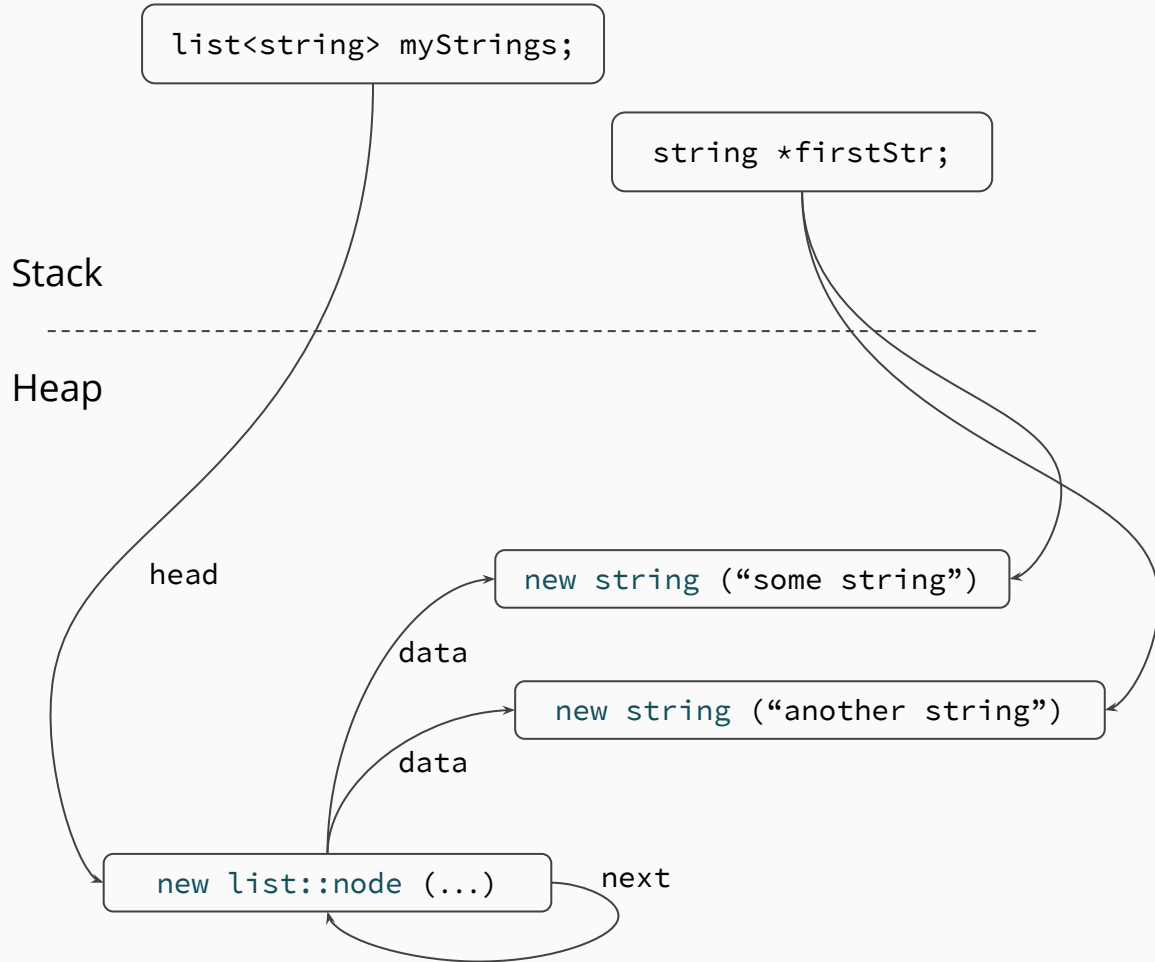
Java Memory Abstraction

- ❑ Clear distinction
 - ❑ variables reside on stack
 - ❑ allocated objects reside on heap
- ❑ Pointer analysis
 - ❑ variables *point-to* heap objects
 - ❑ heap objects *point-to* other heap objects through some field



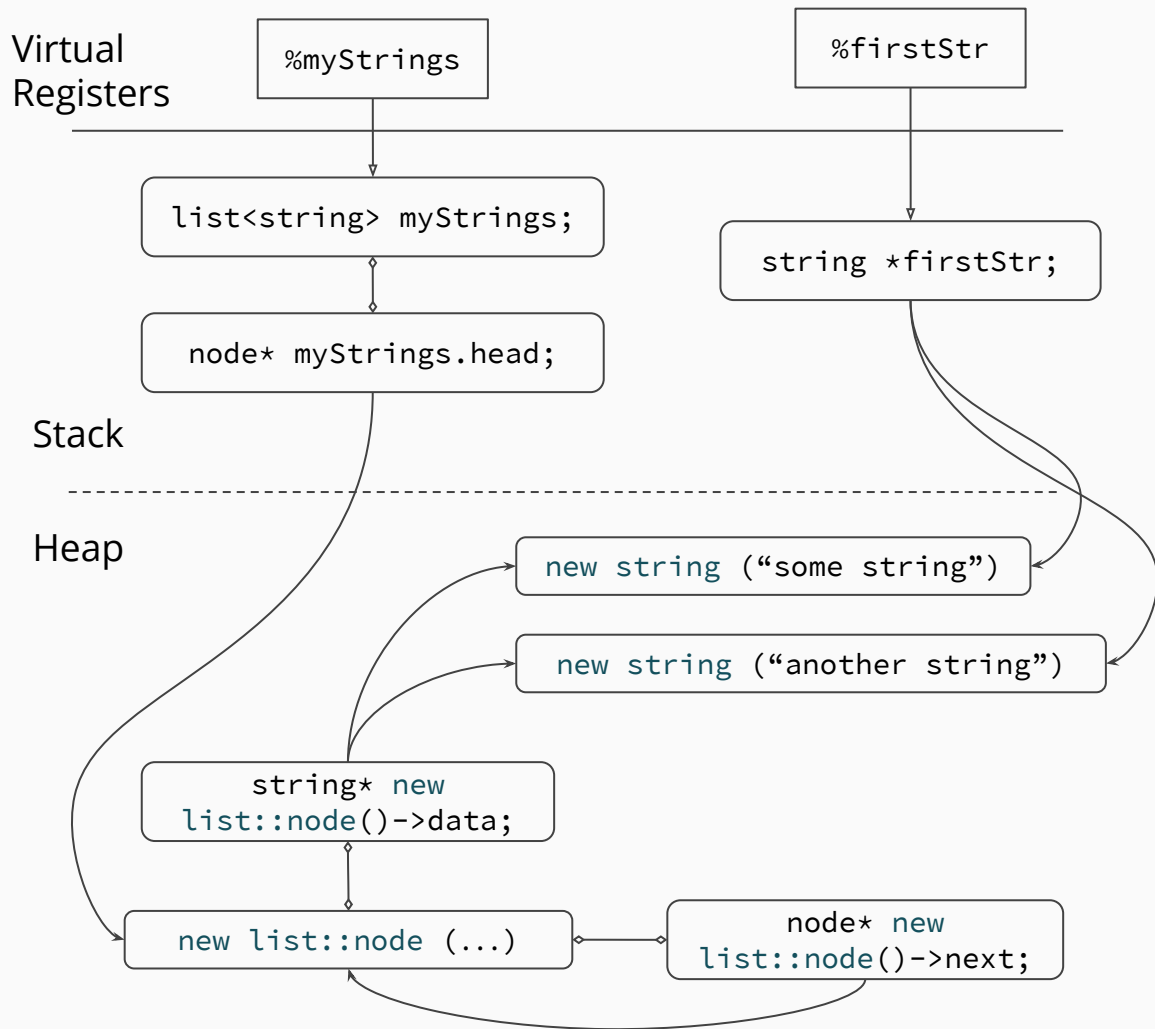
C/C++ Memory Abstraction

- ❑ Objects may be allocated:
 1. either on the heap
 2. or on the stack
- ❑ Pointer analysis
 - ❑ Dereference edges from abstract object to another abstract object
- ❑ What about field edges?
 - ❑ Objects contain other objects; unlike Java
 - ❑ Recall: we can take the address of a field



Our LLVM Memory Abstraction

- ❑ Decouple a variable from its stack allocation
- ❑ From now on, by *variable*, we mean virtual register
- ❑ Pointer analysis
 - ❑ Variables point-to (abstract) objects
 - ❑ Objects, when *dereferenced* point-to other objects
 - ❑ Fields of objects are objects themselves



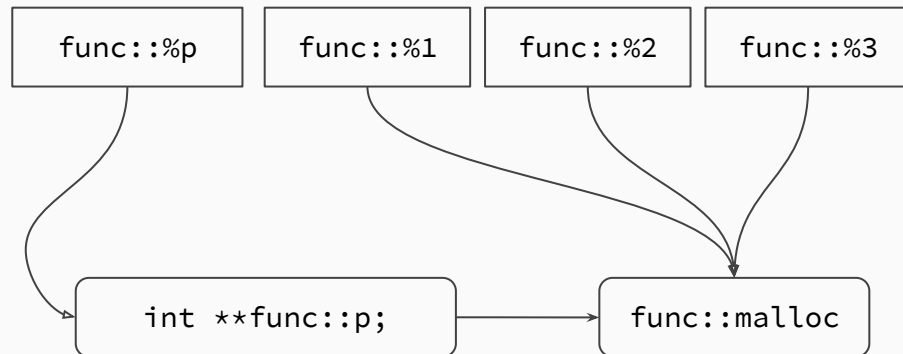
Analyzing C/C++ code with cclyzer

<https://github.com/plast-lab/cclyzer>

Simple Example: Computing Points-to

LLVM Bitcode

```
int func() {  
    int*** %p = alloca [int **];  
    void* %1 = call @malloc(8);  
    int** %2 = bitcast %1 to int**;  
  
    store %2, %p;  
    int** %3 = load %p;  
  
    ret %3;  
}
```



Revisiting points-to

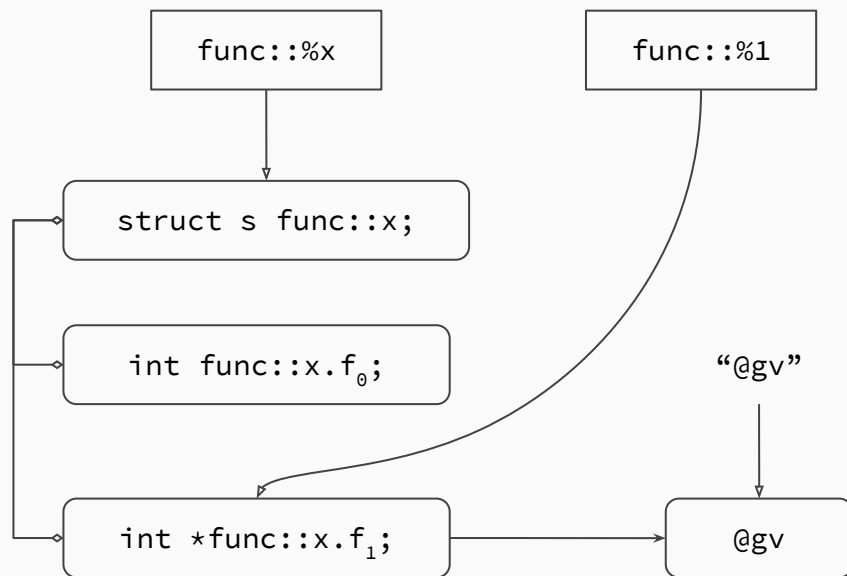
Field Sensitivity

LLVM Bitcode

```
int* @gv = global int 0;

%struct.s = type { int, int* }

void func() {
    %x = alloca [%struct.s];
    %1 = getelementptr %x, 0, 1; // &(x.
    f1)
    store @gv, %1;
}
```



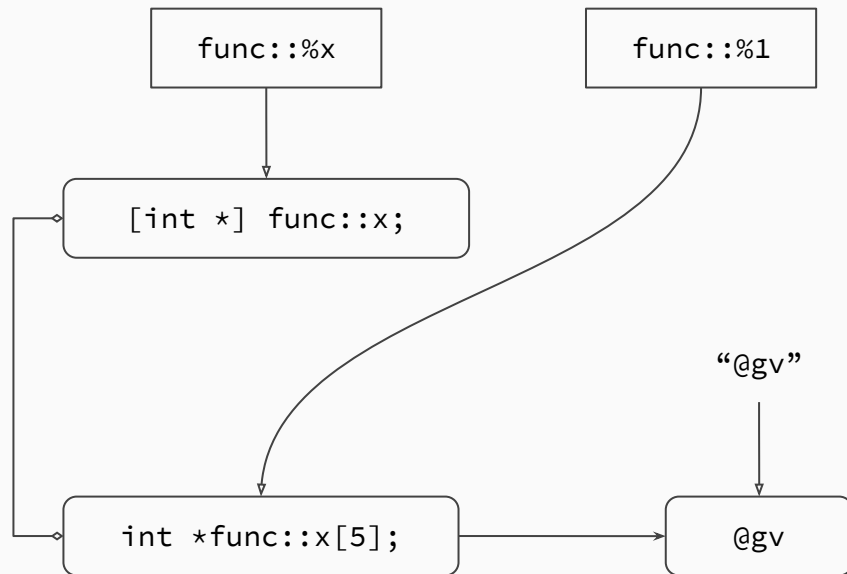
Revisiting points-to

Array Sensitivity

LLVM Bitcode

```
int* @gv = global int 0;
```

```
void func() {  
    %x = alloca [100 x int*];  
    %1 = getelementptr %x, 0, 5; // &(x  
[5])  
    store @gv, %1;  
}
```



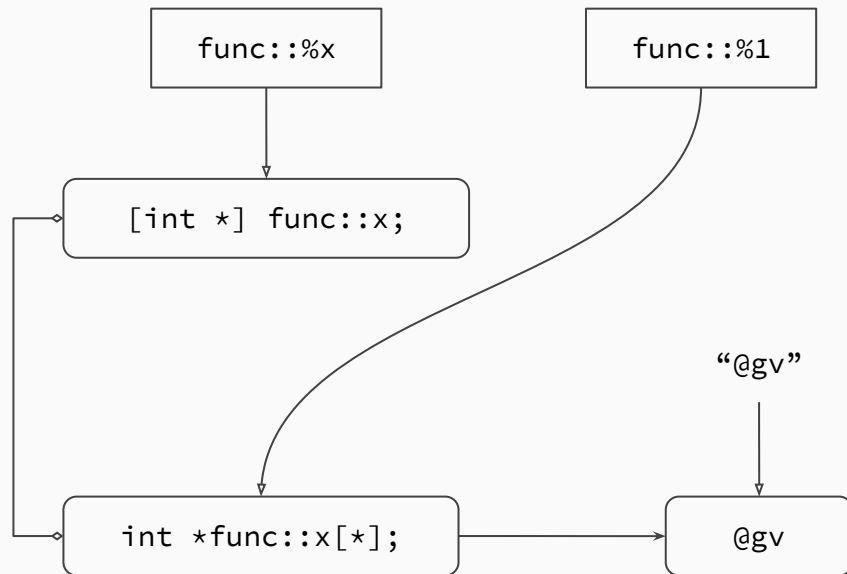
Revisiting points-to

Array Sensitivity

LLVM Bitcode

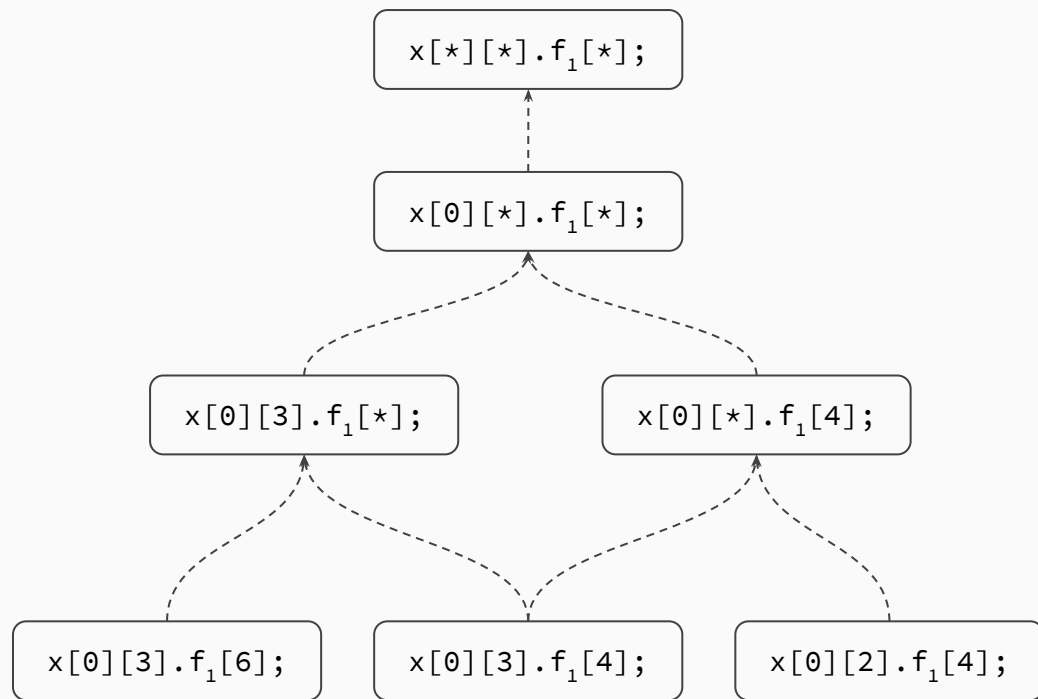
```
int* @gv = global int 0;
```

```
void func() {  
    %x = alloca [100 x int*];  
    %i = ...  
    %1 = getelementptr %x, 0, %i; // &(x  
[i])  
    store @gv, %1;  
}
```



Array Sensitivity

- Define partial order
- (n_1, n_2) when n_1 can be turned to n_2 by substituting constant indices with '*'
- points-to set of a node is a *superset* of the points-to set of its parent
- At *load* instructions, merge with the points-to sets of *all* children nodes



Strong Type Information

Type back-propagation

- ❑ Analysis only creates **typed** abstract subobjects
 - ❑ Must determine the type of their base object
 - ❑ What about objects of unknown type (e.g., `malloc()`)?
- ❑ Type back-propagation:
 - ❑ track *cast instructions* (resp. types) that an object of unknown type flows to
 - ❑ create a *new* abstract object per possible type, for a single allocation site
 - ❑ In turn, more abstract subobjects can now be created

Analyzing C++ code

compiled to LLVM IR

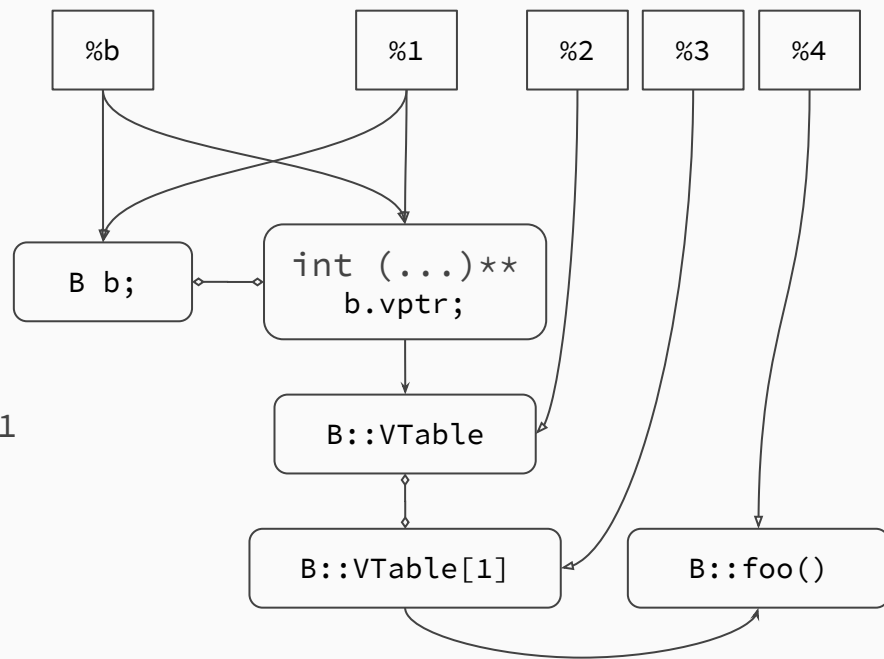
Challenges

- ❑ LLVM bitcode is a representation that is well-suited for C code
- ❑ Too low-level for C++
- ❑ C++ features like classes, v-tables, references, and so on are translated to low-level constructs

Dynamic Dispatch Example

LLVM Bitcode

```
%class.B = type { int (...)**, ...}  
  
void func() {  
  %b = alloca [%class.B];  
  ...  
  %1 = bitcast %b to int (%class.B*)***  
  %2 = load int (%class.B*)** %1  
  %3 = getelementptr int (%class.B*)** %2, 1  
  %4 = load int (%class.B*)* %3  
  call int %4 (%class.B* %b)  
}
```



Upcoming SAS '16 paper:

**Structure-Sensitive Points-To
Analysis for C and C++**

George Balatsouras and Yannis Smaragdakis