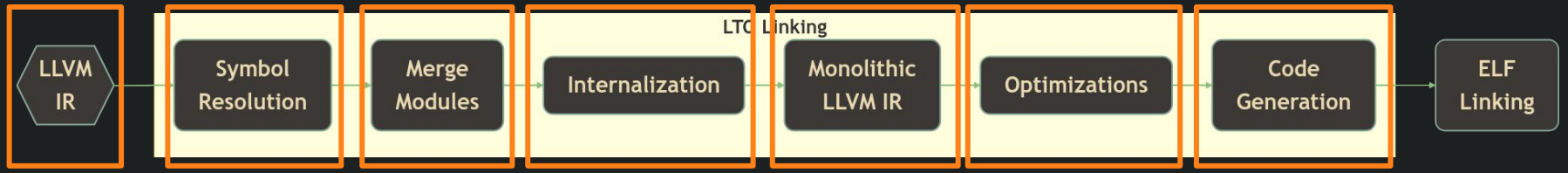
LT-Uh-Oh: Adventures trying to LTO libc

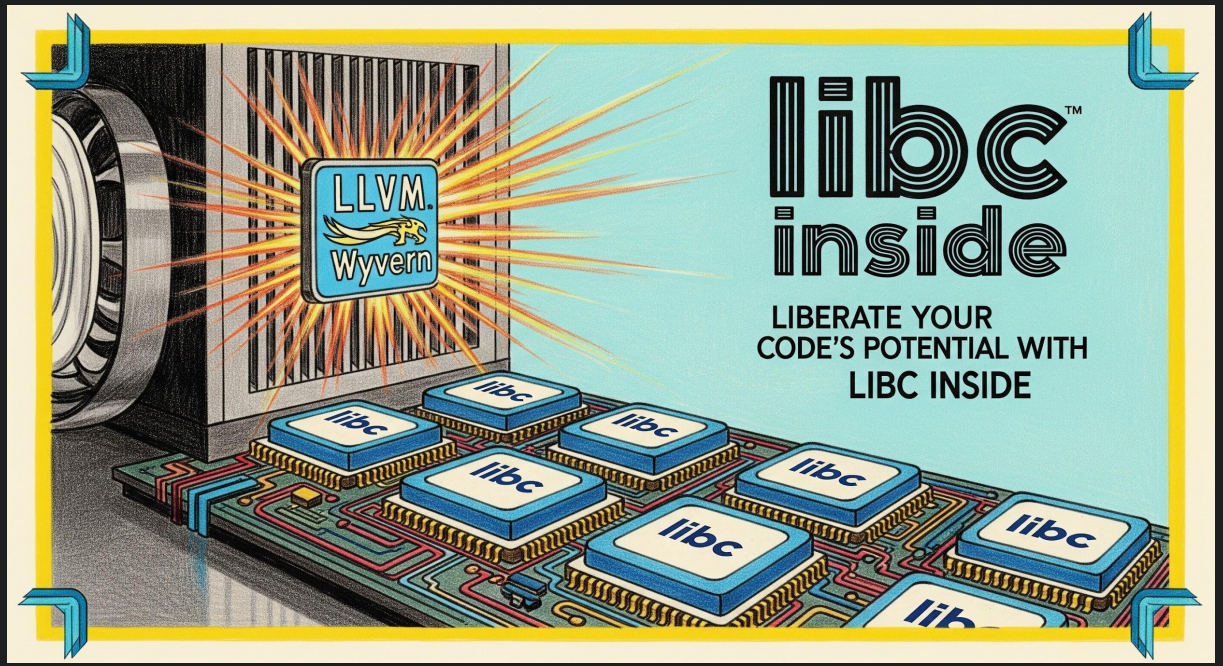
Google
Paul Kirth
Dan Thornburgh

A (brief) Overview of Full LTO



Why LTO libc: moar compiler == moar better

- Inlining
- Optimizing interfaces
- Removing more code
- LTO all the things!



What happens when you try to LTO w/ libc?

```
$ clang -flto app.o libc.a
```

What happens when you try to LTO w/ libc?

```
$ clang -flto app.o libc.a
ld.lld: error: undefined hidden symbol: bcmp
>>> referenced by char_traits.h:125
(../../prebuilt/third_party/clang/linux-x64/bin/../include/c++/v1/__string/char_traits.h:125)
>>>               linux_arm-lto-shared/lib.unstripped/libld-startup.so.lto.o:(StartLd)
```

What happens when you try to LTO w/ libc?

```
$ clang -flto app.o libc.a
ld.lld: error: undefined hidden symbol: bcmp
>>> referenced by char_traits.h:125
(../../prebuilt/third_party/clang/linux-x64/bin/../include/c++/v1/__string/char_traits.h:125)
>>>               linux_arm-lto-shared/lib.unstripped/libld-startup.so.lto.o:(StartLd)
```

Undefined hidden symbol? But bcmp is in the source!

Lets see what's in the IR …

What happens when you try to LTO w/ libc?

```
$ clang -flto app.o libc.a
ld.lld: error: undefined hidden symbol: bcmp
>>> referenced by char_traits.h:125
(../../prebuilt/third_party/clang/linux-x64/bin/../include/c++/v1/__string/char_traits.h:125)
>>>               linux_arm-lto-shared/lib.unstripped/libld-startup.so.lto.o:(StartLd)
```

Undefined hidden symbol? But bcmp is in the source!

Lets see what's in the IR …

```
$ clang -flto app.o libc.a -Wl,--save-temps
$ llvm-opt app.1.preopt.bc -S -o -
```

What happens when you try to LTO w/ libc?

```
define i1 @foo(ptr %0, [2 x i32] %1) {
  %size = extractvalue [2 x i32] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i32 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i32) {
  ; implementation ...
}

define i32 @bcmp(ptr %0, ptr %1, i32 %2) {
  ; implementation ...
}
```

Simplified Example: Initial Module

```
define i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)

define i32 @bcmp(ptr %0, ptr %1, i64 %2) {
  ret i32 0
}
```

Simplified Example: Post GlobalDCE

```
define internal i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)
```

Simplified Example: Post Instcombine

```
define internal i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %bcmp = call i32 @bcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %bcmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)

; Function Attrs: nocallback nofree nounwind willreturn memory(argmem: read)
declare i32 @bcmp(ptr captures(none), ptr captures(none), i64) #0
```

This is why we have an undefined symbol!

Simplified Example: Post Instcombine

```
define dso_local
  %size = extrac
  %bcmp = call i
  %eq = icmp eq
  ret i1 %eq
}

declare i32 @mem

; Function Attrs
declare i32 @bcm
```

Summary of what happened

All the libc APIs get marked `internal` (except RuntimeLibcalls).

GlobalDCE sees `bcmp` is unused and removes it.

SimplifyLibcall “optimizes” `memcmp` to `bcmp` **after GlobalDCE removes `bcmp`**!

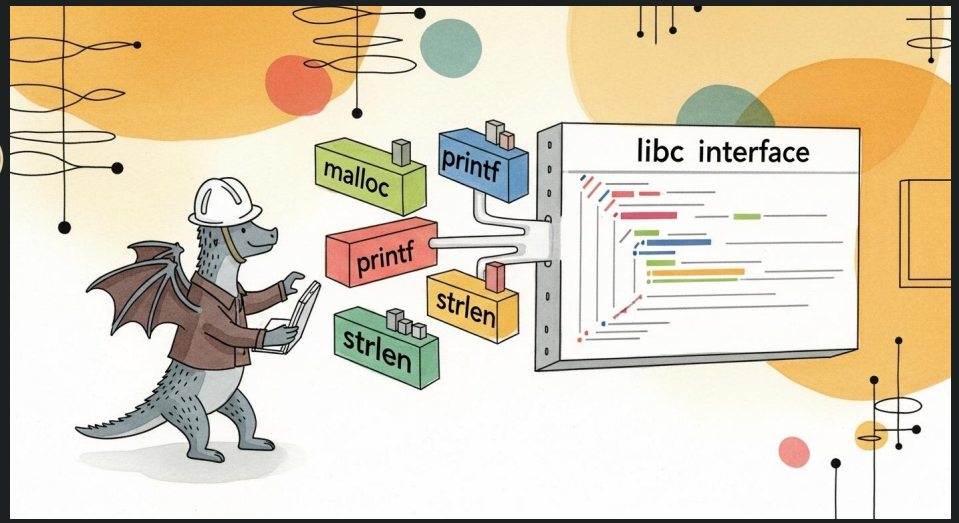
Linking **breaks** because `bcmp` is no longer provided, since it was **deleted**.

What's different about LTO w/ libc

Clang assumes things about libc based on **static** or **dynamic linking**

libc: is available as an **abstract interface** that it can **always** introduce calls into

- High-level transforms:
  - Well known code patterns (`memcpy`)
  - Replacing existing calls (`memcmp`→`bcmp`)
  - Controlled w/ `-fno-builtins`
- Low-level transforms:
  - Some public libc APIs are also **libcalls**
  - No way to opt out



Solution: Make them part of Runtime Libcalls!

RuntimeLibcalls are treated specially by the optimization pipeline.

- Can’t be removed (they provide a definition)
- Always extracted from an archive
- Why not make all the APIs the compiler uses part of that set?

Builtins vs. No-Builtins “world”

We run into this problem because libc is treated as an abstract interface.

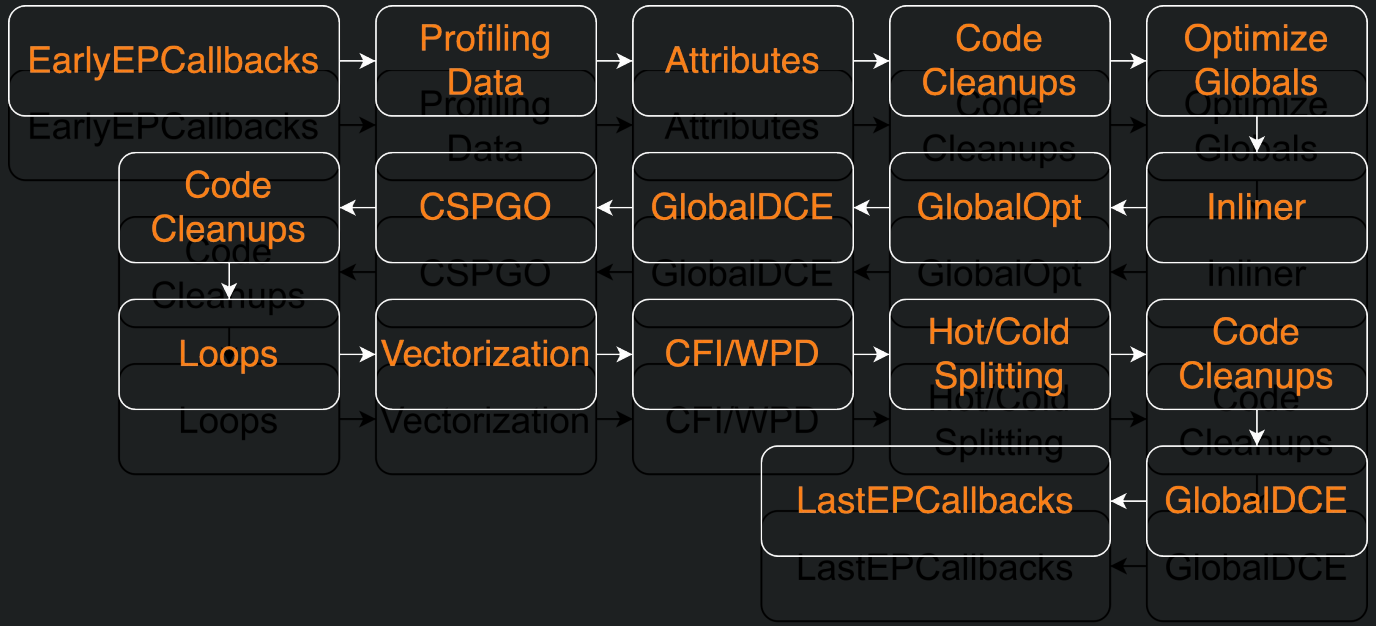
LLVM already has a mechanism to avoid this problem: `no-builtins`

We want to conceptually partition the post-link optimization pipeline:

- Initial section allows certain builtins (e.g. “builtins world”).
- After that all optimization, consider the “soup” of IR (“no-builtins world”)

Encoding the “cut” in IR

Slap “no-builtins” on all the functions in the module after the cut point



No-Builtins World

Just mark every functions `no-builtins` after the cut.

```
struct NoBuiltinsPass : public PassInfoMixin<NoBuiltinsPass> {
    PreservedAnalyses run(Module &M, ModuleAnalysisManager &MAM) {
        for (Function &F : M) {
            if (!F.isDeclaration()) {
                F.addFnAttribute("no-builtins");
            }
        }
        return PreservedAnalyses::none();
    }
};
```

It Works!

In no-builtins world, `bcmp` can be DCEd, but it's illegal to rewrite `memcmp` → `bcmp`

```
define dso_local i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)
```

Never replaced during GlobalOpt!

Oh No … It Works

```
define float @foo(float %x) {
  %call = tail call nnan ninf float @__sqrtf_finite(float %x) readnone
  ret float %call
}

declare float @__sqrtf_finite(float) readnone
```

Old behavior

```
foo:
        .cfi_startproc
        sqrtss  %xmm0, %xmm0
        retq
```

New behavior

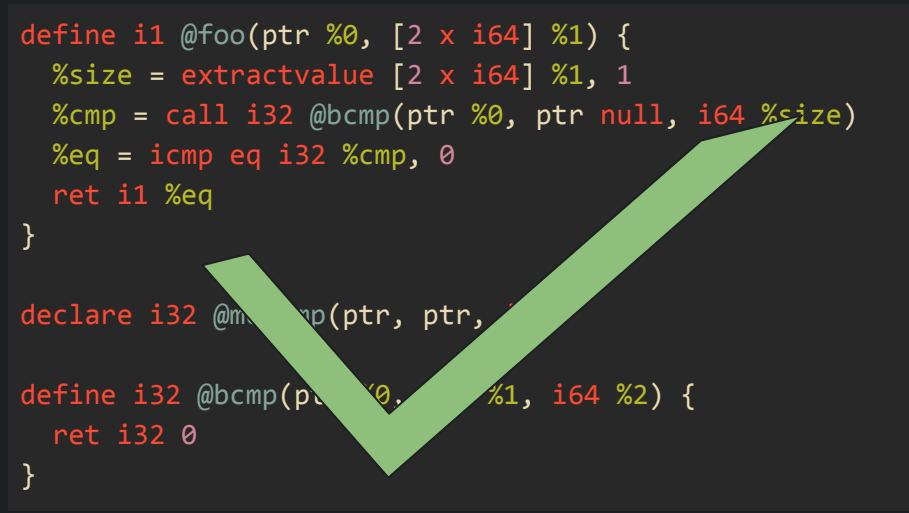
```
foo:
        .cfi_startproc
        jmp     __sqrtf_finite@PLT
```

`no-builtins` prevents using more efficient instructions over calls

Builtins World

Can add references to linked-in libc IR (builtin)

```
define i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)

define i32 @bcmp(ptr %0, ptr %1, i64 %2) {
  ret i32 0
}
```



Builtins World

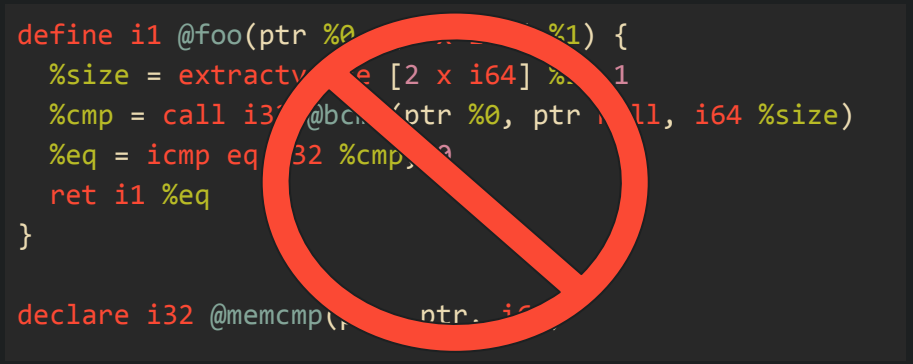
Cannot DCE or change semantics of linked-in libc IR

```
define i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)

define i32 @bcmp(ptr %0, ptr %1, i64 %2) {
  ret i32 0
}
```

Builtins World

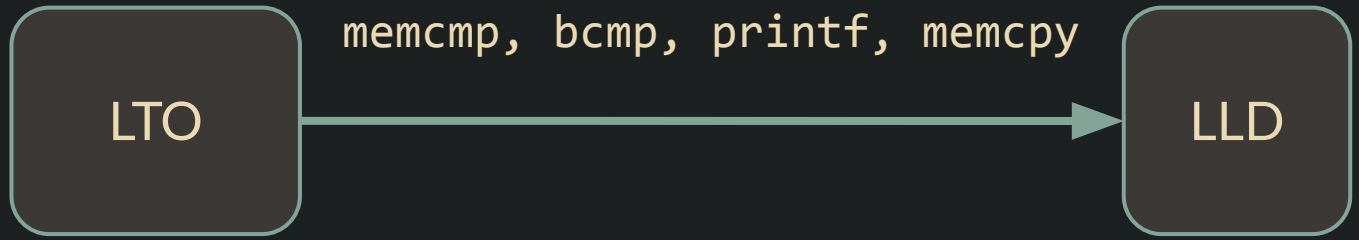
Cannot add references to not-linked-in libc IR (no-builtin)

```
define i1 @foo(ptr %0, [2 x i64] %1) {
  %size = extractvalue [2 x i64] %1, 1
  %cmp = call i32 @memcmp(ptr %0, ptr null, i64 %size)
  %eq = icmp eq i32 %cmp, 0
  ret i1 %eq
}

declare i32 @memcmp(ptr, ptr, i64)
```

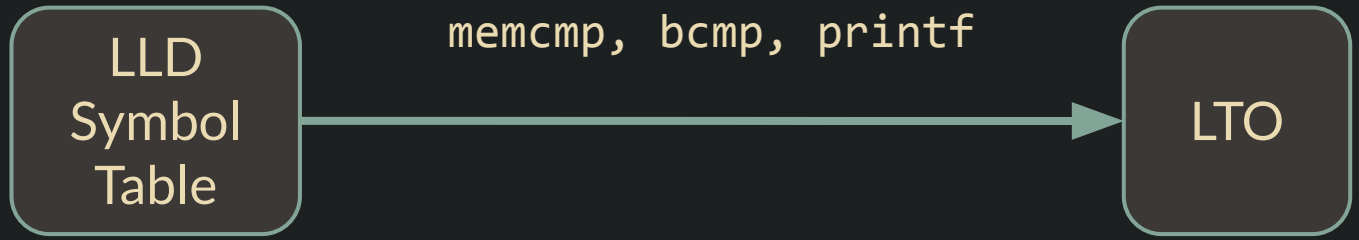


Builtins World: Mechanics

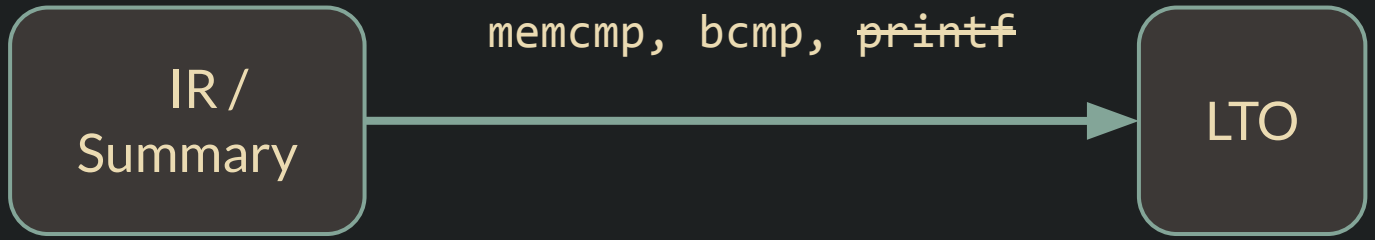
What are all the LibFuncs?



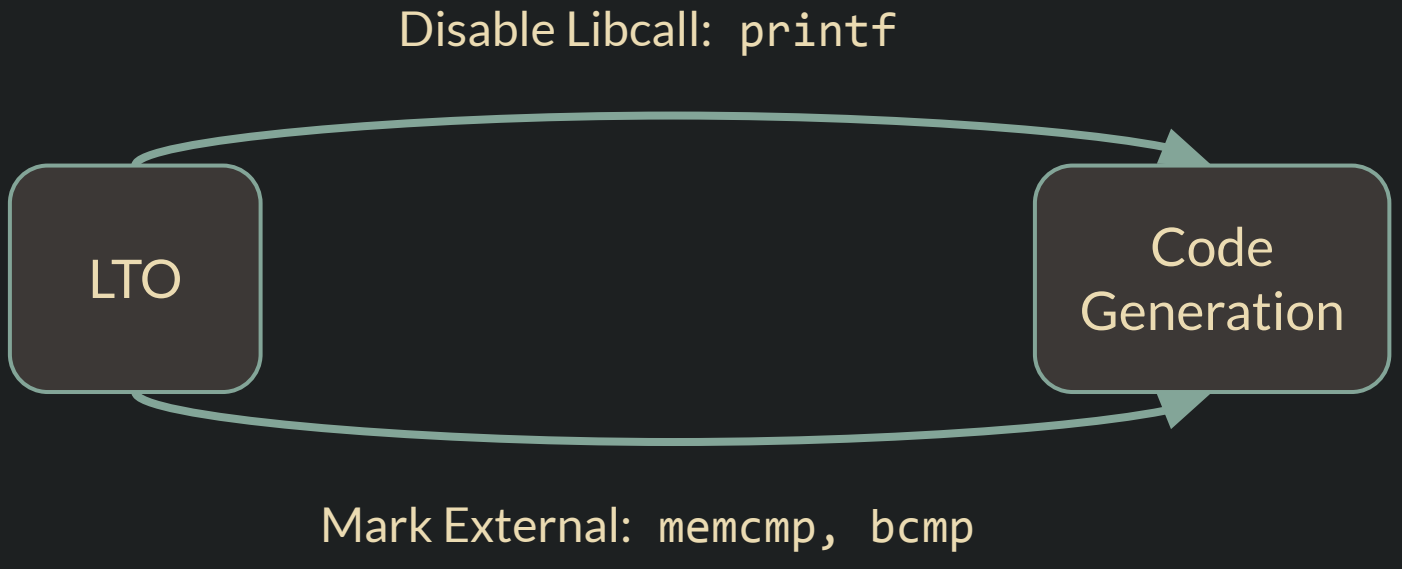
Which LibFuncs are in bitcode?



Which bitcode LibFuncs are available?



Adjust Code Generation



Where to go from here?

1. More Eval
2. More discussion
3. New attributes for libc?
4. Changes to the pass pipeline?
5. What about compiler builtins (e.g. compiler-rt)?
6. What *should* the semantics of LTO be?
7. Stuff *you* think of

Questions?

Discourse RFC:
https://discourse.llvm.org/t/rfc-addressing-deficiencies-in-llvm-s-lto-implementation/84999