Introducing ClangIR

High-Level IR for the C/C++ Family of Languages

Bruno Cardoso Lopes

Meta

Background

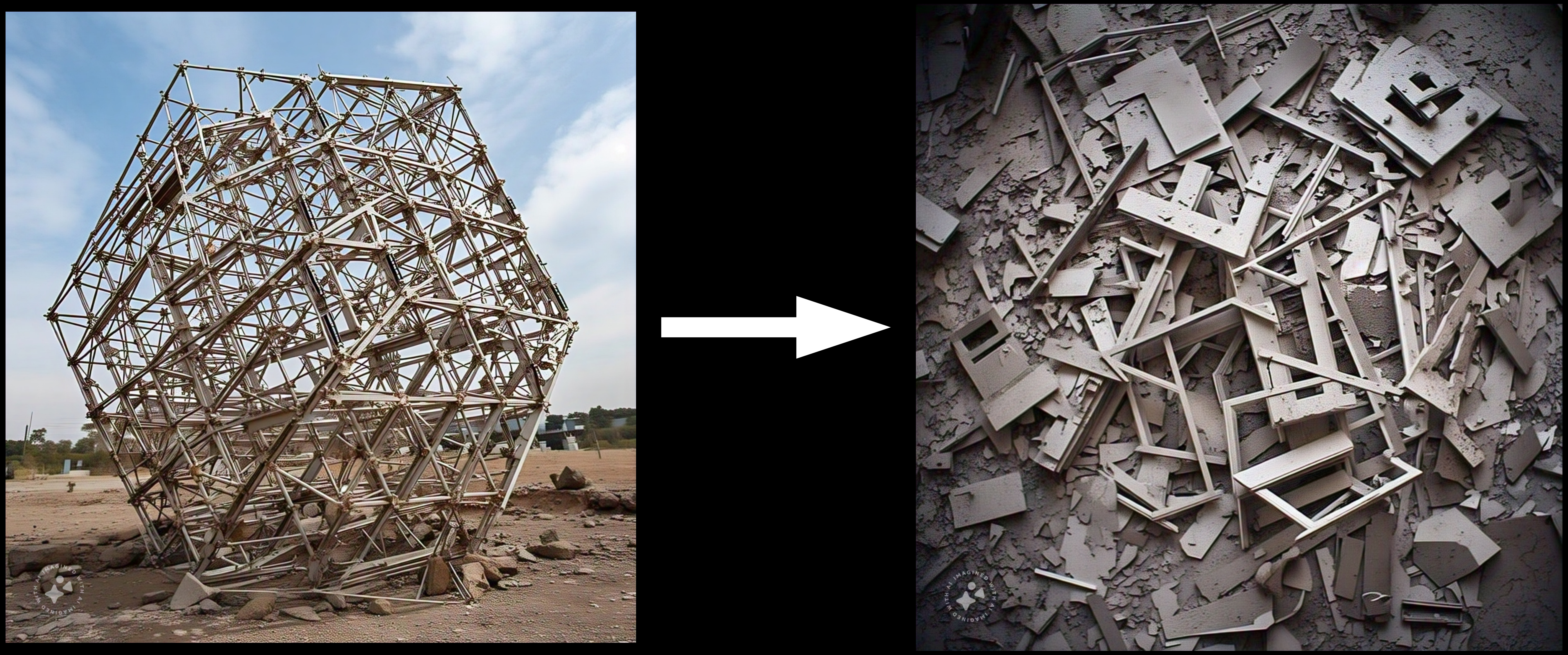
Compilation pipeline

- Multiple representations from source to machine code
- Each translation level requires **specific** information

Compilation pipeline

Progressive lowering

- Lowering: loss of information, each level is better at something



Compilation pipeline

Premature lowering

- May preclude language specific analysis & optimizations
- Reconstruction can be hard, expensive and brittle

Clang

Compiler C/C++ family of languages

- C++ is hard: more opt and analysis require richer IR
- Pipeline: C++ → AST → LLVM IR → […] → assembly
  - AST too high level
  - LLVM IR too low level (e.g. opaque ptrs)

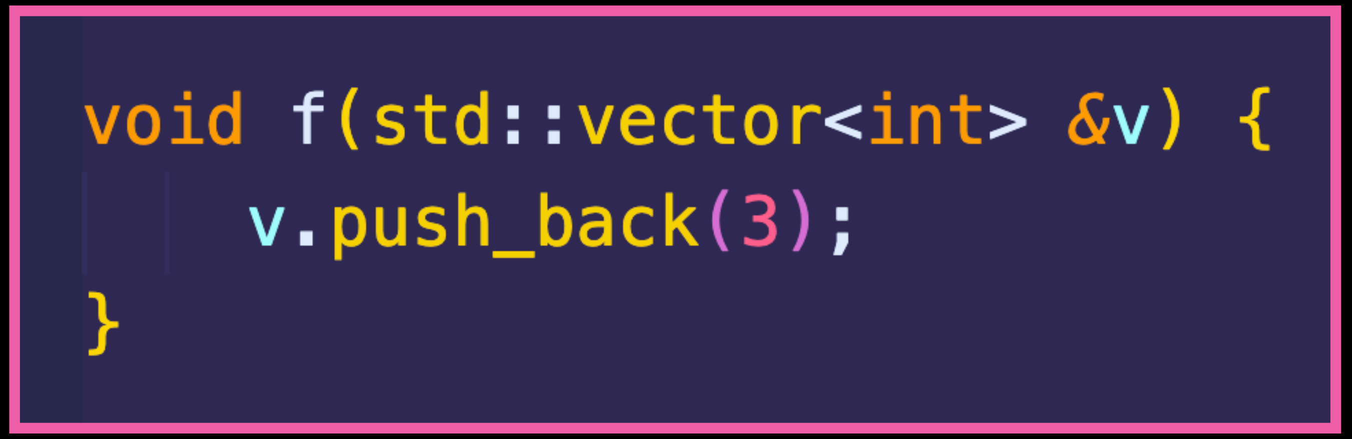
Clang

Why we need a new IR?

- Enable more static analysis and unlock optimization opportunities
- Success stories of high-level IRs
- Flang, Mojo, Rust, Swift, Open64’s WHIRL

Clang

Reconstruction is hard

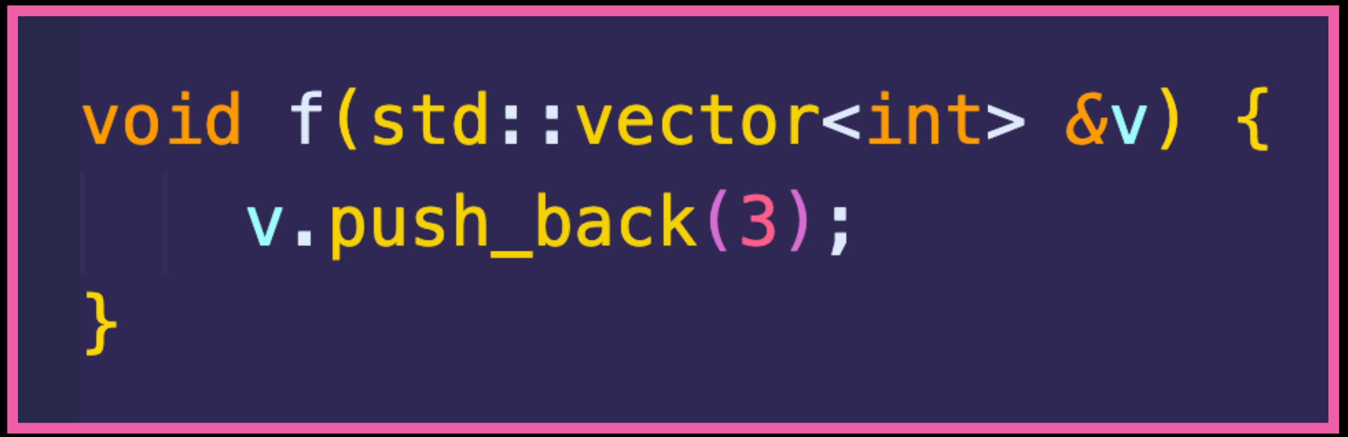


```
void f(std::vector<int> &v) {
    v.push_back(3);
}
```

Clang

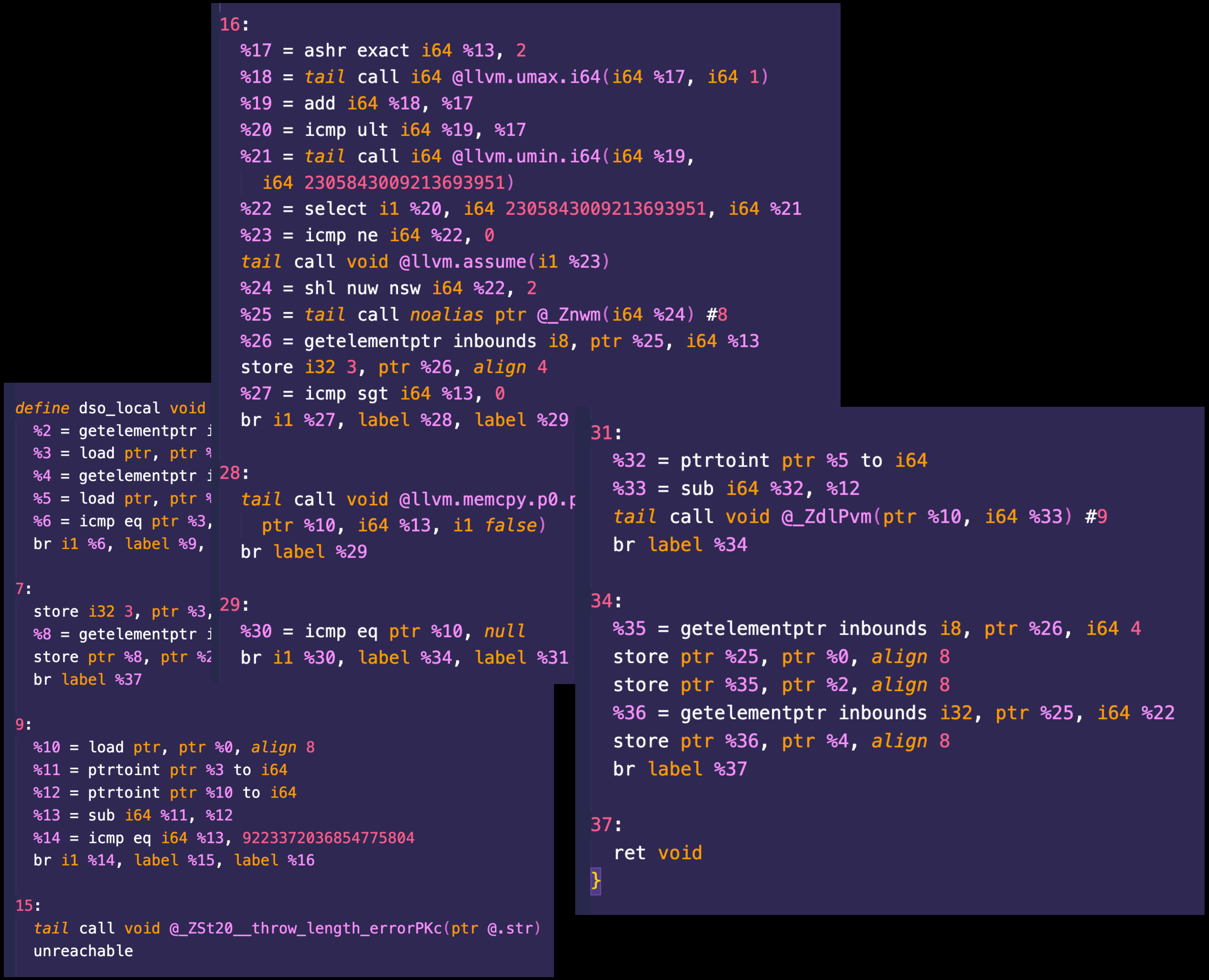
Reconstruction is hard

-emit-llvm -O1



```
void f(std::vector<int> &v) {
    v.push_back(3);
}
```

https://godbolt.org/z/zd15hK9cb



```
define dso_local void
  %2 = getelementptr i
  %3 = load ptr, ptr %
  %4 = getelementptr i
  %5 = load ptr, ptr %
  %6 = icmp eq ptr %3,
  br i1 %6, label %9,

7:
  store i32 3, ptr %3,
  %8 = getelementptr i
  store ptr %8, ptr %2
  br label %37

9:
  %10 = load ptr, ptr %0, align 8
  %11 = ptrtoint ptr %3 to i64
  %12 = ptrtoint ptr %10 to i64
  %13 = sub i64 %11, %12
  %14 = icmp eq i64 %13, 9223372036854775804
  br i1 %14, label %15, label %16

15:
  tail call void @_ZSt20__throw_length_errorPKc(ptr @.str)
  unreachable
```

```
16:
  %17 = ashr exact i64 %13, 2
  %18 = tail call i64 @llvm.umax.i64(i64 %17, i64 1)
  %19 = add i64 %18, %17
  %20 = icmp ult i64 %19, %17
  %21 = tail call i64 @llvm.umin.i64(i64 %19,
    i64 2305843009213693951)
  %22 = select i1 %20, i64 2305843009213693951, i64 %21
  %23 = icmp ne i64 %22, 0
  tail call void @llvm.assume(i1 %23)
  %24 = shl nuw nsw i64 %22, 2
  %25 = tail call noalias ptr @_Znwm(i64 %24) #8
  %26 = getelementptr inbounds i8, ptr %25, i64 %13
  store i32 3, ptr %26, align 4
  %27 = icmp sgt i64 %13, 0
  br i1 %27, label %28, label %29

28:
  tail call void @llvm.memcpy.p0.p
    ptr %10, i64 %13, i1 false)
  br label %29

29:
  %30 = icmp eq ptr %10, null
  br i1 %30, label %34, label %31
```

```
31:
  %32 = ptrtoint ptr %5 to i64
  %33 = sub i64 %32, %12
  tail call void @_ZdlPvm(ptr %10, i64 %33) #9
  br label %34

34:
  %35 = getelementptr inbounds i8, ptr %26, i64 4
  store ptr %25, ptr %0, align 8
  store ptr %35, ptr %2, align 8
  %36 = getelementptr inbounds i32, ptr %25, i64 %22
  store ptr %36, ptr %4, align 8
  br label %37

37:
  ret void
}
```

ClangIR

ClangIR (CIR)

High-level IR for Clang

- Represents C/C++ closely
- Translated out of Clang’s AST
- Move Clang onto the MLIR substrate
  - Use MLIR from C, C++ and extensions

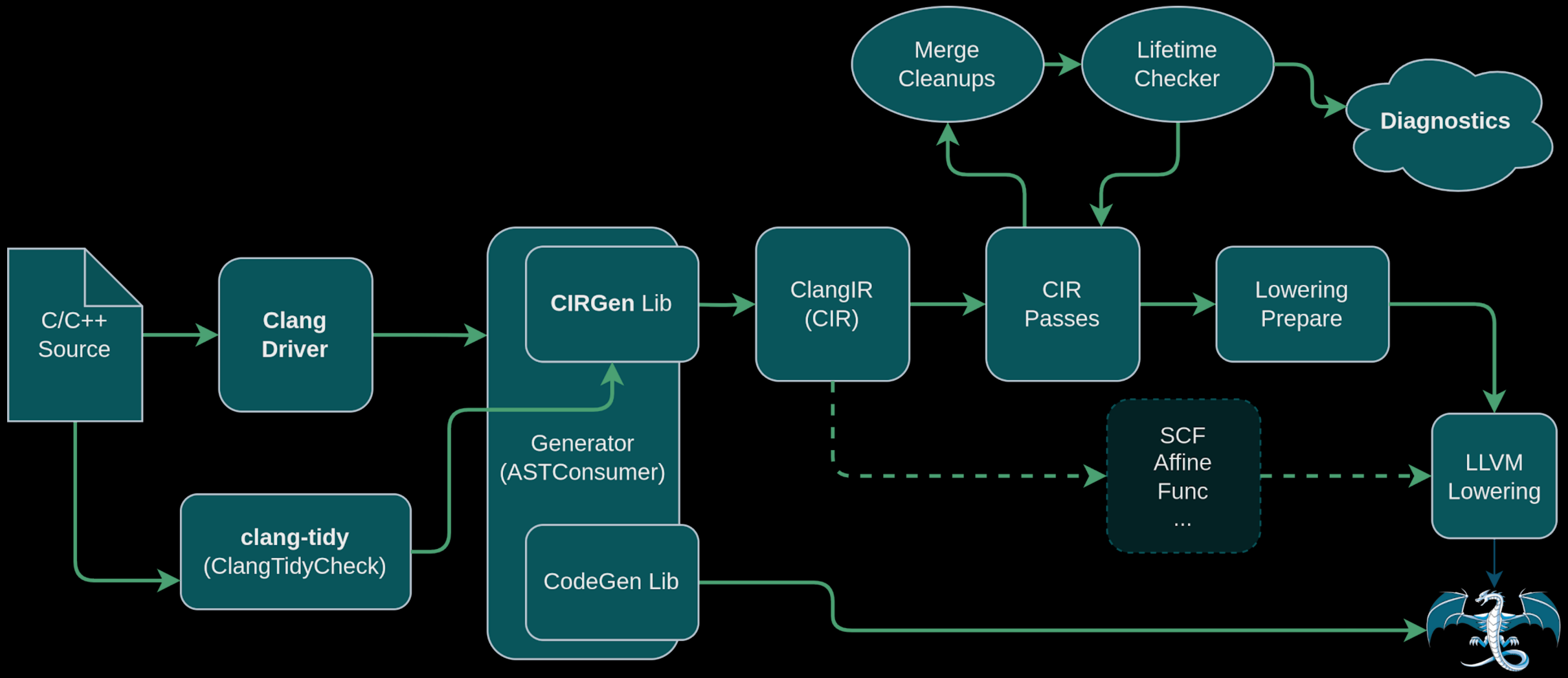
ClangIR (CIR)

Open Source

- llvm-project incubator, currently being upstreamed
    - June 2022: Introductory RFC to LLVM project
    - Feb 2024: Upstream RFC in Feb 2024 (accepted)
- Github, 46 unique contributors since 2021
- Industry commitment

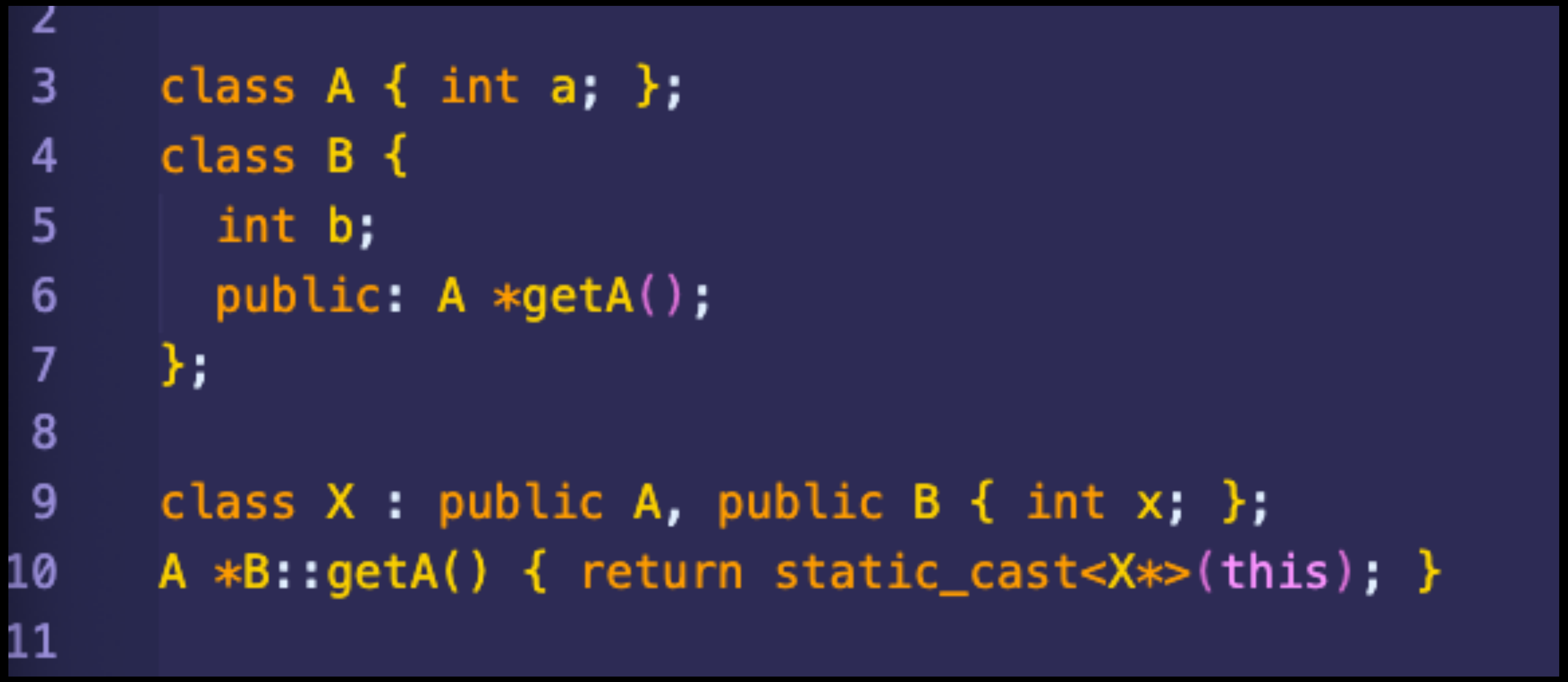
ClangIR (CIR)

Pipeline purview



CIR example

High-level IR for Clang

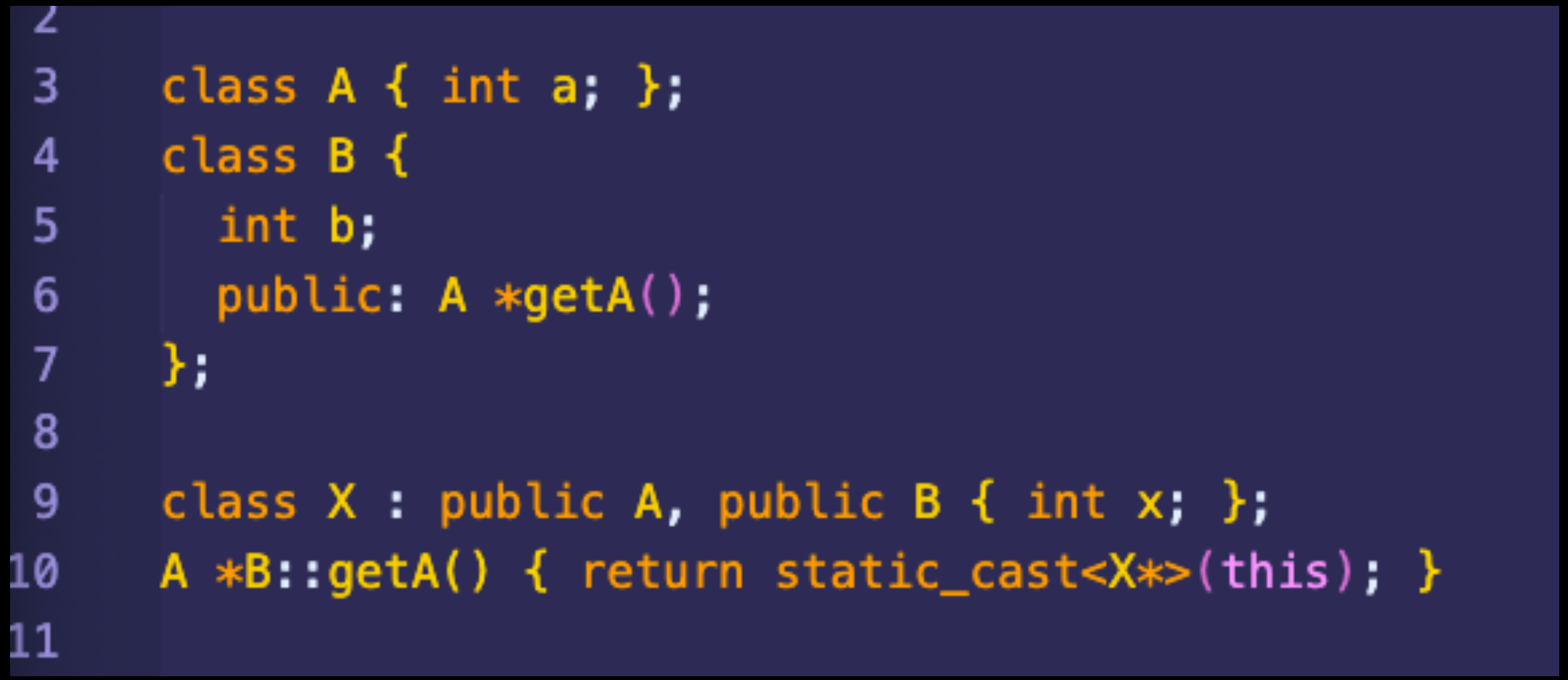


```
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

CIR example

High-level IR for Clang

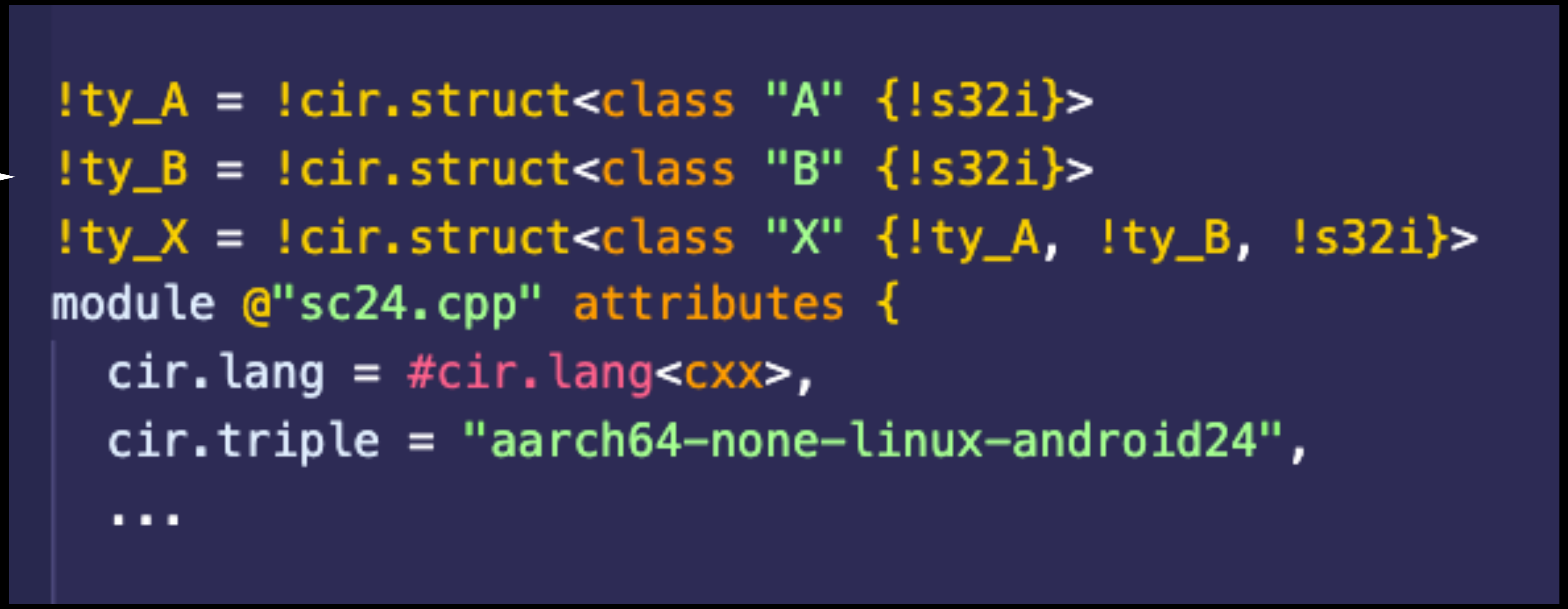


```
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

https://godbolt.org/z/MTaPP7xdc

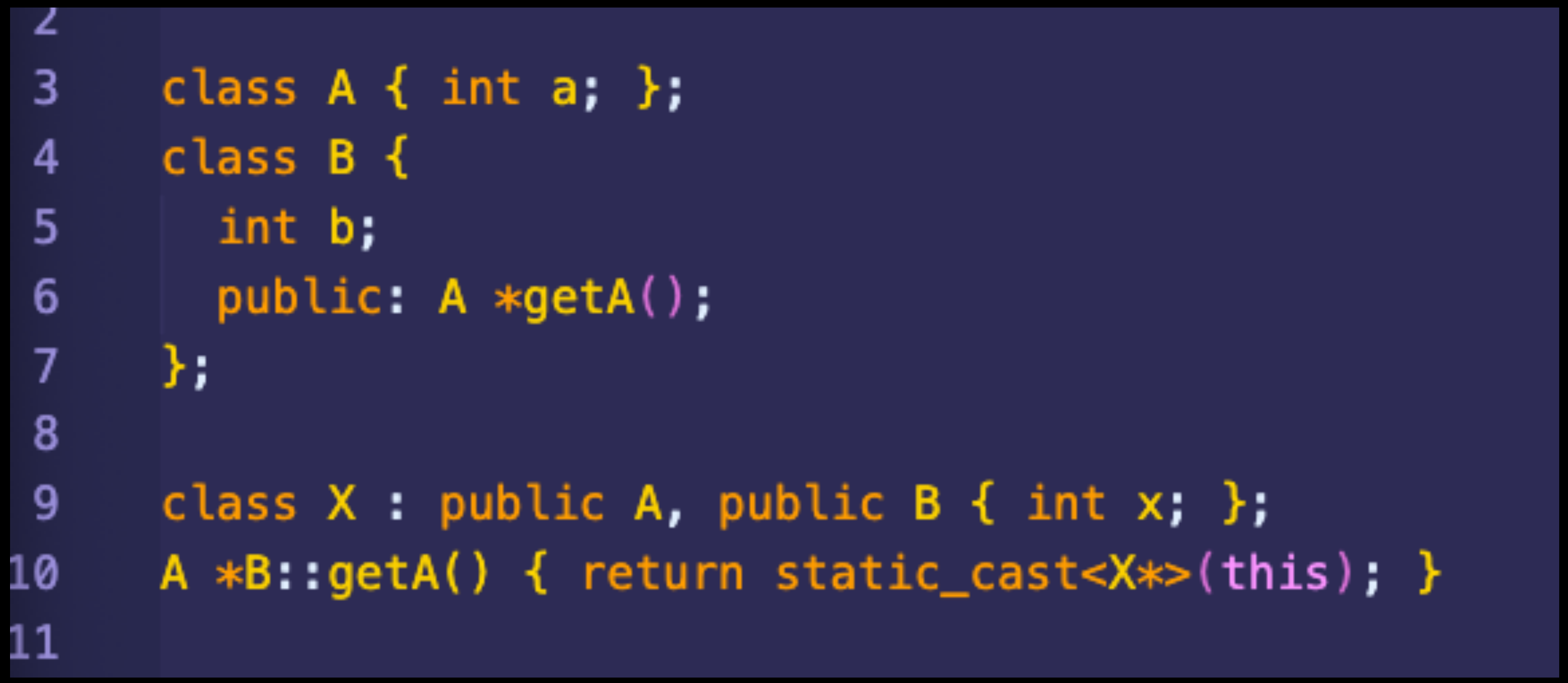
Types, ABI information



```
!ty_A = !cir.struct<class "A" {!s32i}>
!ty_B = !cir.struct<class "B" {!s32i}>
!ty_X = !cir.struct<class "X" {!ty_A, !ty_B, !s32i}>
module @"sc24.cpp" attributes {
  cir.lang = #cir.lang<cxx>,
  cir.triple = "aarch64-none-linux-android24",
  ...
```

CIR example

High-level IR for Clang

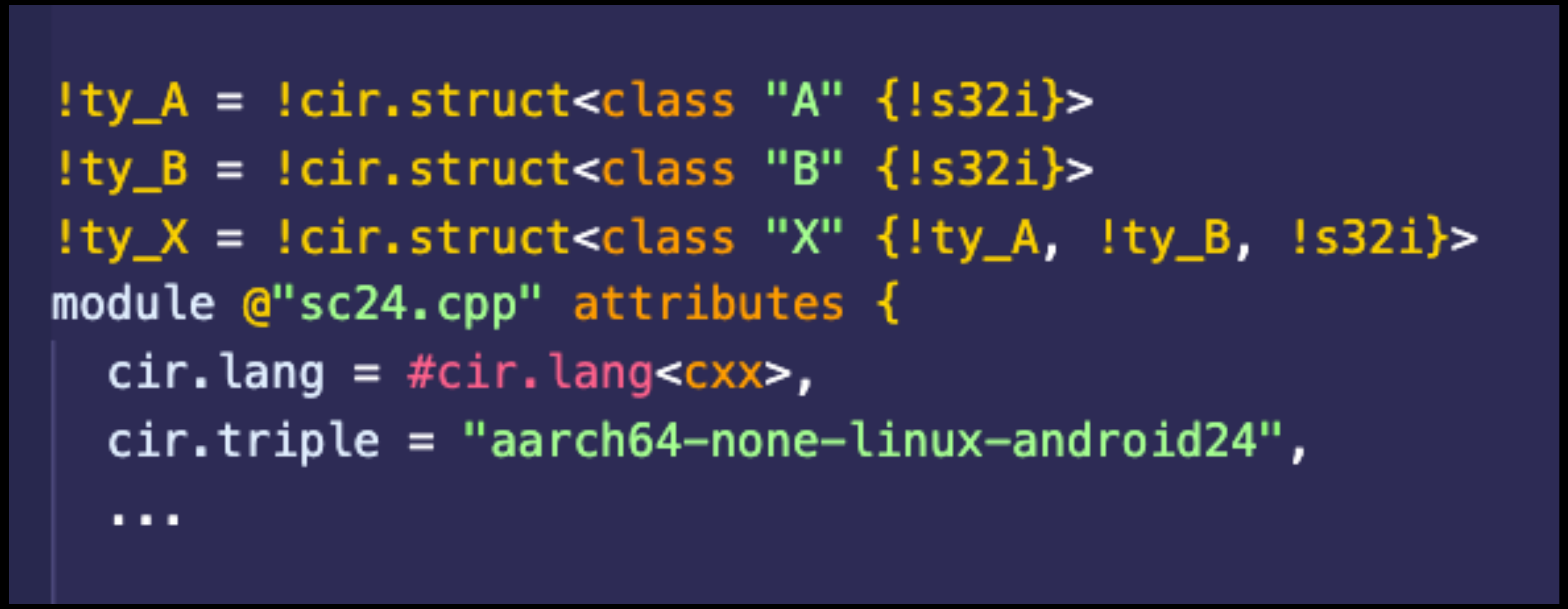


```
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

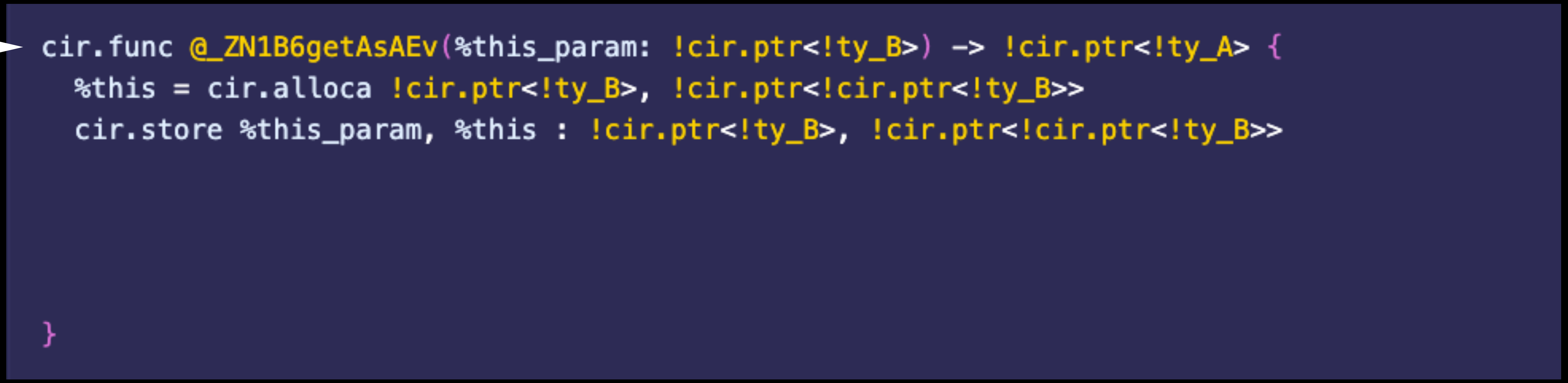
https://godbolt.org/z/MTaPP7xdc

Types, ABI information



```
!ty_A = !cir.struct<class "A" {!s32i}>
!ty_B = !cir.struct<class "B" {!s32i}>
!ty_X = !cir.struct<class "X" {!ty_A, !ty_B, !s32i}>
module @"sc24.cpp" attributes {
  cir.lang = #cir.lang<cxx>,
  cir.triple = "aarch64-none-linux-android24",
  ...
```

C++ idioms



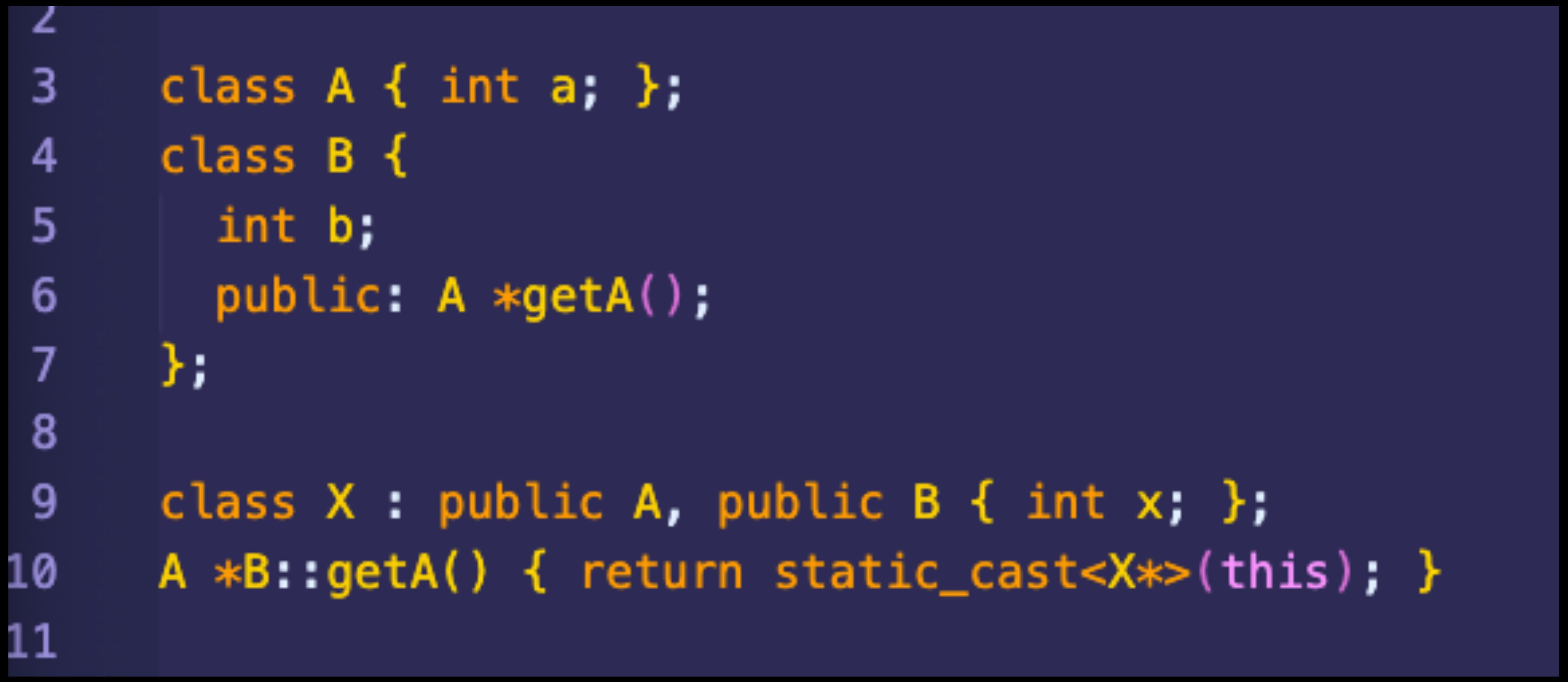
```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>




}
```

CIR example

High-level IR for Clang

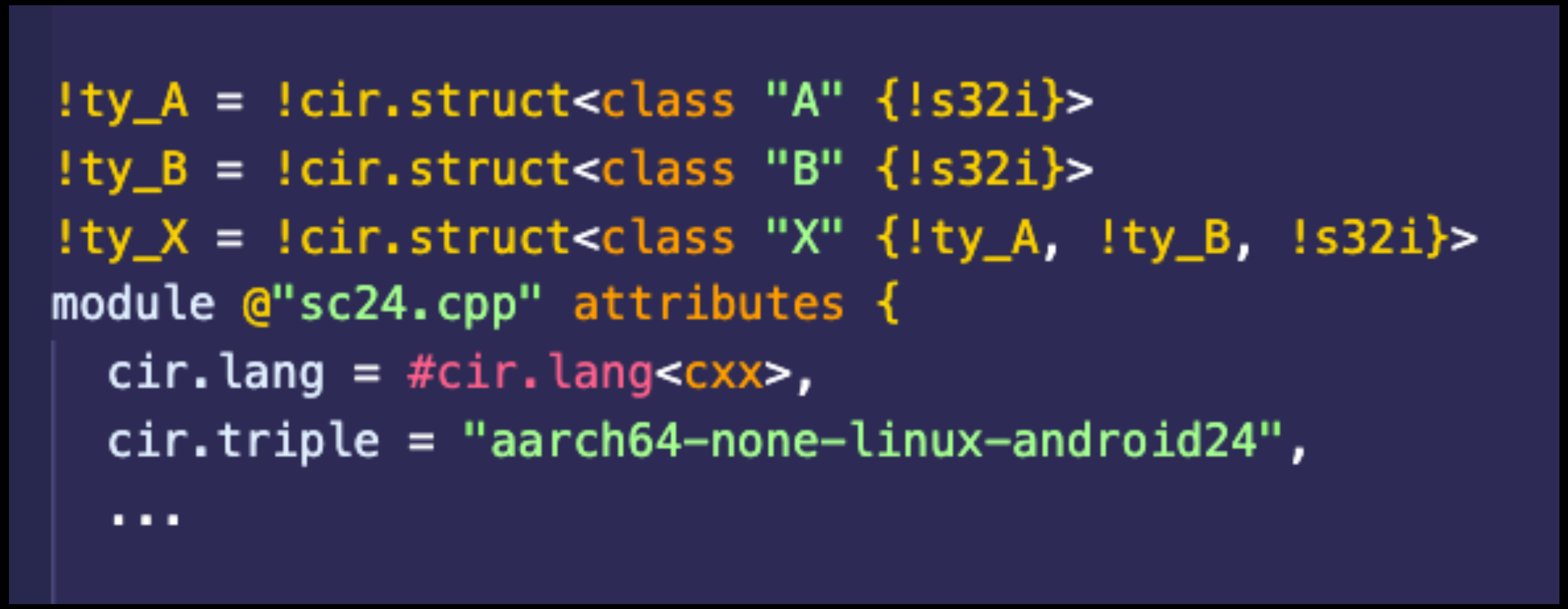


```
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

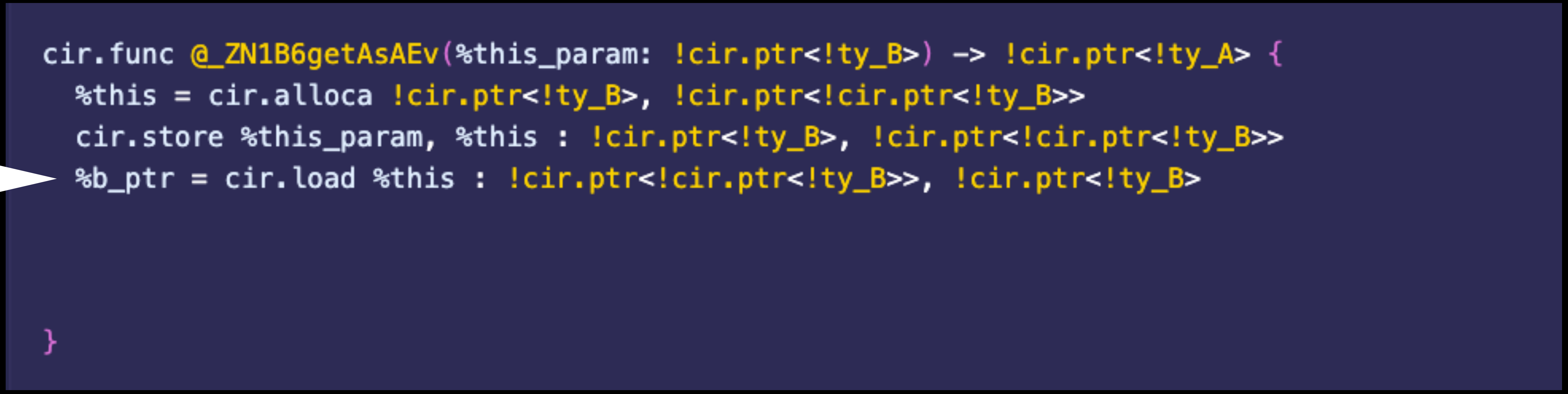
https://godbolt.org/z/MTaPP7xdc

Types, ABI information



```
!ty_A = !cir.struct<class "A" {!s32i}>
!ty_B = !cir.struct<class "B" {!s32i}>
!ty_X = !cir.struct<class "X" {!ty_A, !ty_B, !s32i}>
module @"sc24.cpp" attributes {
  cir.lang = #cir.lang<cxx>,
  cir.triple = "aarch64-none-linux-android24",
  ...
```

C++ idioms



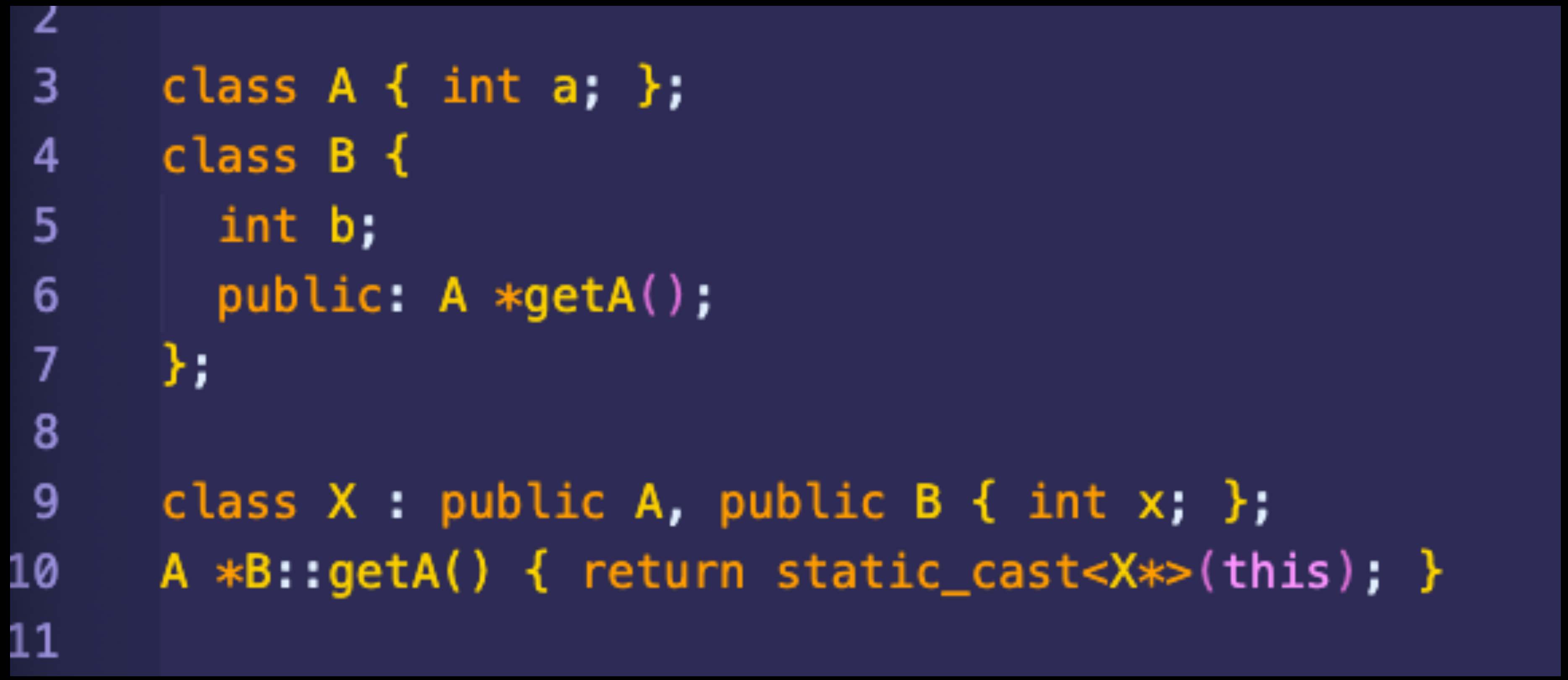
```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  %b_ptr = cir.load %this : !cir.ptr<!cir.ptr<!ty_B>>, !cir.ptr<!ty_B>



}
```

CIR example

High-level IR for Clang

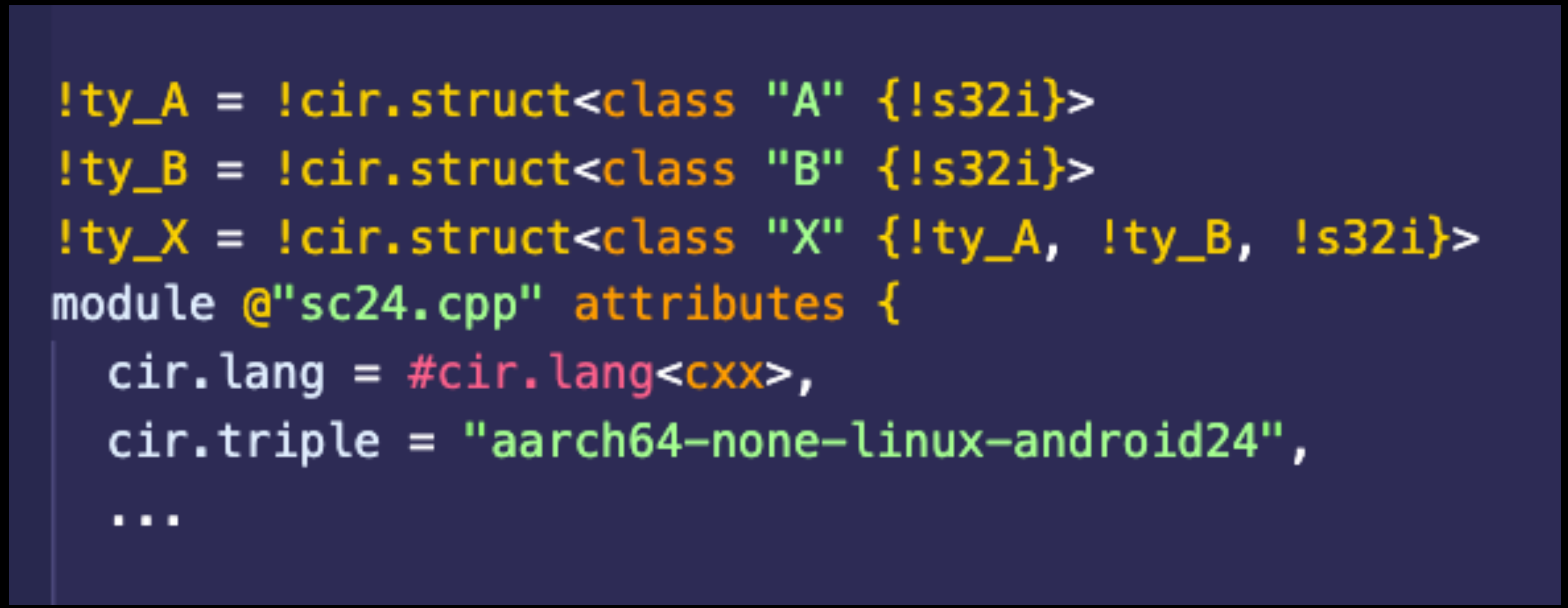


```
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

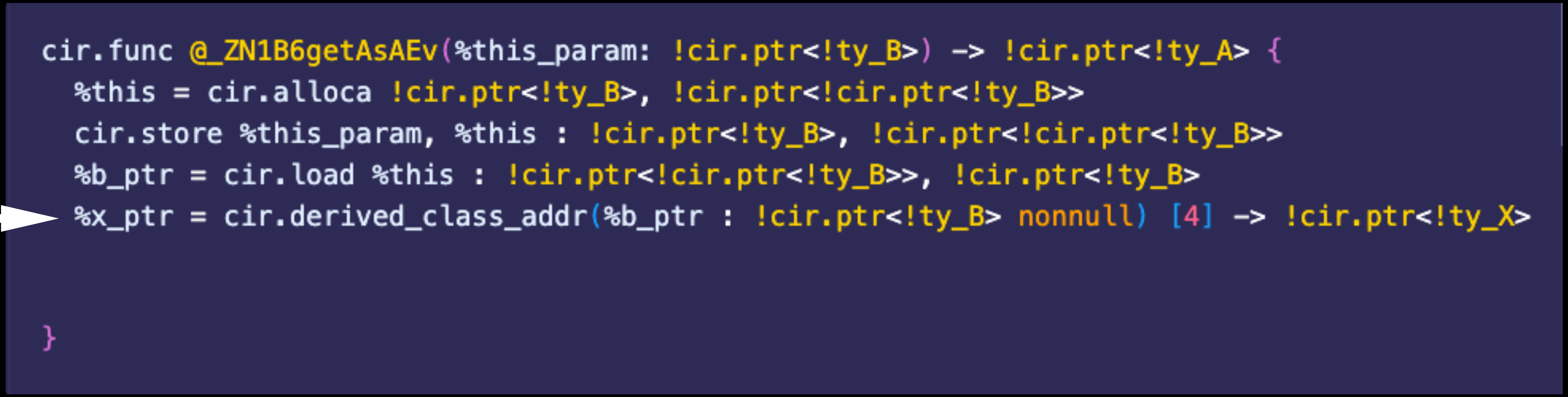
https://godbolt.org/z/MTaPP7xdc

Types, ABI information



```
!ty_A = !cir.struct<class "A" {!s32i}>
!ty_B = !cir.struct<class "B" {!s32i}>
!ty_X = !cir.struct<class "X" {!ty_A, !ty_B, !s32i}>
module @"sc24.cpp" attributes {
  cir.lang = #cir.lang<cxx>,
  cir.triple = "aarch64-none-linux-android24",
  ...
```

C++ idioms

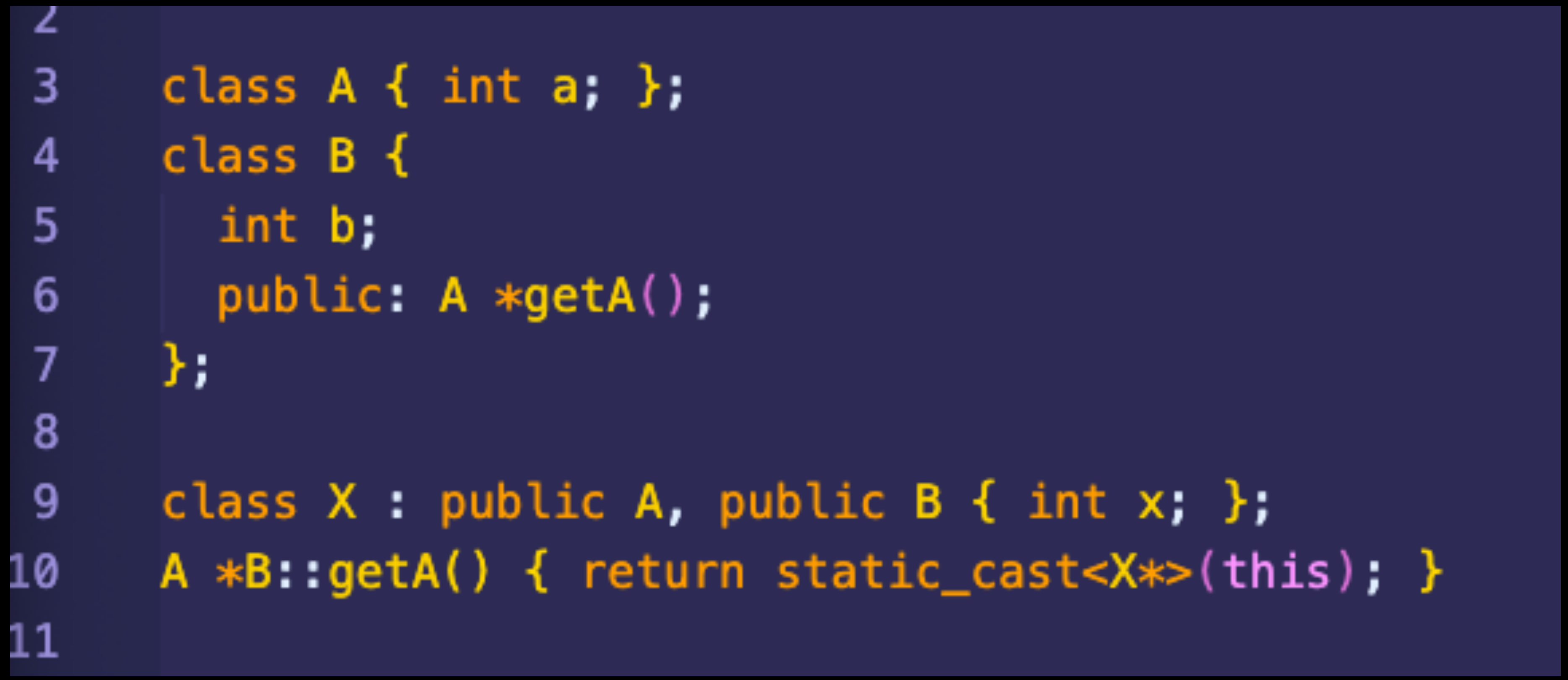


```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  %b_ptr = cir.load %this : !cir.ptr<!cir.ptr<!ty_B>>, !cir.ptr<!ty_B>
  %x_ptr = cir.derived_class_addr(%b_ptr : !cir.ptr<!ty_B> nonnull) [4] -> !cir.ptr<!ty_X>


}
```

CIR example

High-level IR for Clang

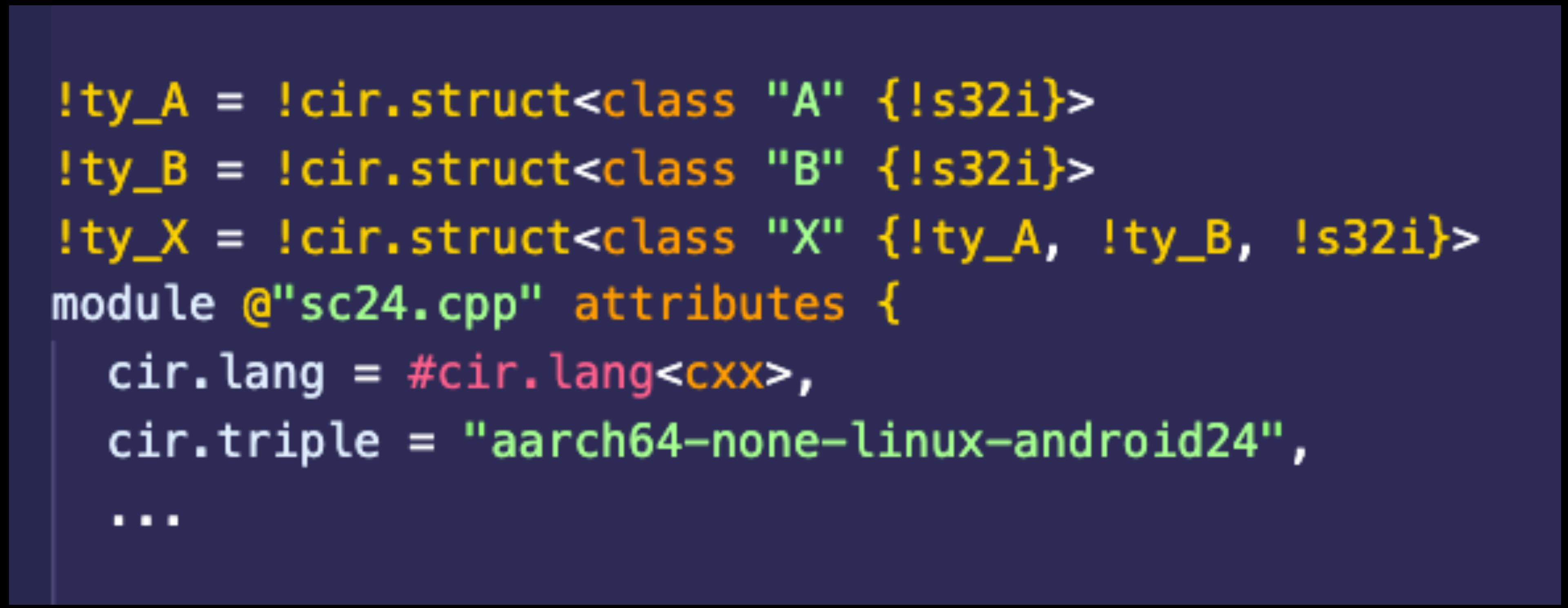


```cpp
class A { int a; };
class B {
  int b;
  public: A *getA();
};

class X : public A, public B { int x; };
A *B::getA() { return static_cast<X*>(this); }
```

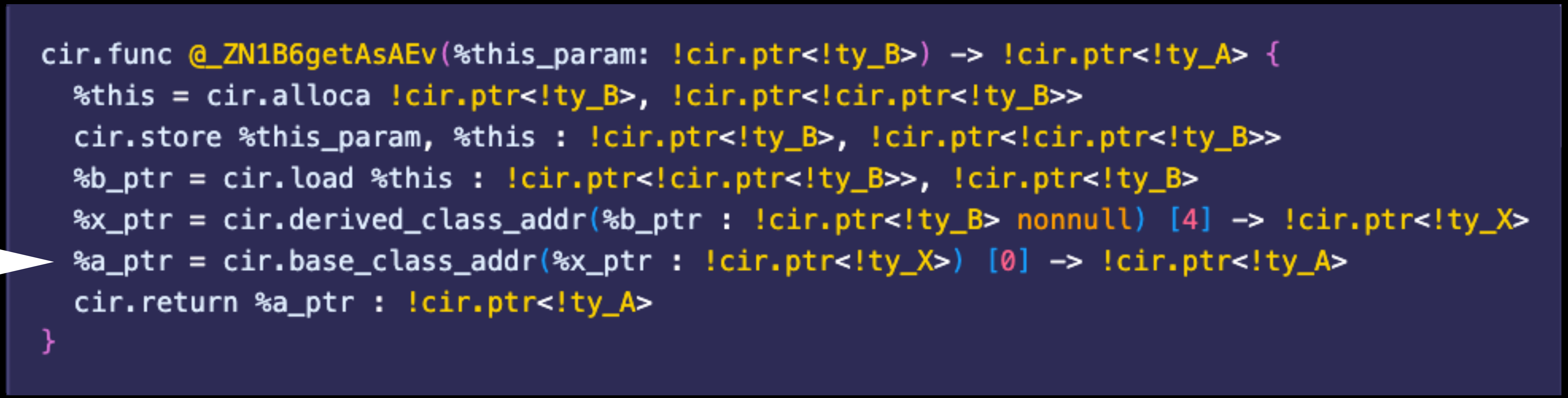
https://godbolt.org/z/MTaPP7xdc

Types, ABI information



```
!ty_A = !cir.struct<class "A" {!s32i}>
!ty_B = !cir.struct<class "B" {!s32i}>
!ty_X = !cir.struct<class "X" {!ty_A, !ty_B, !s32i}>
module @"sc24.cpp" attributes {
  cir.lang = #cir.lang<cxx>,
  cir.triple = "aarch64-none-linux-android24",
  ...
```

C++ idioms



```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  %b_ptr = cir.load %this : !cir.ptr<!cir.ptr<!ty_B>>, !cir.ptr<!ty_B>
  %x_ptr = cir.derived_class_addr(%b_ptr : !cir.ptr<!ty_B> nonnull) [4] -> !cir.ptr<!ty_X>
  %a_ptr = cir.base_class_addr(%x_ptr : !cir.ptr<!ty_X>) [0] -> !cir.ptr<!ty_A>
  cir.return %a_ptr : !cir.ptr<!ty_A>
}
```

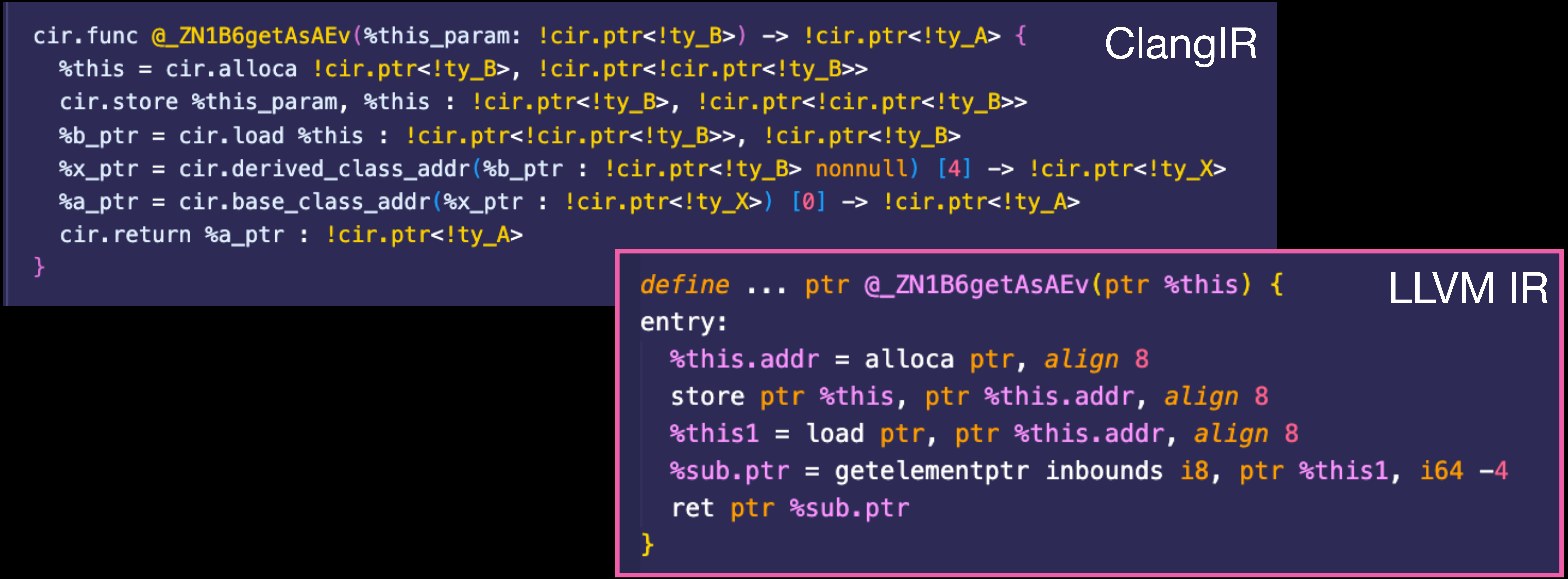
ClangIR progress

LLVM IR backend

- CIR to LLVM IR dialect pass
- Supports: x86_64, ARM64 and **SPIRV** LLVM IR
- Initial **OpenCL** support, toy O**penMP** support
- Builds SPEC2017 C, 90% of Social App
- C++ under heavy development (WIP building libc++)

LLVM lowering

Different representation levels



ClangIR

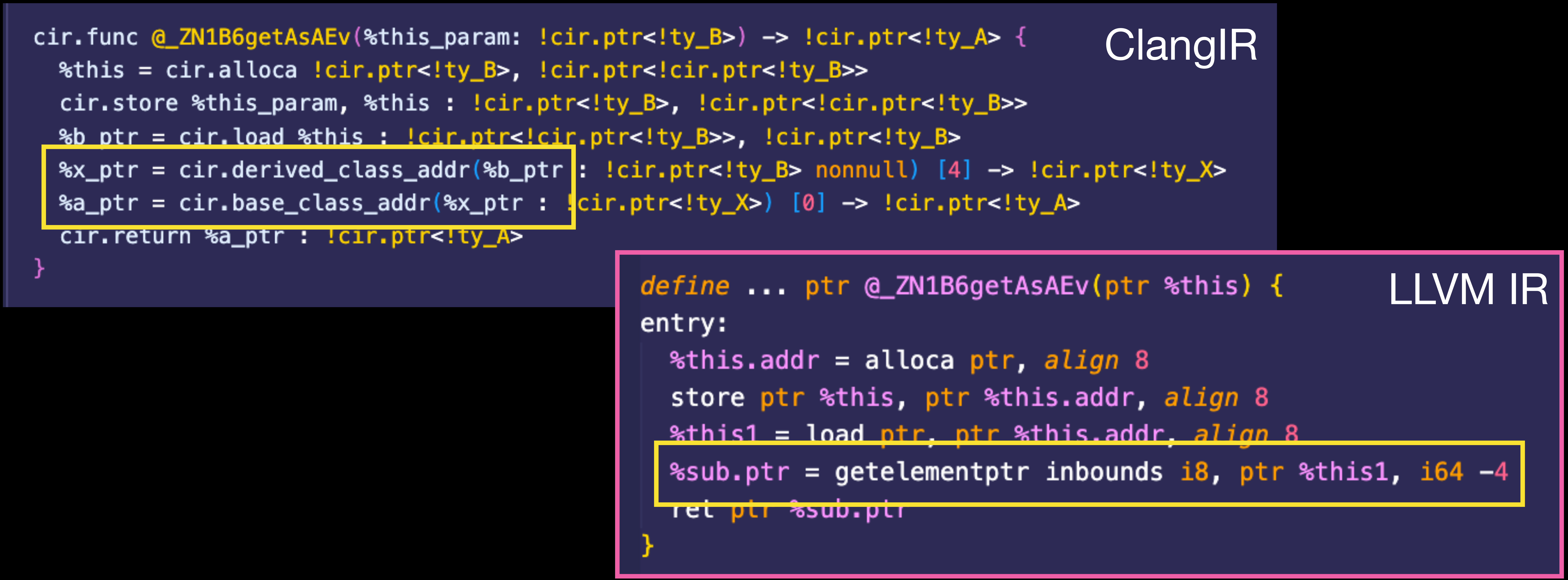
```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  %b_ptr = cir.load %this : !cir.ptr<!cir.ptr<!ty_B>>, !cir.ptr<!ty_B>
  %x_ptr = cir.derived_class_addr(%b_ptr : !cir.ptr<!ty_B> nonnull) [4] -> !cir.ptr<!ty_X>
  %a_ptr = cir.base_class_addr(%x_ptr : !cir.ptr<!ty_X>) [0] -> !cir.ptr<!ty_A>
  cir.return %a_ptr : !cir.ptr<!ty_A>
}
```

LLVM IR

```
define ... ptr @_ZN1B6getAsAEv(ptr %this) {
entry:
  %this.addr = alloca ptr, align 8
  store ptr %this, ptr %this.addr, align 8
  %this1 = load ptr, ptr %this.addr, align 8
  %sub.ptr = getelementptr inbounds i8, ptr %this1, i64 -4
  ret ptr %sub.ptr
}
```

LLVM lowering

Different representation levels



ClangIR

```
cir.func @_ZN1B6getAsAEv(%this_param: !cir.ptr<!ty_B>) -> !cir.ptr<!ty_A> {
  %this = cir.alloca !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  cir.store %this_param, %this : !cir.ptr<!ty_B>, !cir.ptr<!cir.ptr<!ty_B>>
  %b_ptr = cir.load %this : !cir.ptr<!cir.ptr<!ty_B>>, !cir.ptr<!ty_B>
  %x_ptr = cir.derived_class_addr(%b_ptr : !cir.ptr<!ty_B> nonnull) [4] -> !cir.ptr<!ty_X>
  %a_ptr = cir.base_class_addr(%x_ptr : !cir.ptr<!ty_X>) [0] -> !cir.ptr<!ty_A>
  cir.return %a_ptr : !cir.ptr<!ty_A>
}
```

LLVM IR

```
define ... ptr @_ZN1B6getAsAEv(ptr %this) {
entry:
  %this.addr = alloca ptr, align 8
  store ptr %this, ptr %this.addr, align 8
  %this1 = load ptr, ptr %this.addr, align 8
  %sub.ptr = getelementptr inbounds i8, ptr %this1, i64 -4
  ret ptr %sub.ptr
}
```

Other Lowering

Easy to write conversions

- Built on top of MLIR
- CIR to MLIR "standard" dialects:
  - affine, arithmetic, mermen, scf, math, etc
  - Not as advanced as LLVM lowering

Tooling and Usages

Integration with existing tools

- CIR support in Compiler Explorer
- C++ lifetime analysis
  - Handle most of C++ support constructs
  - clang-tidy & clangd integration
- PoC of cross-library optimization framework

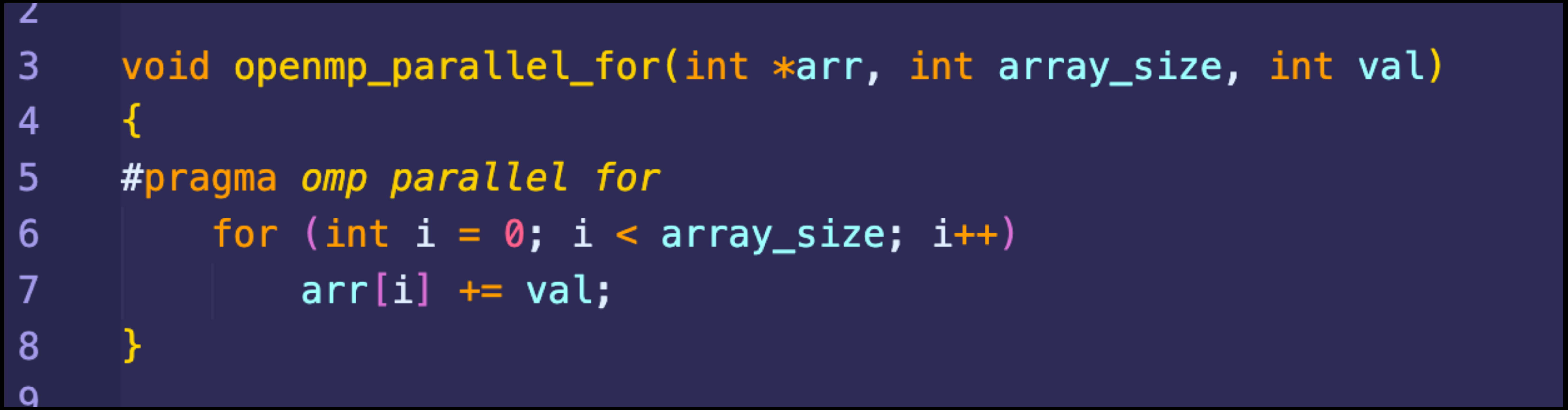
Why the HPC community should care?

HPC & ClangIR

- Lower C/C++ extensions to MLIR
- Mix CIR with downstream and custom dialects
- High level mapping of specific C/C++ extension idioms
  - Domain specific optimizations, analysis, diagnostics
  - Avoid premature lowering

Case study

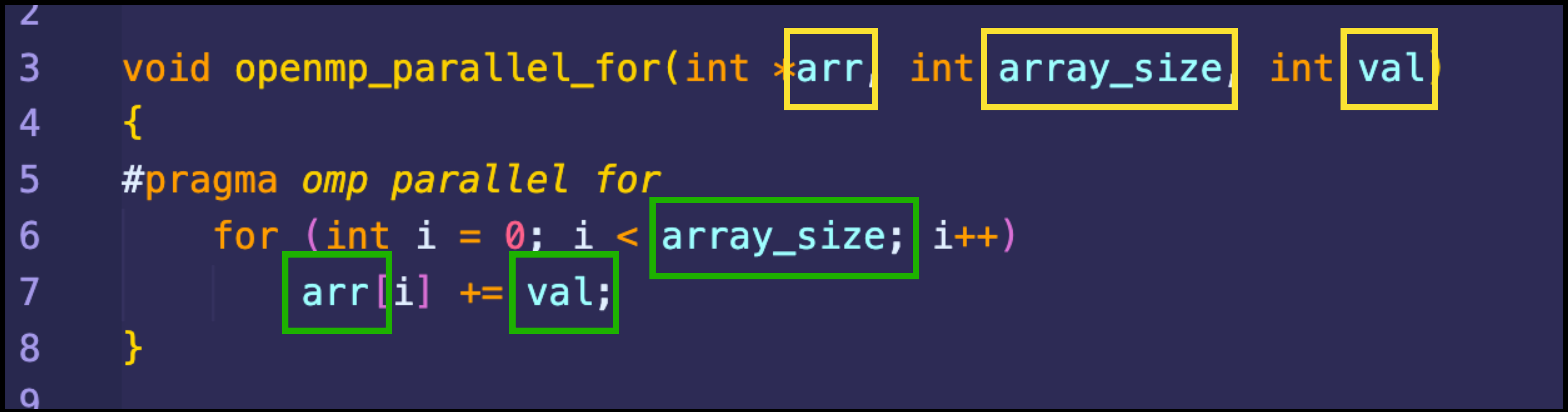
OpenMP in Clang



```
void openmp_parallel_for(int *arr, int array_size, int val)
{
#pragma omp parallel for
    for (int i = 0; i < array_size; i++)
        arr[i] += val;
}
```

Case study

OpenMP in Clang

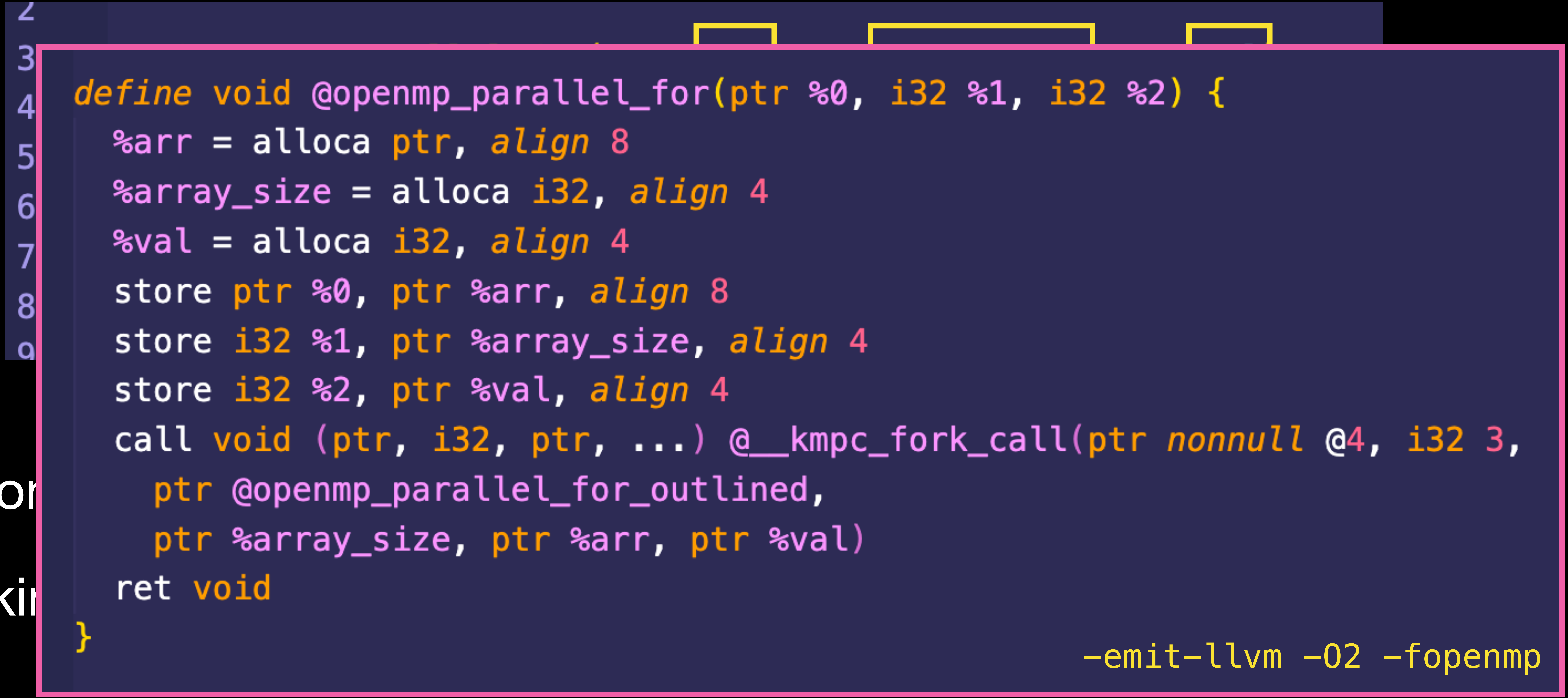


```
void openmp_parallel_for(int *arr, int array_size, int val)
{
#pragma omp parallel for
    for (int i = 0; i < array_size; i++)
        arr[i] += val;
}
```

- Read-only variables above
- What kind of code generation we get?

Case study

OpenMP in Clang



```
define void @openmp_parallel_for(ptr %0, i32 %1, i32 %2) {
  %arr = alloca ptr, align 8
  %array_size = alloca i32, align 4
  %val = alloca i32, align 4
  store ptr %0, ptr %arr, align 8
  store i32 %1, ptr %array_size, align 4
  store i32 %2, ptr %val, align 4
  call void (ptr, i32, ptr, ...) @__kmpc_fork_call(ptr nonnull @4, i32 3,
    ptr @openmp_parallel_for_outlined,
    ptr %array_size, ptr %arr, ptr %val)
  ret void
}
```

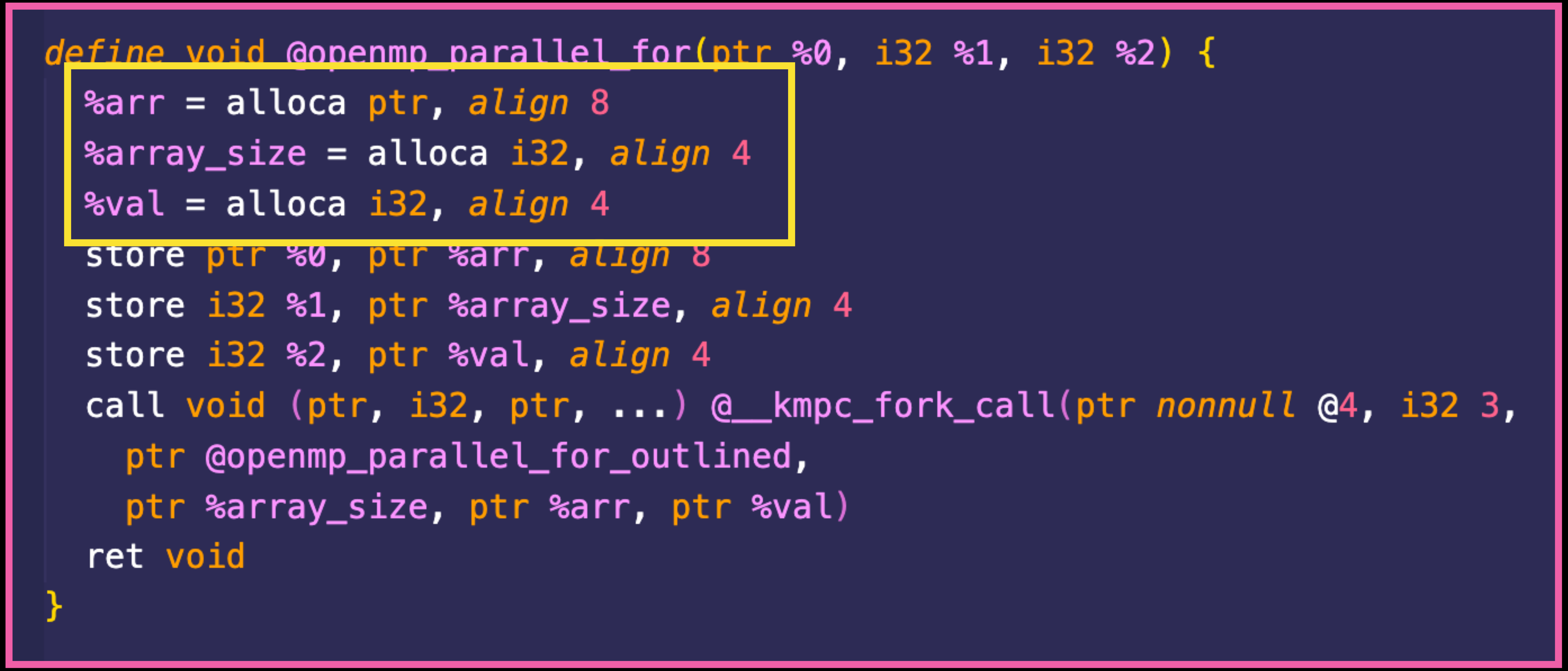
-emit-llvm -O2 -fopenmp

- Read-or
- What kir

https://godbolt.org/z/EsYodTKW9

Case study

OpenMP in Clang

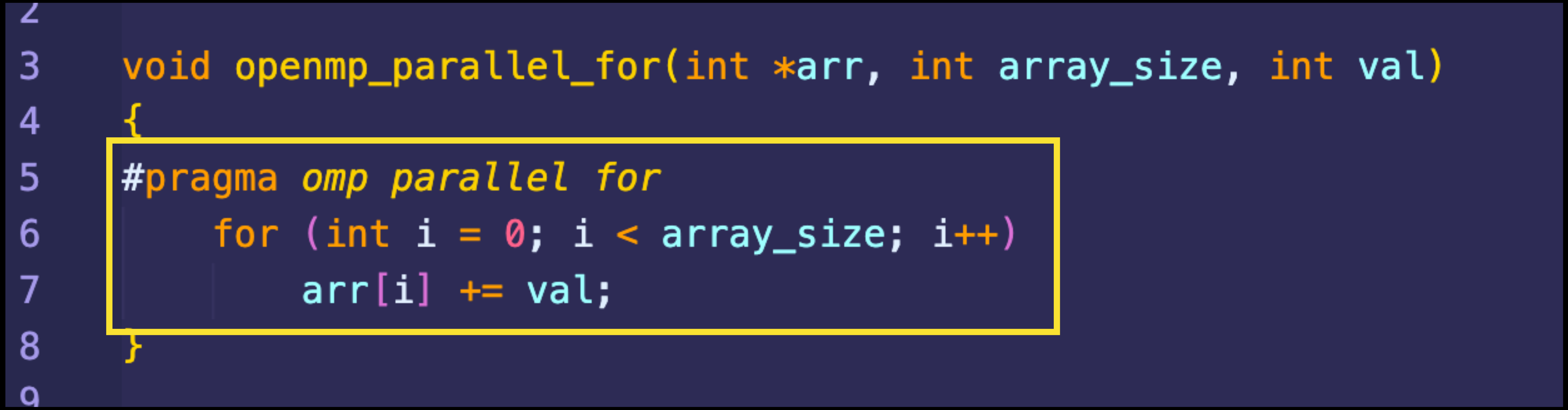


```
define void @openmp_parallel_for(ptr %0, i32 %1, i32 %2) {
  %arr = alloca ptr, align 8
  %array_size = alloca i32, align 4
  %val = alloca i32, align 4
  store ptr %0, ptr %arr, align 8
  store i32 %1, ptr %array_size, align 4
  store i32 %2, ptr %val, align 4
  call void (ptr, i32, ptr, ...) @__kmpc_fork_call(ptr nonnull @4, i32 3,
    ptr @openmp_parallel_for_outlined,
    ptr %array_size, ptr %arr, ptr %val)
  ret void
}
```

- Unnecessary alloca's before forking

Case study

OpenMP in Clang

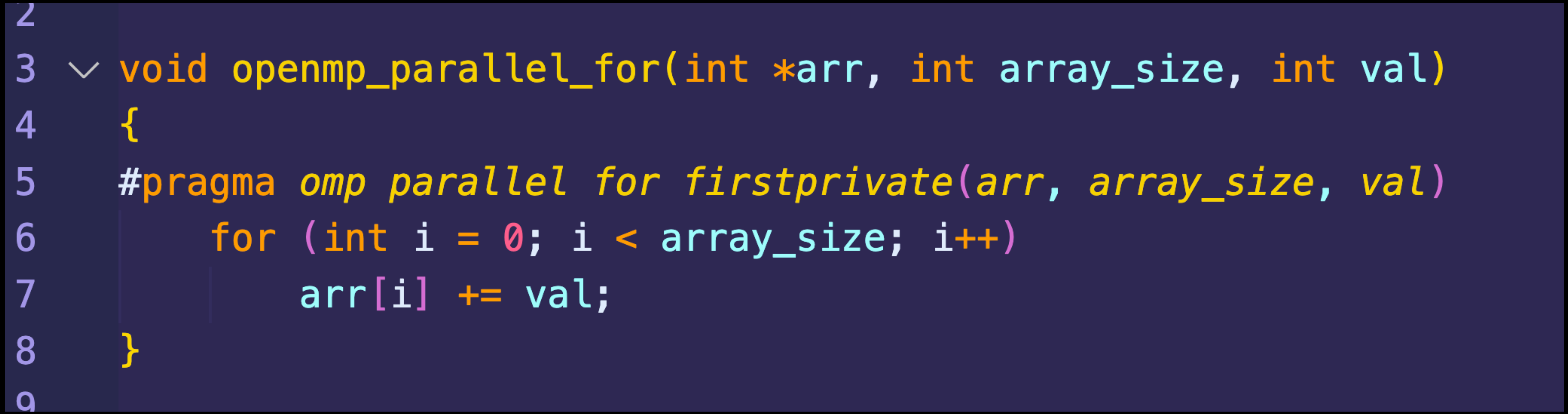


```
void openmp_parallel_for(int *arr, int array_size, int val)
{
#pragma omp parallel for
    for (int i = 0; i < array_size; i++)
        arr[i] += val;
}
```

- Function is outlined prematurely, too late for classic clang
- ClangIR: mix OpenMP + CIR
    - mem2reg remove allocas

Case study

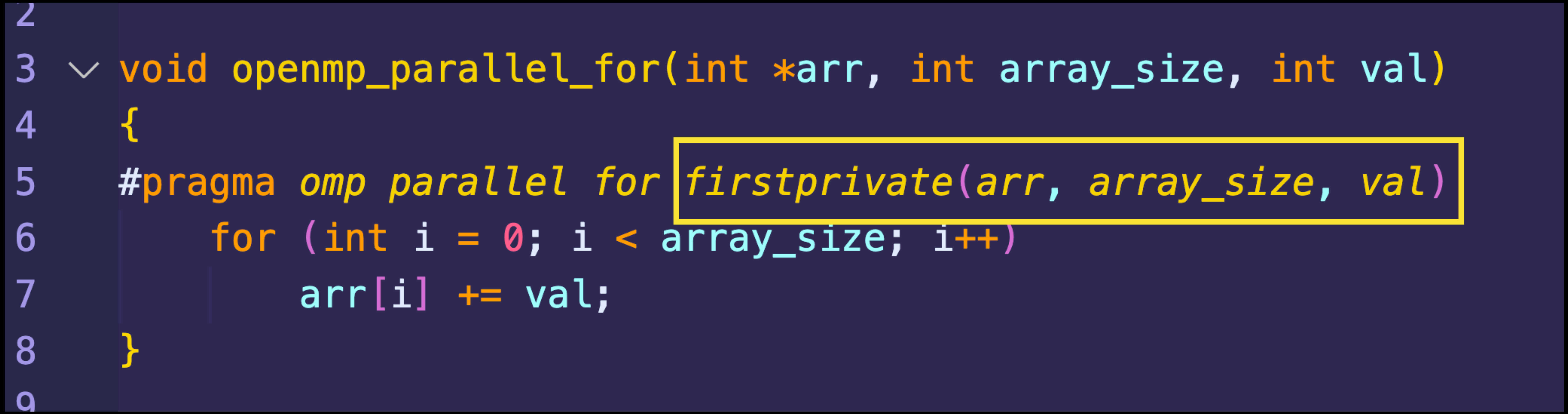
OpenMP in Clang



```c
void openmp_parallel_for(int *arr, int array_size, int val)
{
#pragma omp parallel for firstprivate(arr, array_size, val)
    for (int i = 0; i < array_size; i++)
        arr[i] += val;
}
```

Case study

OpenMP in Clang

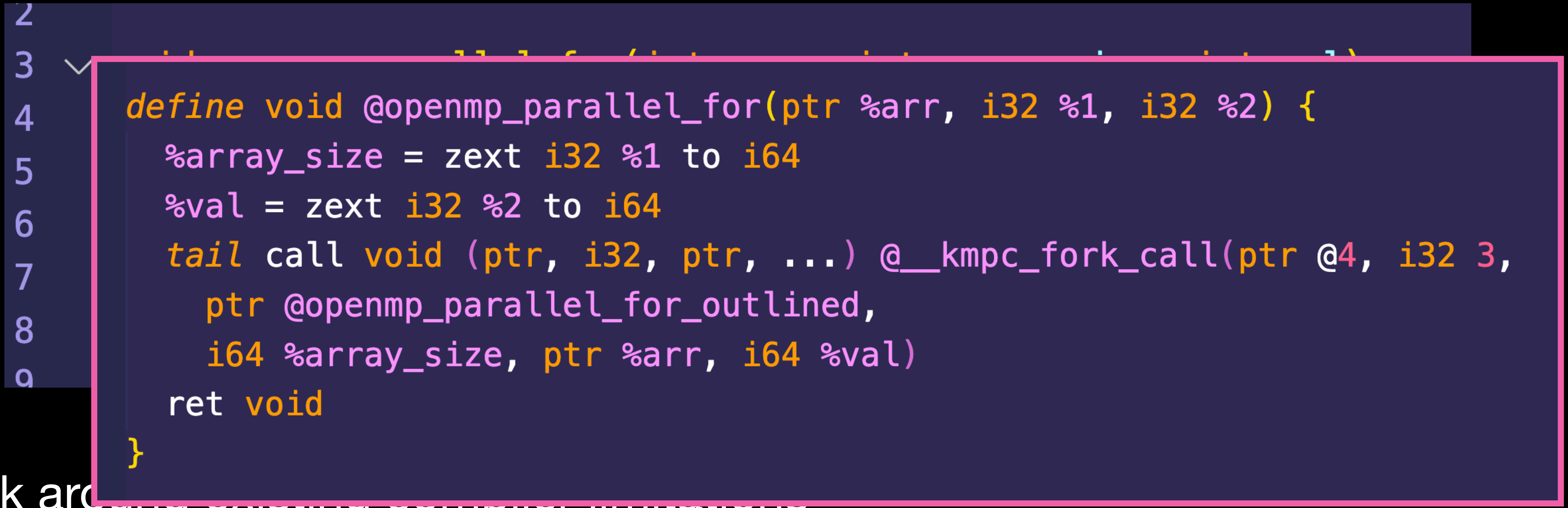


```
void openmp_parallel_for(int *arr, int array_size, int val)
{
#pragma omp parallel for firstprivate(arr, array_size, val)
    for (int i = 0; i < array_size; i++)
        arr[i] += val;
}
```

- Work around existing compiler limitations
- No diagnostics on “writes" to those variables

Case study

OpenMP in Clang



```
define void @openmp_parallel_for(ptr %arr, i32 %1, i32 %2) {
  %array_size = zext i32 %1 to i64
  %val = zext i32 %2 to i64
  tail call void (ptr, i32, ptr, ...) @__kmpc_fork_call(ptr @4, i32 3,
    ptr @openmp_parallel_for_outlined,
    i64 %array_size, ptr %arr, i64 %val)
  ret void
}
```

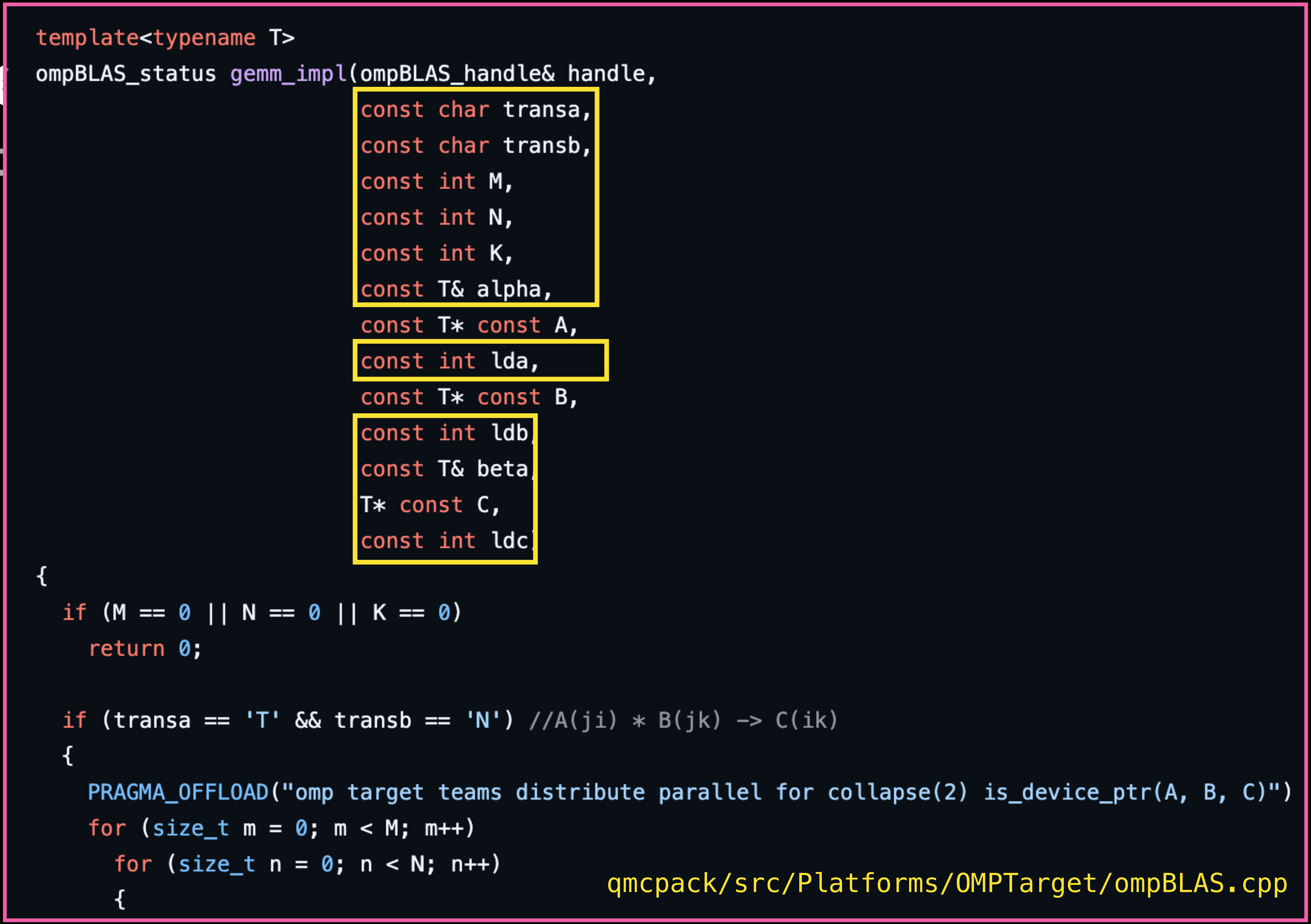
- Work around existing compiler limitations
- No diagnostics on “writes" to those variables

Case study

Does this happen in real code?

Case

Does t



```
template<typename T>
ompBLAS_status gemm_impl(ompBLAS_handle& handle,
                         const char transa,
                         const char transb,
                         const int M,
                         const int N,
                         const int K,
                         const T& alpha,
                         const T* const A,
                         const int lda,
                         const T* const B,
                         const int ldb,
                         const T& beta,
                         T* const C,
                         const int ldc)
{
  if (M == 0 || N == 0 || K == 0)
    return 0;

  if (transa == 'T' && transb == 'N') //A(ji) * B(jk) -> C(ik)
  {
    PRAGMA_OFFLOAD("omp target teams distribute parallel for collapse(2) is_device_ptr(A, B, C)")
    for (size_t m = 0; m < M; m++)
      for (size_t n = 0; n < N; n++)
      {
```

qmcpack/src/Platforms/OMPTarget/ompBLAS.cpp

Takeaway

- Premature lowering hurts
- A higher level for C, C++ and extensions brings a clear benefit to the Clang compiler community (looking at you HPC folks)
- ClangIR is under heavy development, joins us!

Resources

- clangir.org
- Compiler explorer (ClangIR branch)
- C/C++ MLIR WG meeting monthly (1st Monday of the month)
- Discord: #clangir
- Github: https://github.com/llvm/clangir

Questions