Skip the FFI!

Embedding Clang for C Interoperability

Jordan Rose
Compiler Engineer, Apple

John McCall
Compiler Engineer, Apple

Problem

Problem

Languages don’t exist in a vacuum

Problem

Languages don’t exist in a vacuum

But C has its own ABI

Problem

Languages don't exist in a vacuum

But C has its own ABI

And its APIs are written in C, not ${LANG}

Solutions?

Solutions?

Manually write glue code (JNI, Python, Ruby)

Solutions?

Manually write glue code (JNI, Python, Ruby)

Generate the glue code (SWIG)

Solutions?

Manually write glue code (JNI, Python, Ruby)

Generate the glue code (SWIG)

Extend C (C++, Objective-C)

Better solution:
just use Clang

Embedding Clang for C Interoperability

- Clang as a library
- Importing from C
- ABI compatibility
- Sharing an `llvm::Module`

Goal

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

```
let flipped = flipOverXAxis(originalPoint)
```

Goal

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

```
typedef struct {
  float x, y;
} Point2f;
```

Goal

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

```
let flipped = flipOverXAxis(originalPoint)
```

Goal

No external symbol!

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

```
let flipped = flipOverXAxis(originalPoint)
```

Goal

No external symbol!

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

Handled differently on different platforms!

```
let flipped = flipOverXAxis(originalPoint)
```

From C to ${LANG}…

Roadmap

Roadmap

Set up a `clang::CompilerInstance`

Roadmap

Set up a `clang::CompilerInstance`

Load Clang modules

Roadmap

Set up a `clang::CompilerInstance`

Load Clang modules

Import declarations we care about

Setting up a `clang::CompilerInstance`

Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

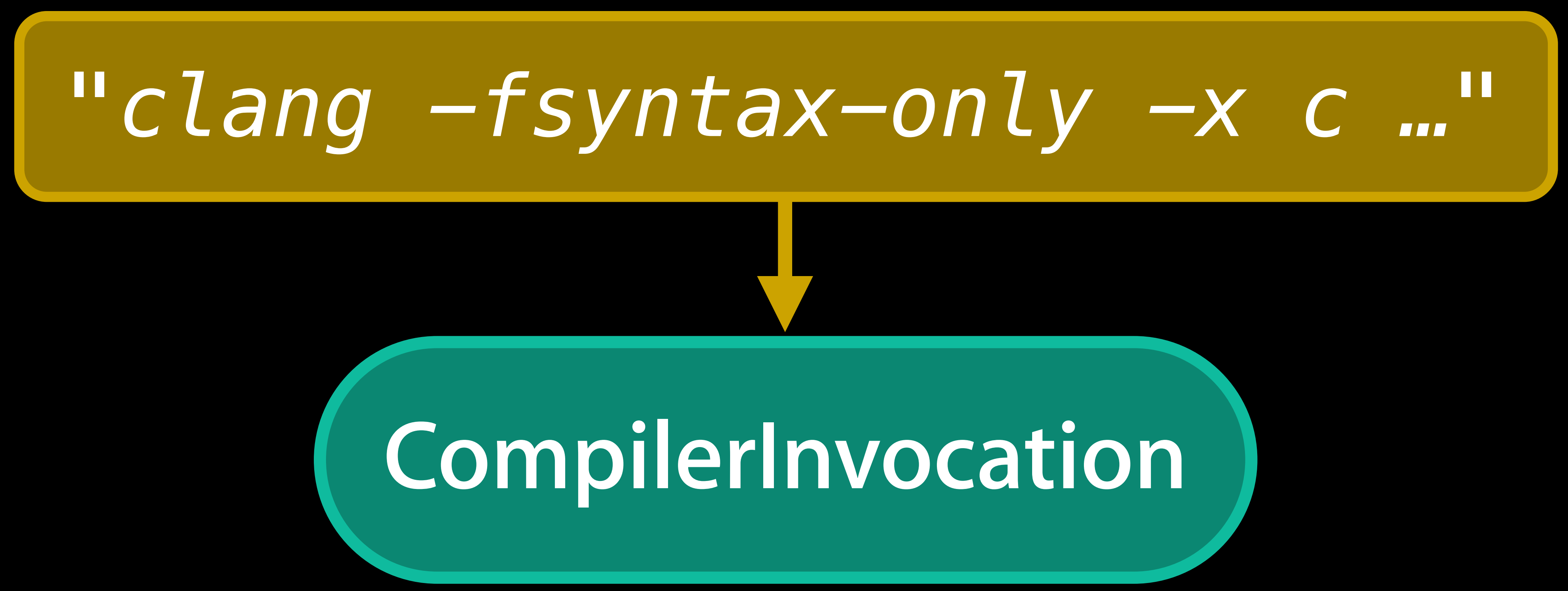
Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

"clang -fsyntax-only -x c …"

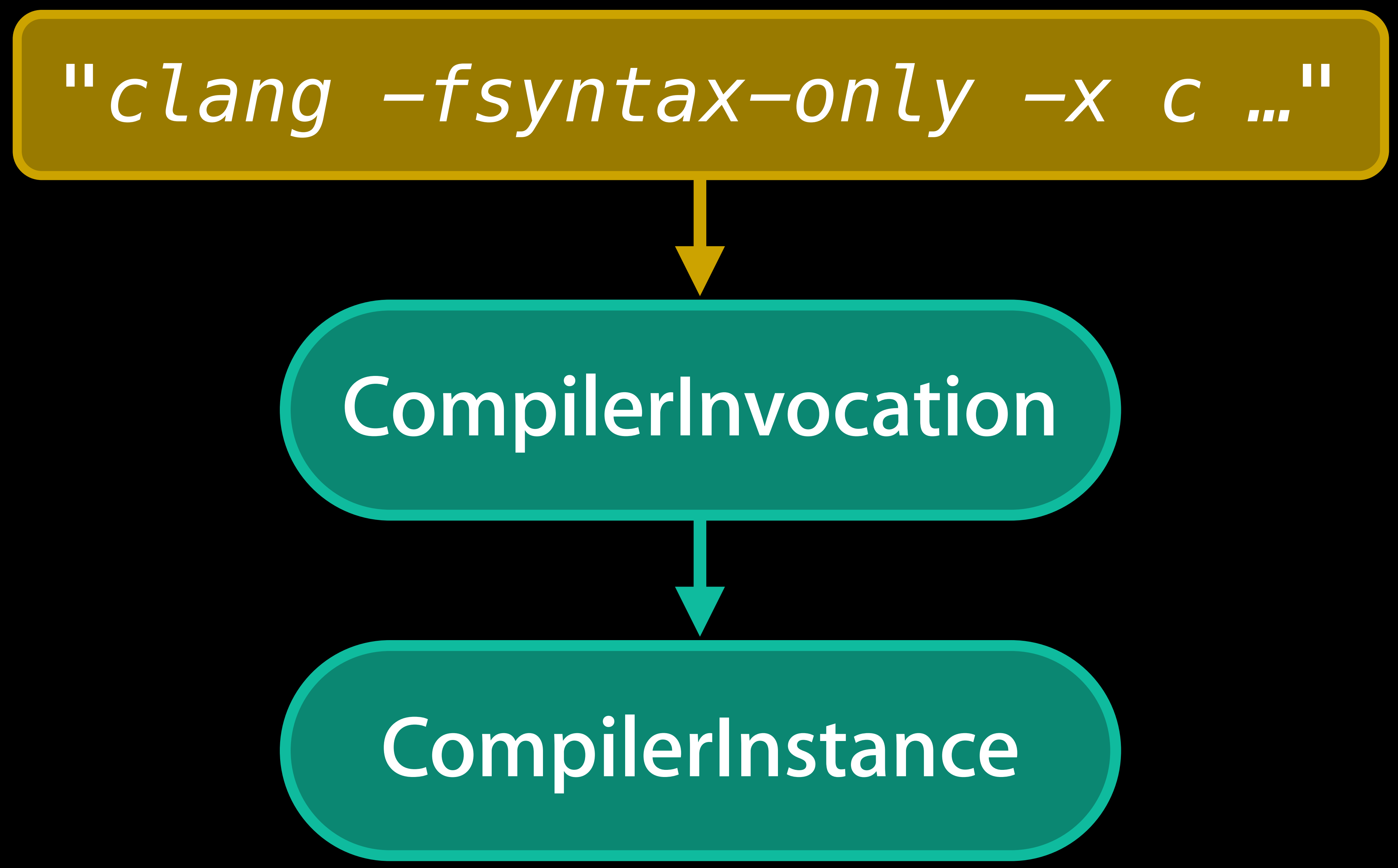
Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`



Setting up a clang::CompilerInstance

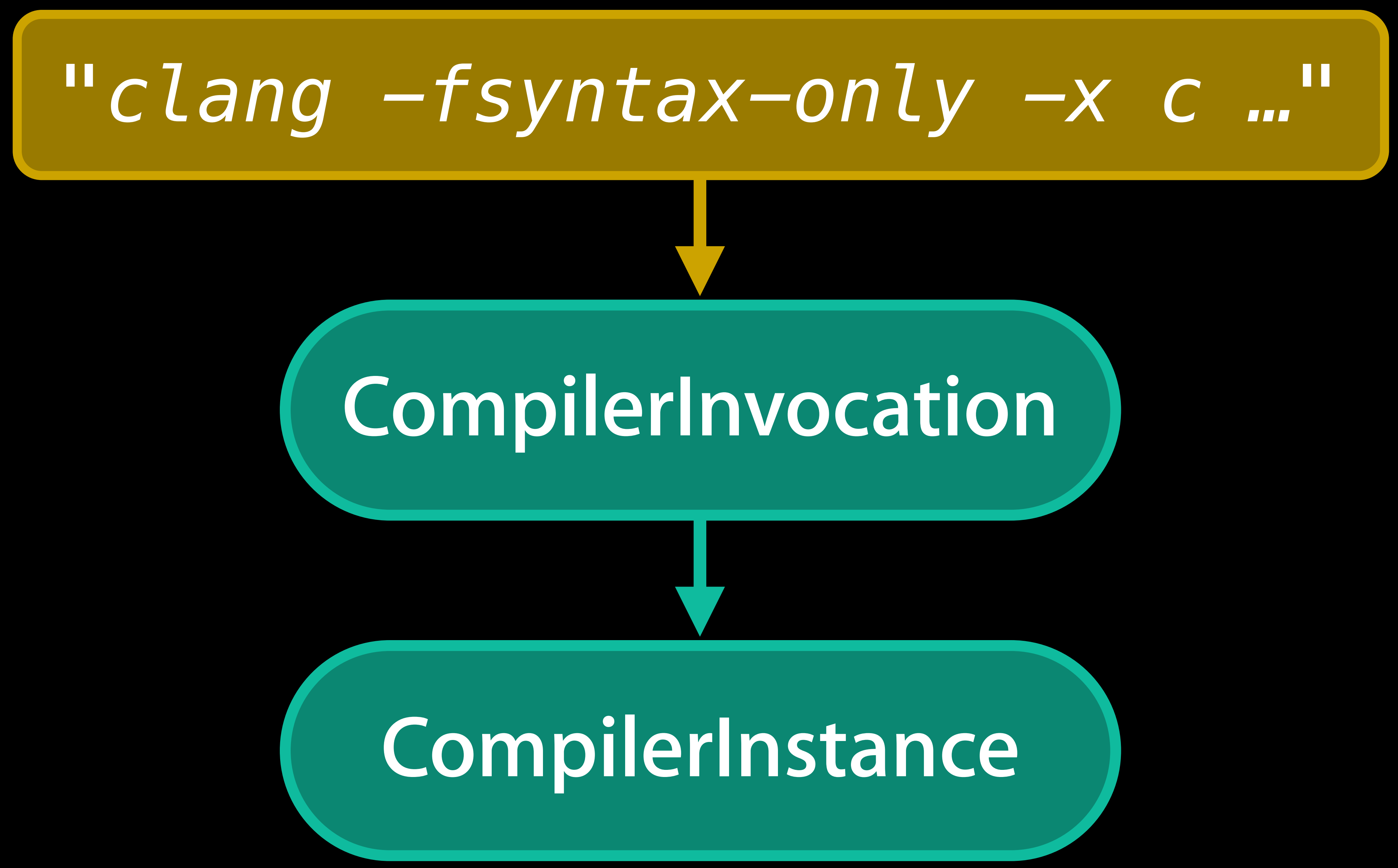
`createInvocationFromCommandLine()`



Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

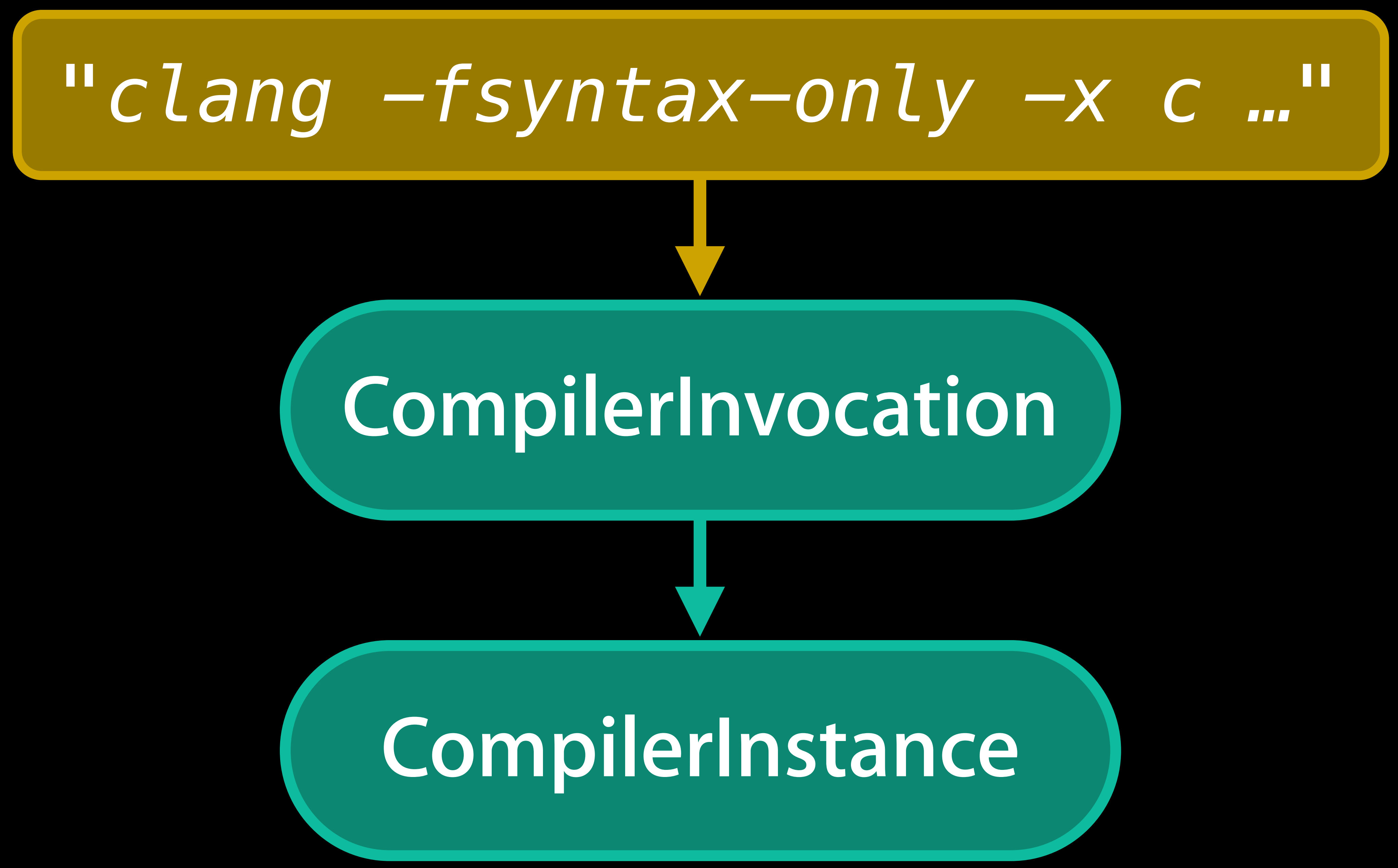


Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer

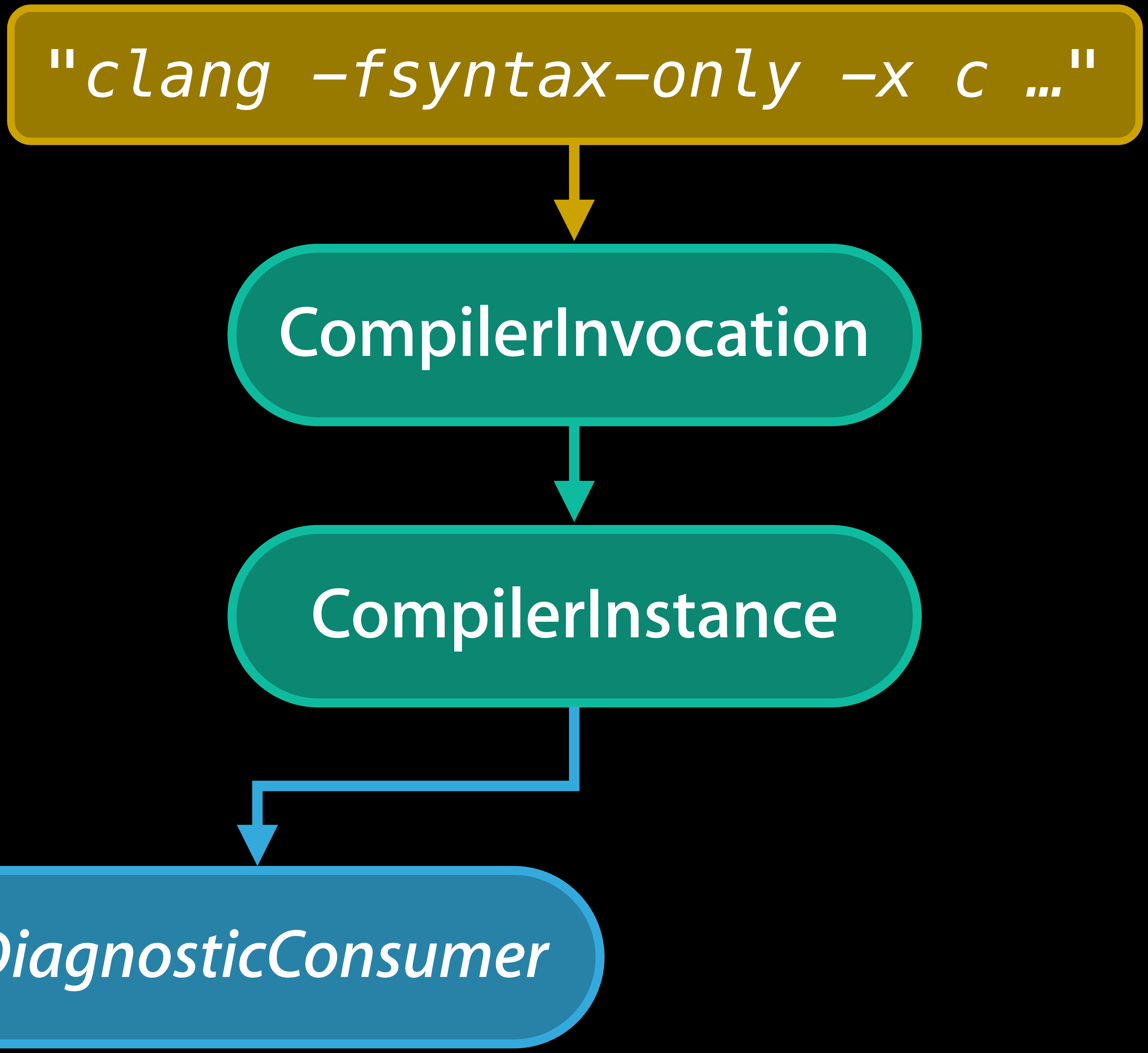


Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer

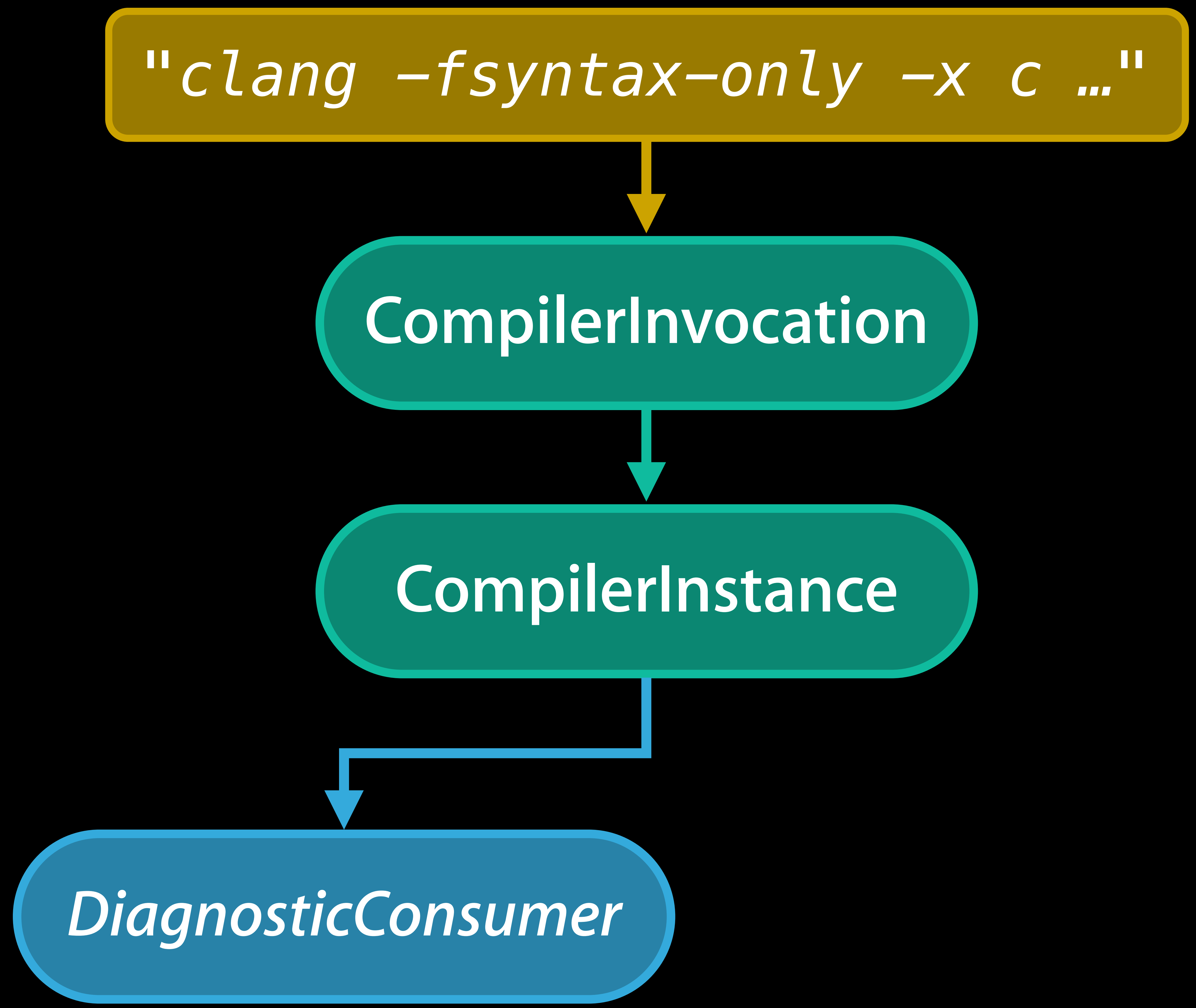


Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

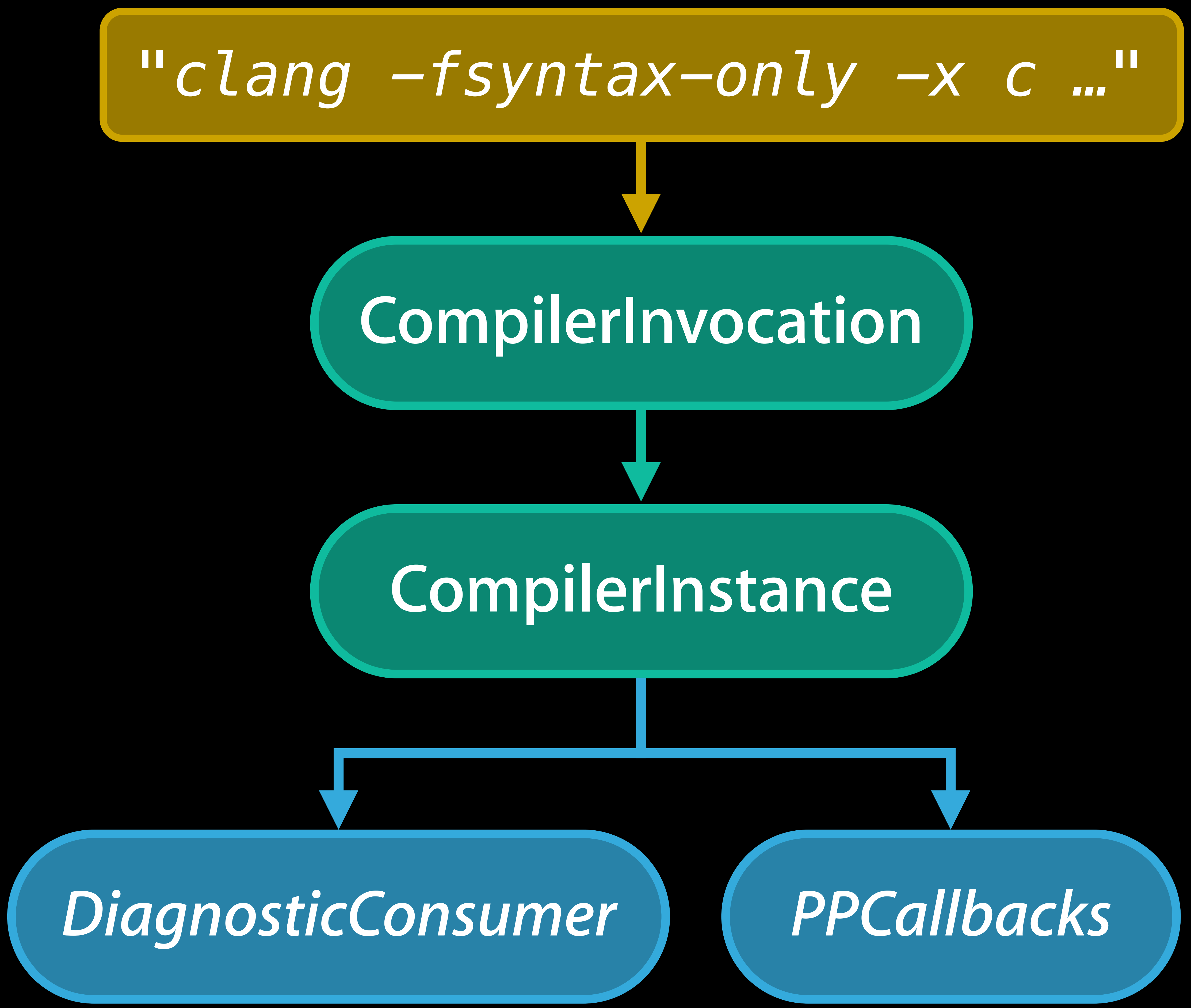


Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)



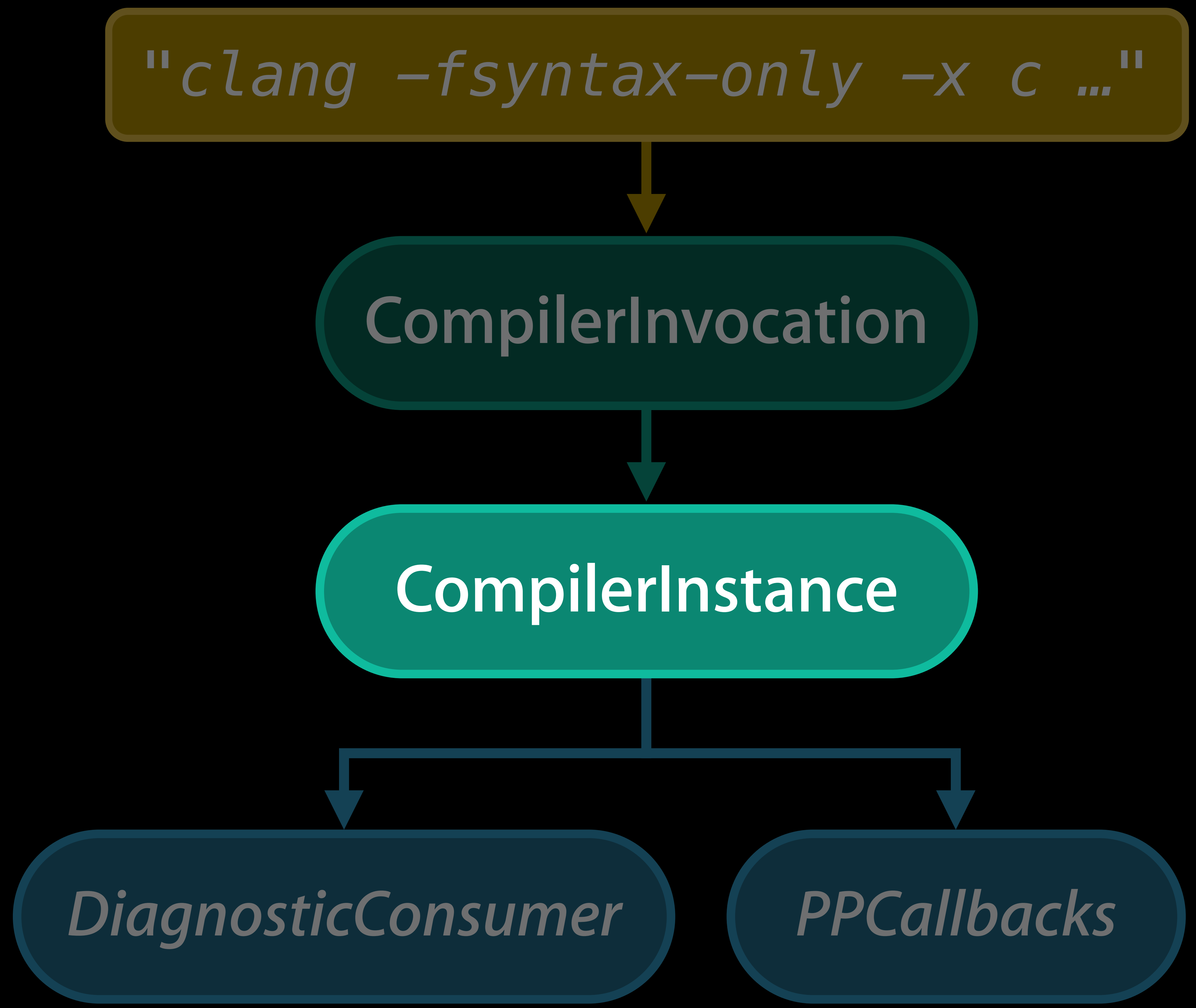
Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`



Setting up a `clang::CompilerInstance`

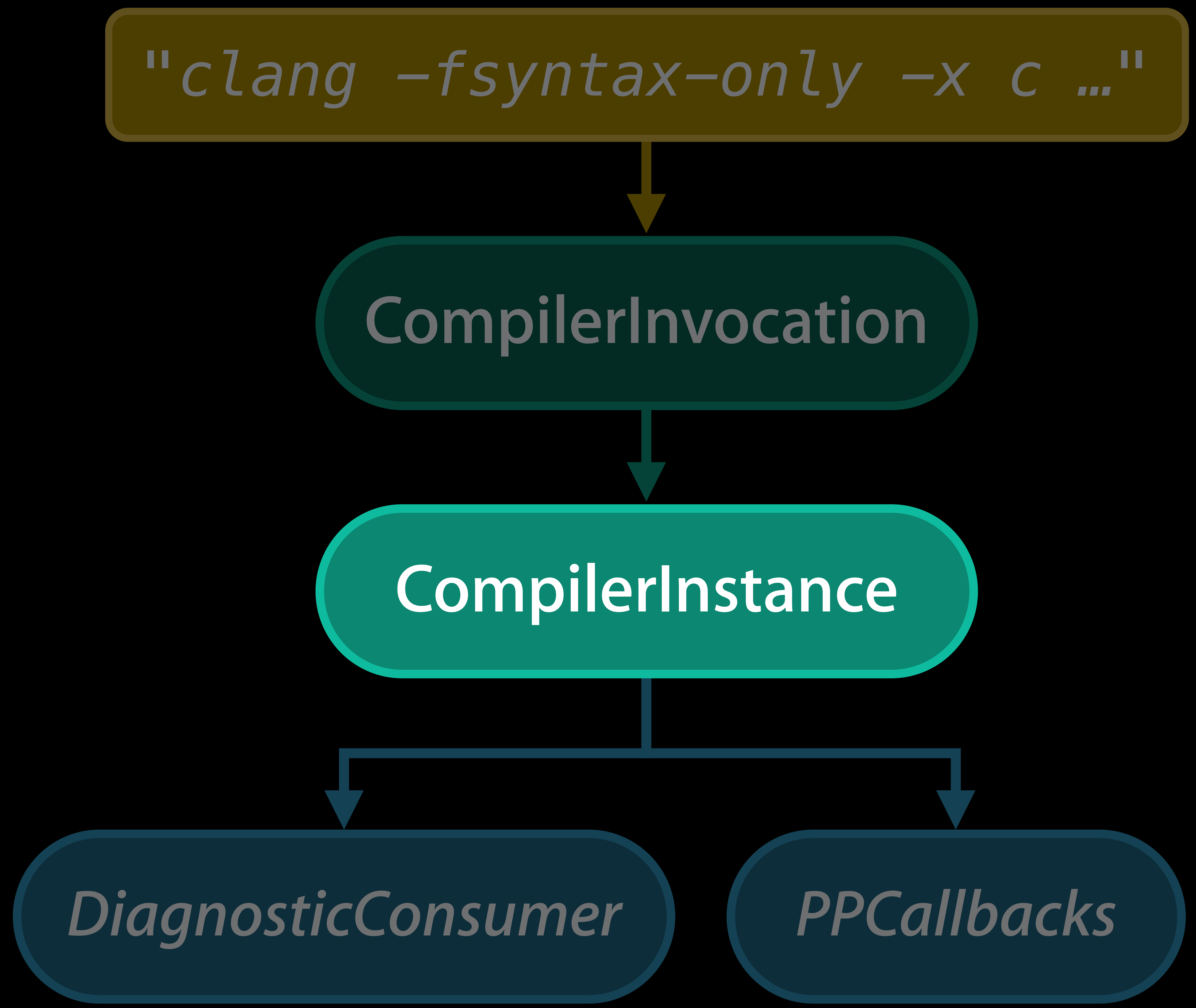
`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components



Setting up a `clang::CompilerInstance`

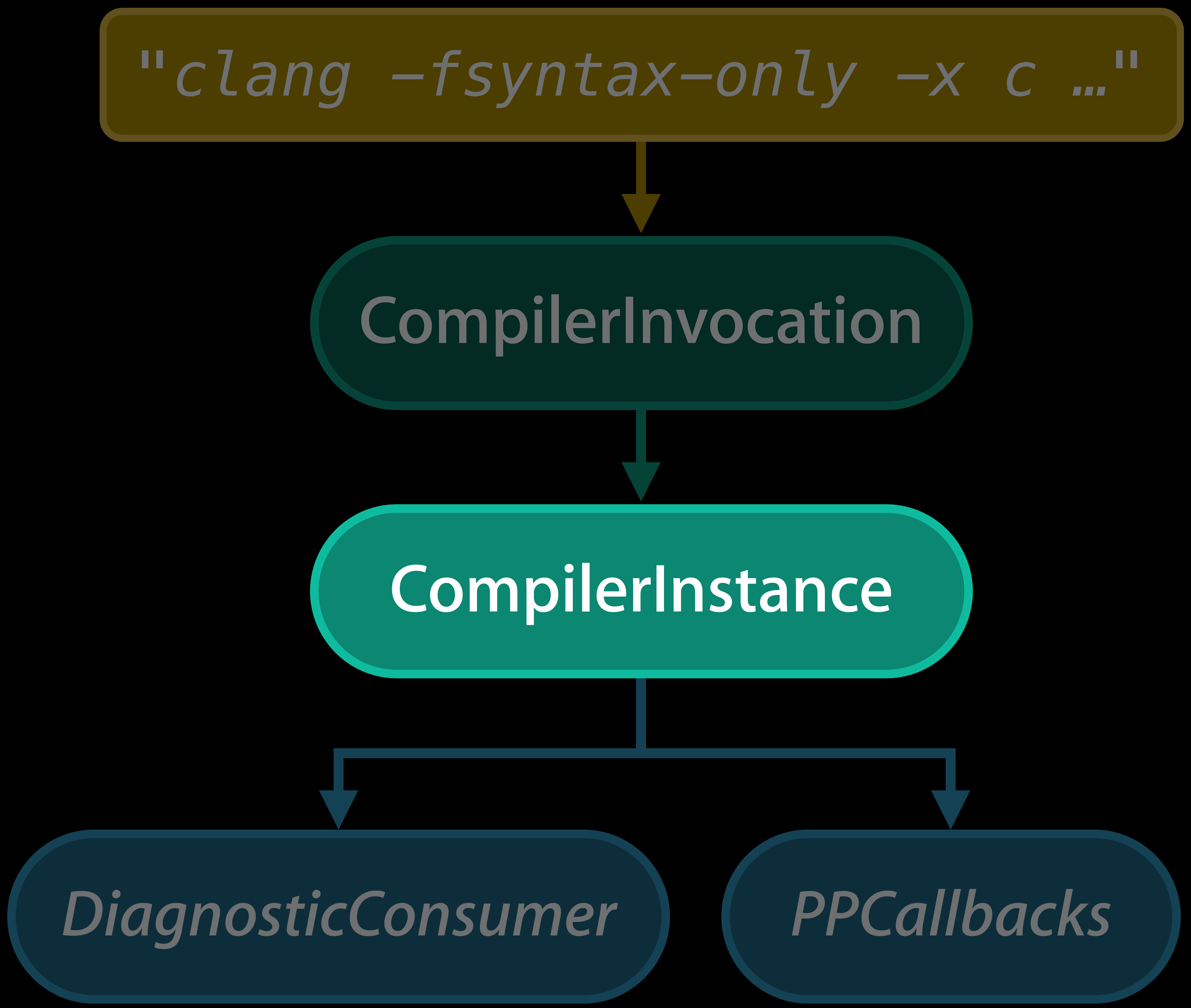
`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file



Setting up a `clang::CompilerInstance`

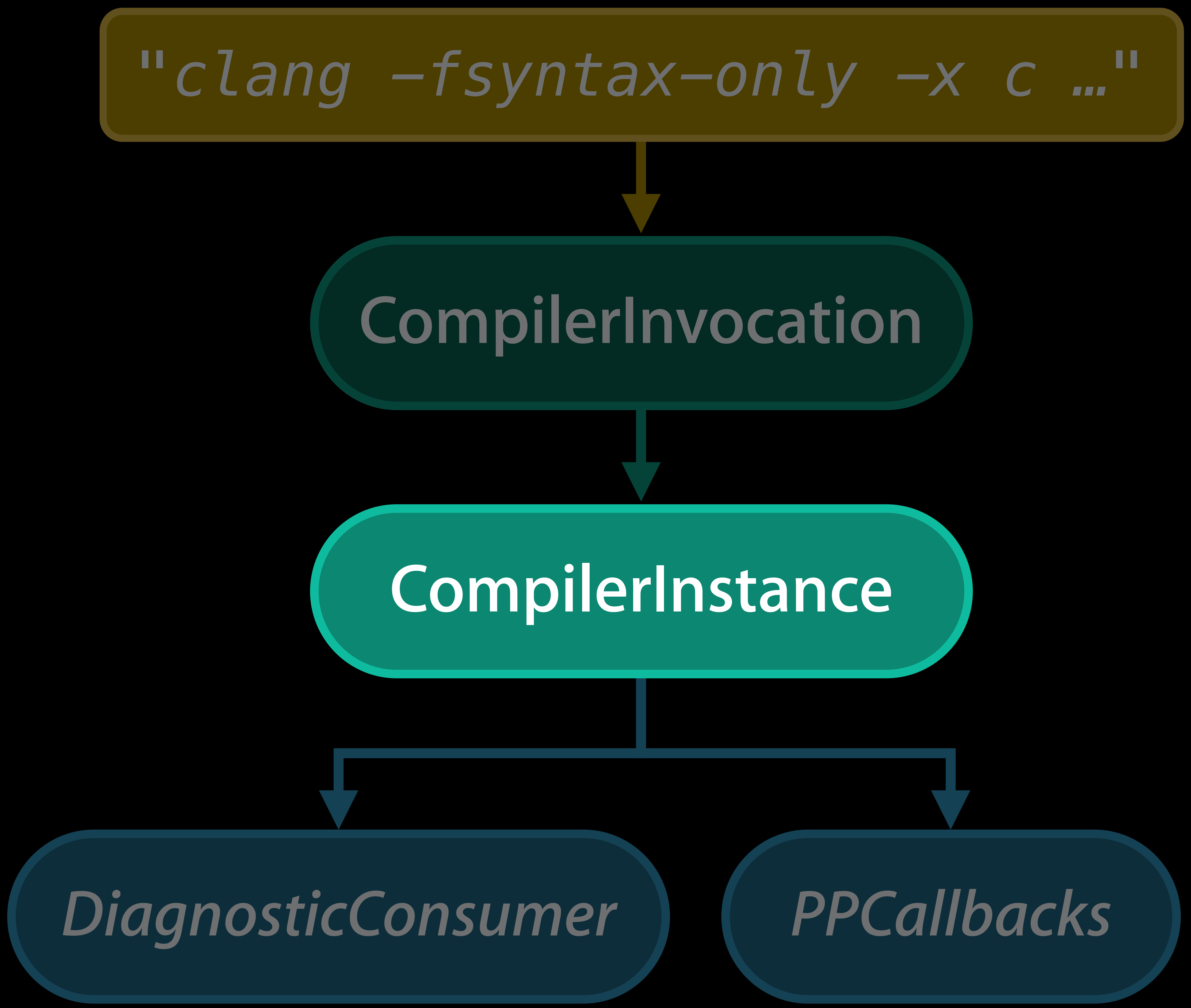
`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file
- Finalize the AST



Setting up a `clang::CompilerInstance`

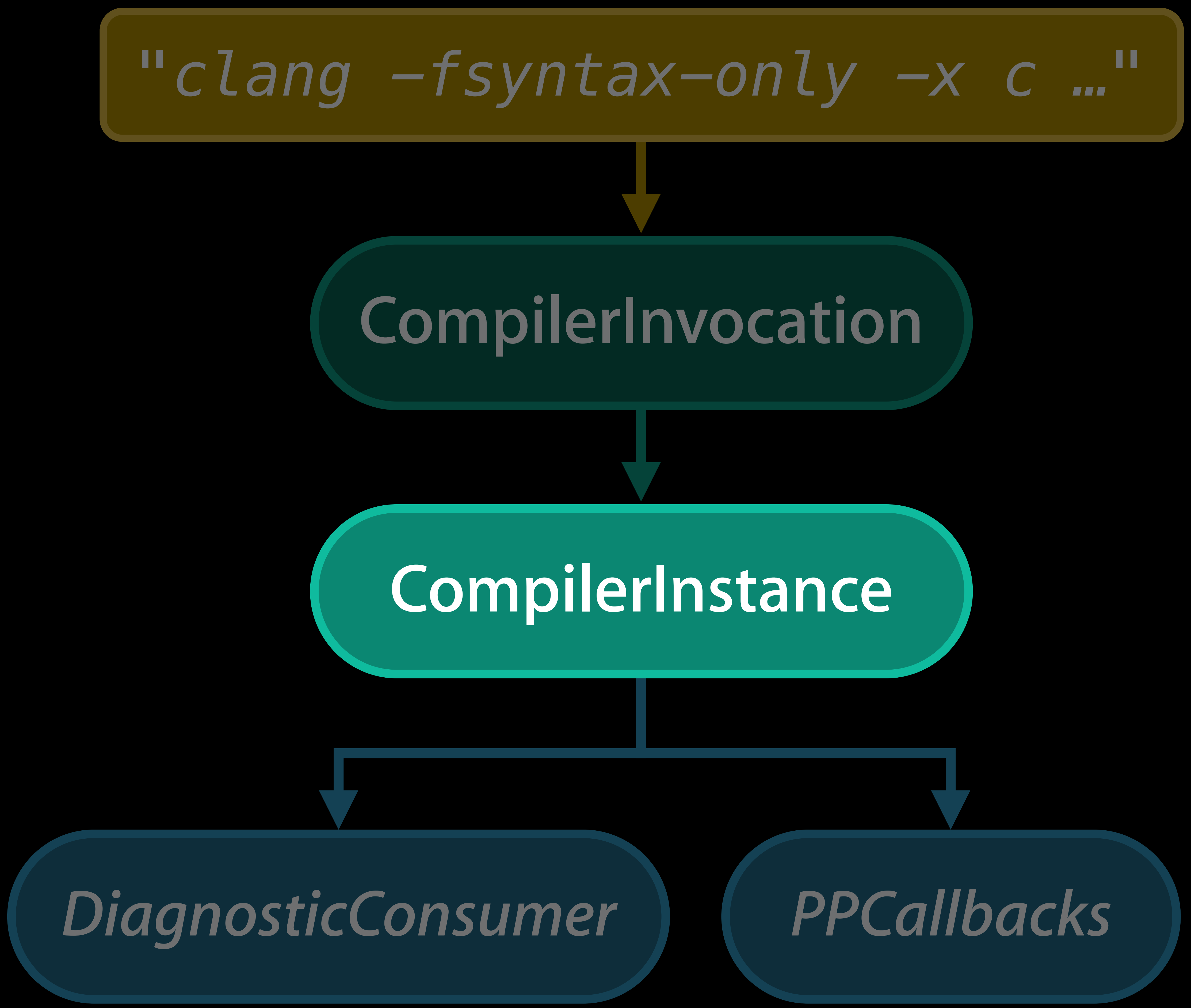
`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file
- ~~Finalize the AST~~



Setting up a `clang::CompilerInstance`

Actually works well

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file
- ~~Finalize the AST~~

Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file
- ~~Finalize the AST~~

A bit harder than it should be

Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Mostly okay

Manually run most of `ExecuteAction()`

- Set up several compiler components
- Parse a single decl from a dummy file
- ~~Finalize the AST~~

Setting up a `clang::CompilerInstance`

`createInvocationFromCommandLine()`

Attach custom observers

- Diagnostic consumer
- PP callbacks (for module import)

Manually run most of `ExecuteAction()`

sadness

- Set up several compiler components
- Parse a single decl from a dummy file
- ~~Finalize the AST~~

(this is really the only reason)

Clang Modules

Clang Modules

Self-contained units of API

Clang Modules

Self-contained units of API

- No cross-header pollution!

Clang Modules

Self-contained units of API

- No cross-header pollution!

Separate semantics from syntax

Clang Modules

Self-contained units of API

- No cross-header pollution!

Separate semantics from syntax

- Same mechanism as PCH

Clang Modules

Self-contained units of API

- No cross-header pollution!

Separate semantics from syntax

- Same mechanism as PCH

Importing Clang Modules

Importing Clang Modules

`CompilerInstance::loadModule`

Importing Clang Modules

`CompilerInstance::loadModule`

Geometry

```
typedef … Point2f;
Point2f flipOverXAxis(…);
Point2f flipOverYAxis(…);
void drawGraph(…);
…
```

Importing Clang Modules

`CompilerInstance::loadModule`

Look up the decls we want

Geometry

```
typedef … Point2f;
Point2f flipOverXAxis(…);
Point2f flipOverYAxis(…);
void drawGraph(…);
…
```

Importing Clang Modules

`CompilerInstance::loadModule`

Look up the decls we want

Geometry

```
typedef … Point2f;
Point2f flipOverXAxis(…);
Point2f flipOverYAxis(…);
void drawGraph(…);
…
```

```
flipOverXAxis(originalPoint)
```

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

Geometry

```
typedef … Point2f;
Point2f flipOverXAxis(…);
Point2f flipOverYAxis(…);
void drawGraph(…);
…
```

```
flipOverXAxis(originalPoint)
```

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

- Use TU-wide lookup and filter

Geometry

```
typedef … Point2f;
Point2f flipOverXAxis(…);
Point2f flipOverYAxis(…);
void drawGraph(…);
…
```

```
flipOverXAxis(originalPoint)
```

Importing Clang Modules

`CompilerInstance::loadModule`

Requires a SourceLocation
Awkward for submodules

~~Look up the decls we want~~

- Use TU-wide lookup and filter

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

- Use TU-wide lookup and filter

Definitely something to improve

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

- Use TU-wide lookup and filter

What if two modules conflict?

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

- Use TU-wide lookup and filter

What if two modules conflict?

OldLibrary

```
…
typedef unsigned status_t;
…
```

Importing Clang Modules

`CompilerInstance::loadModule`

~~Look up the decls we want~~

- Use TU-wide lookup and filter

What if two modules conflict?

OldLibrary

```
…
typedef unsigned status_t;
…
```

NewLibrary

```
…
typedef enum {…} status_t;
…
```

Importing Declarations

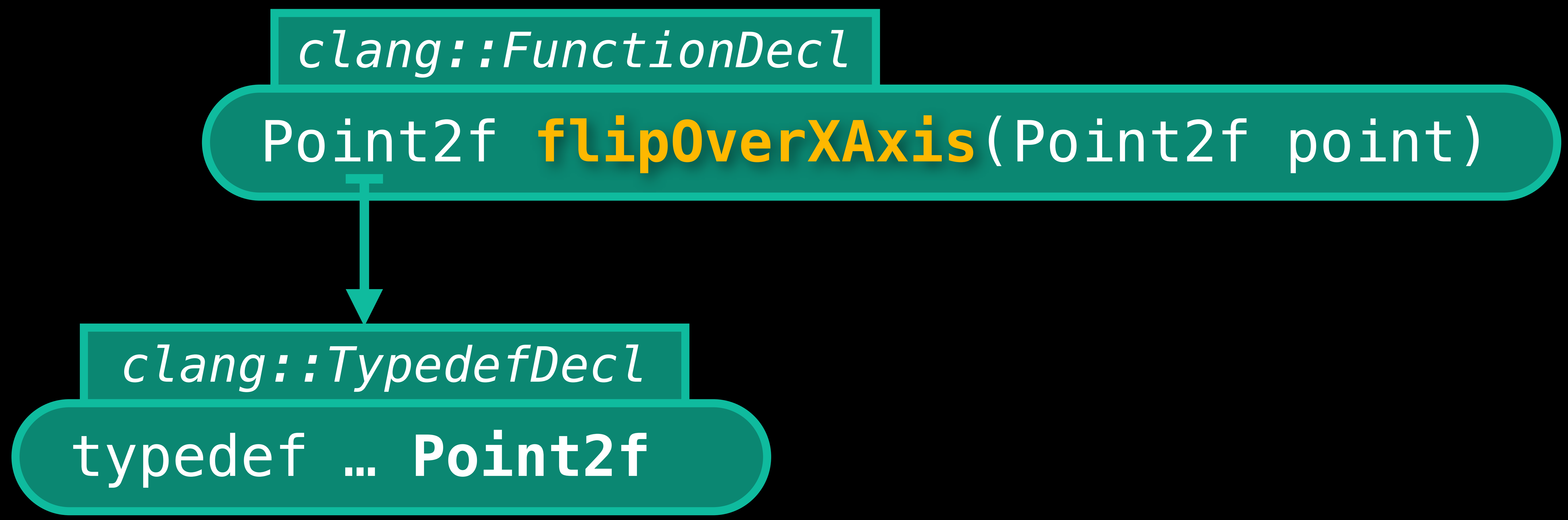
```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}
```

Importing Declarations

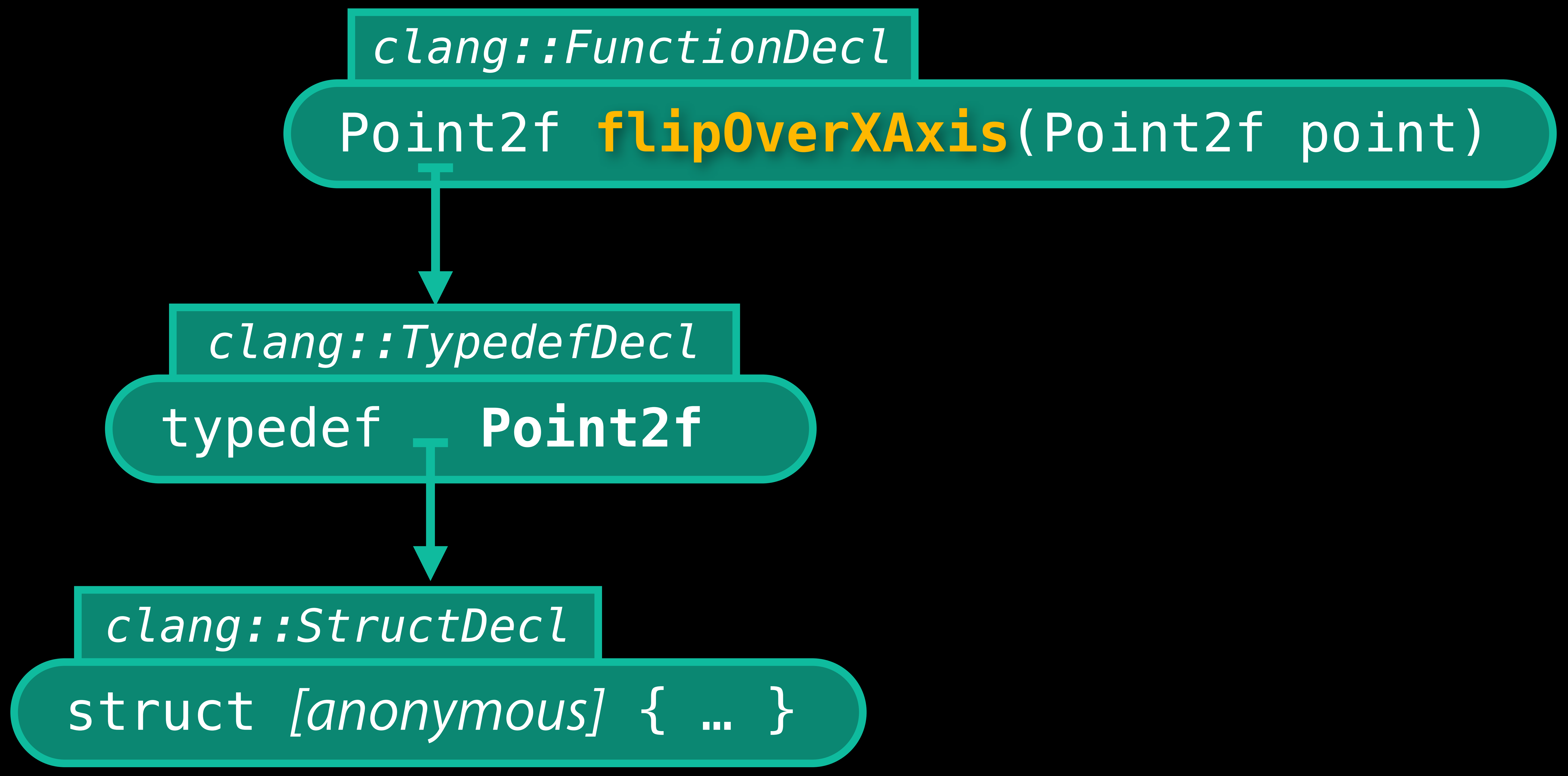
`clang::FunctionDecl`

```
Point2f flipOverXAxis(Point2f point)
```

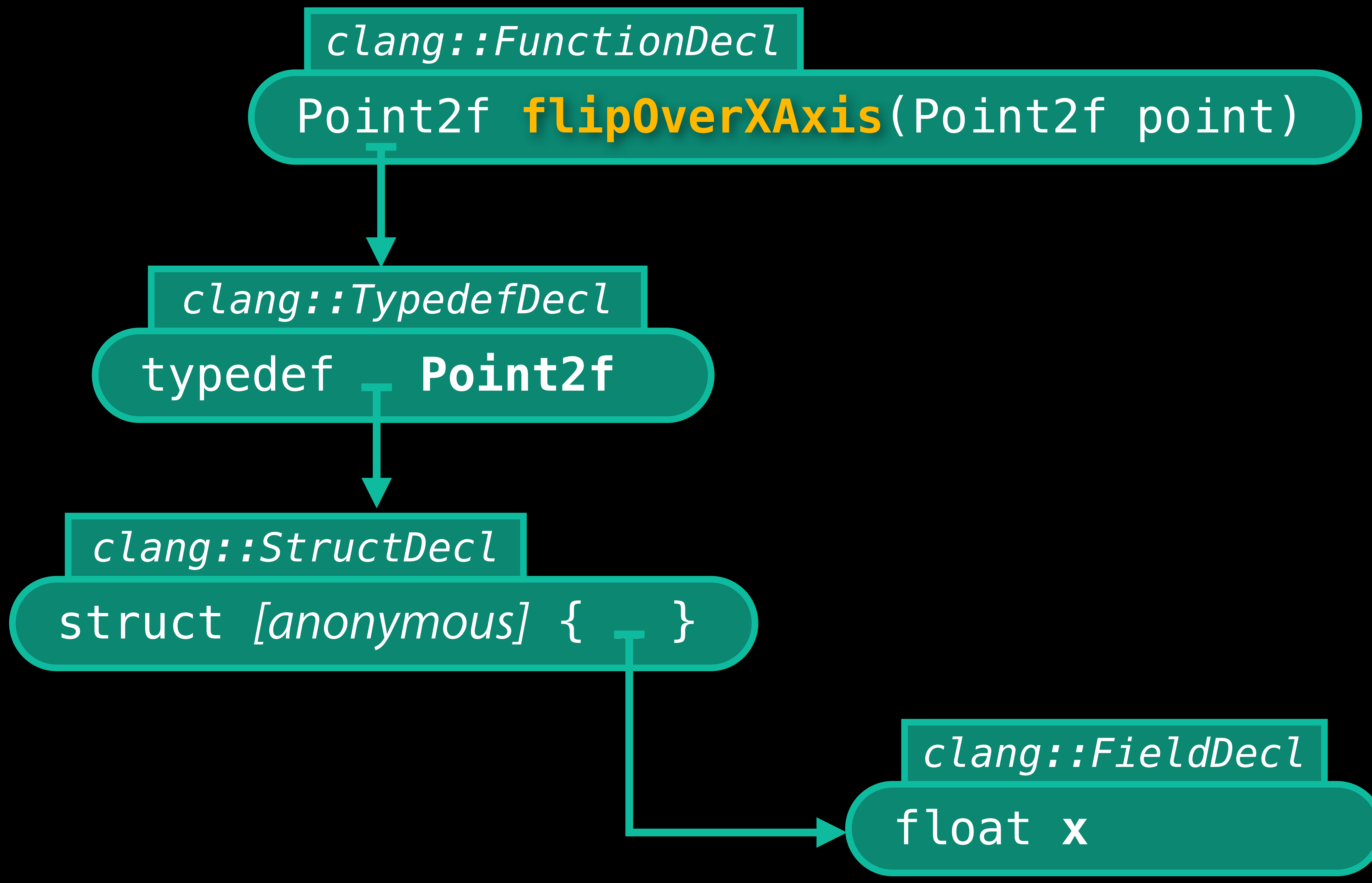
Importing Declarations



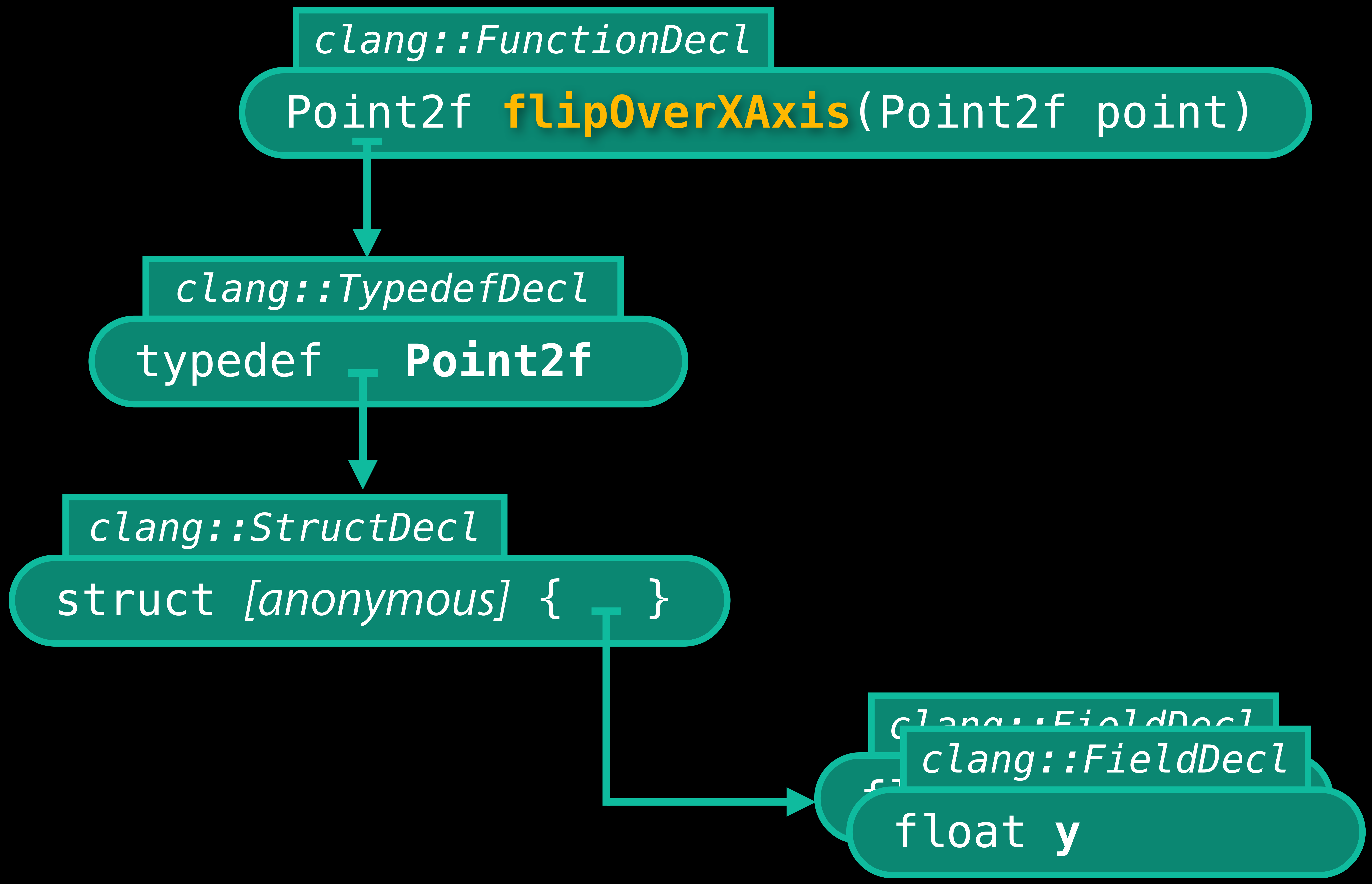
Importing Declarations



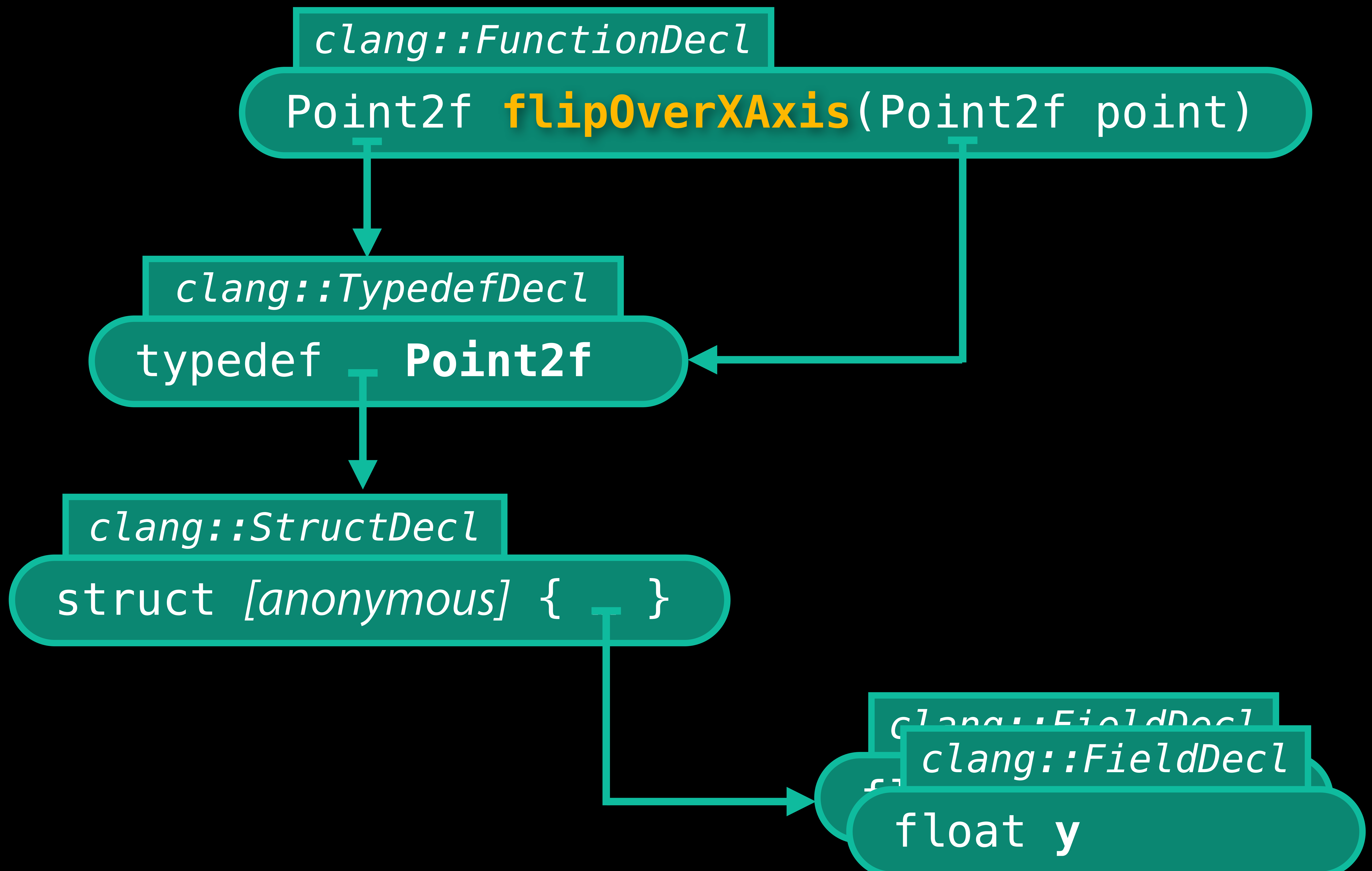
Importing Declarations



Importing Declarations

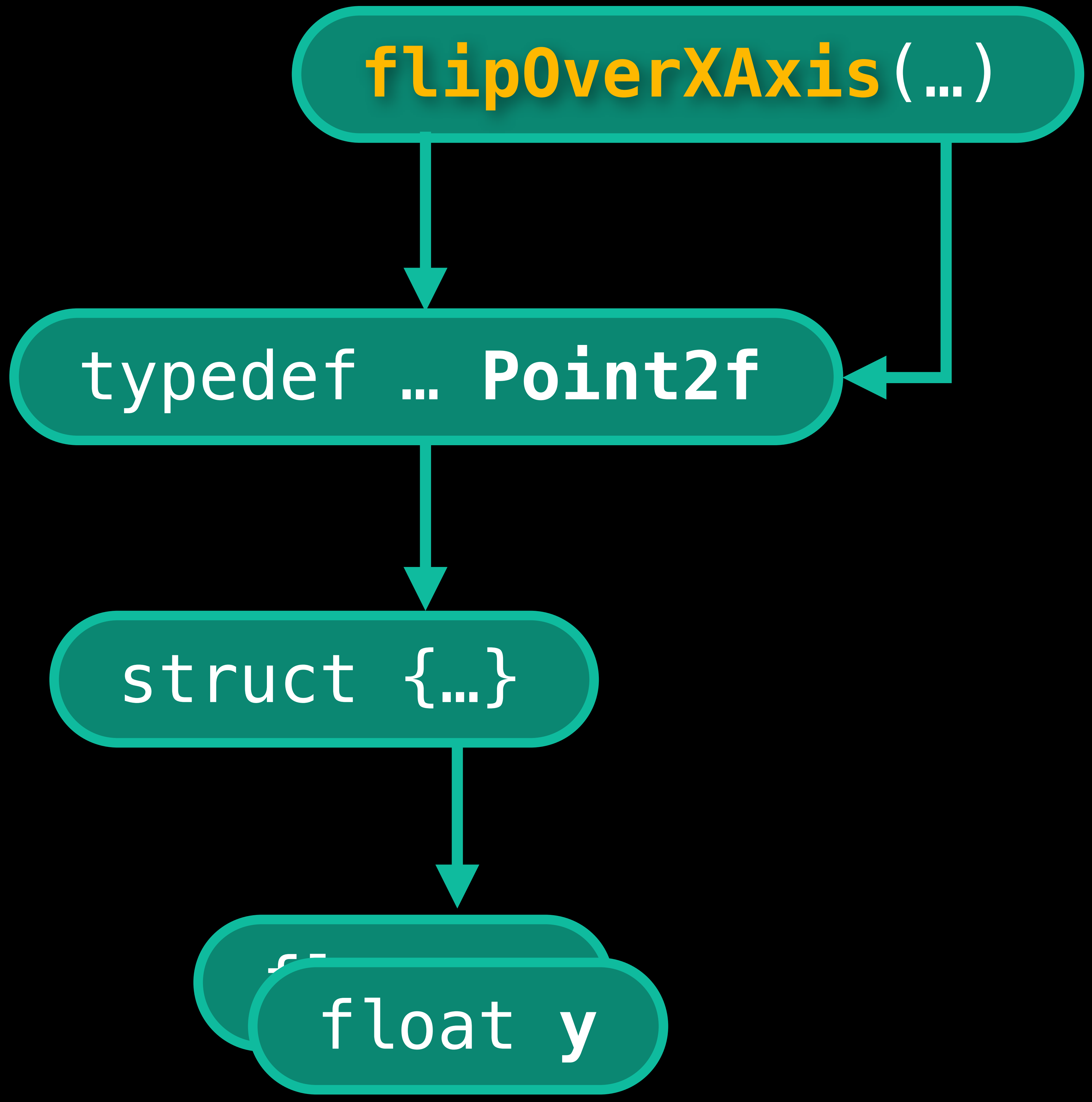


Importing Declarations



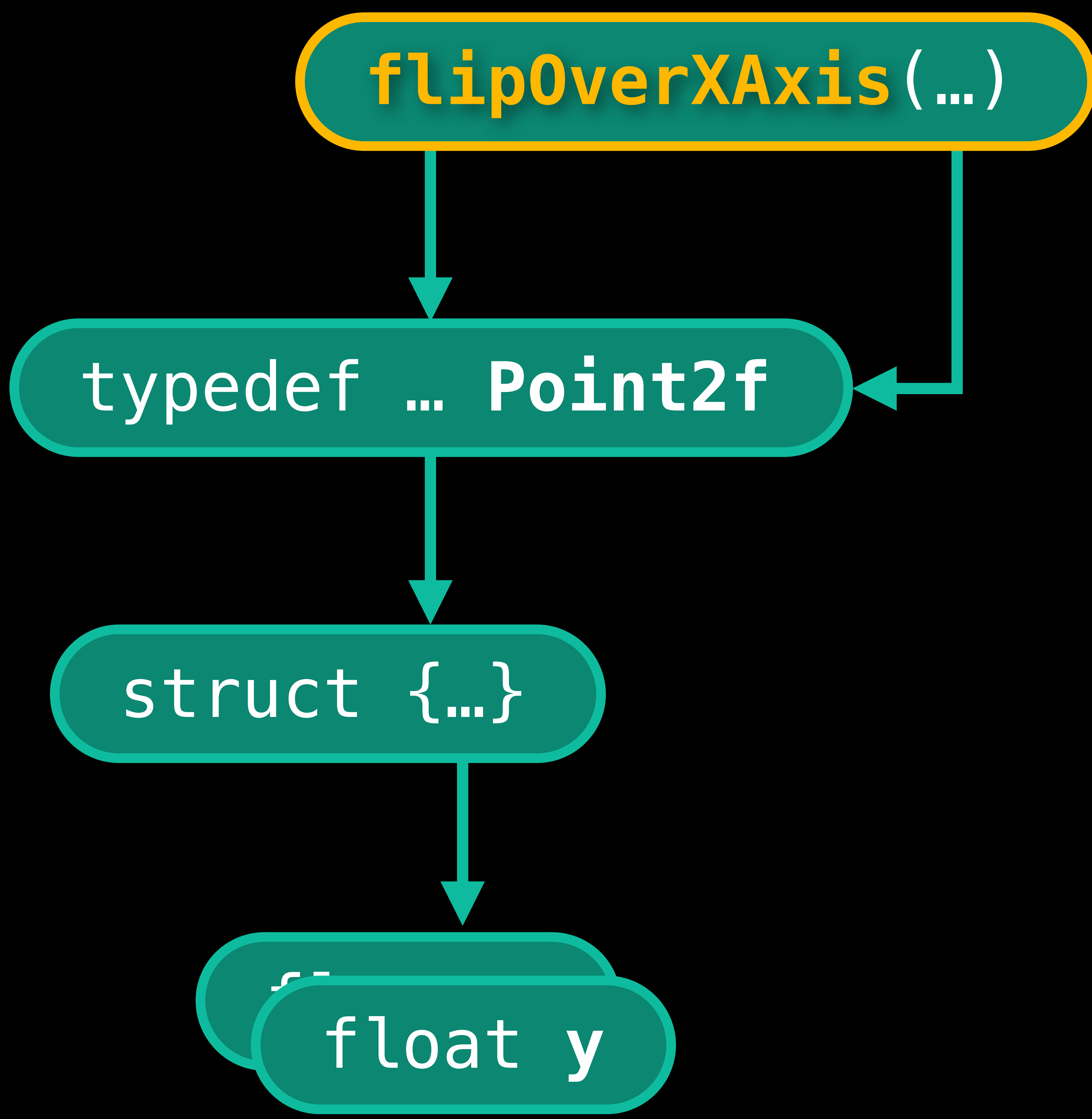
Importing Declarations

...using clang::ASTVisitor



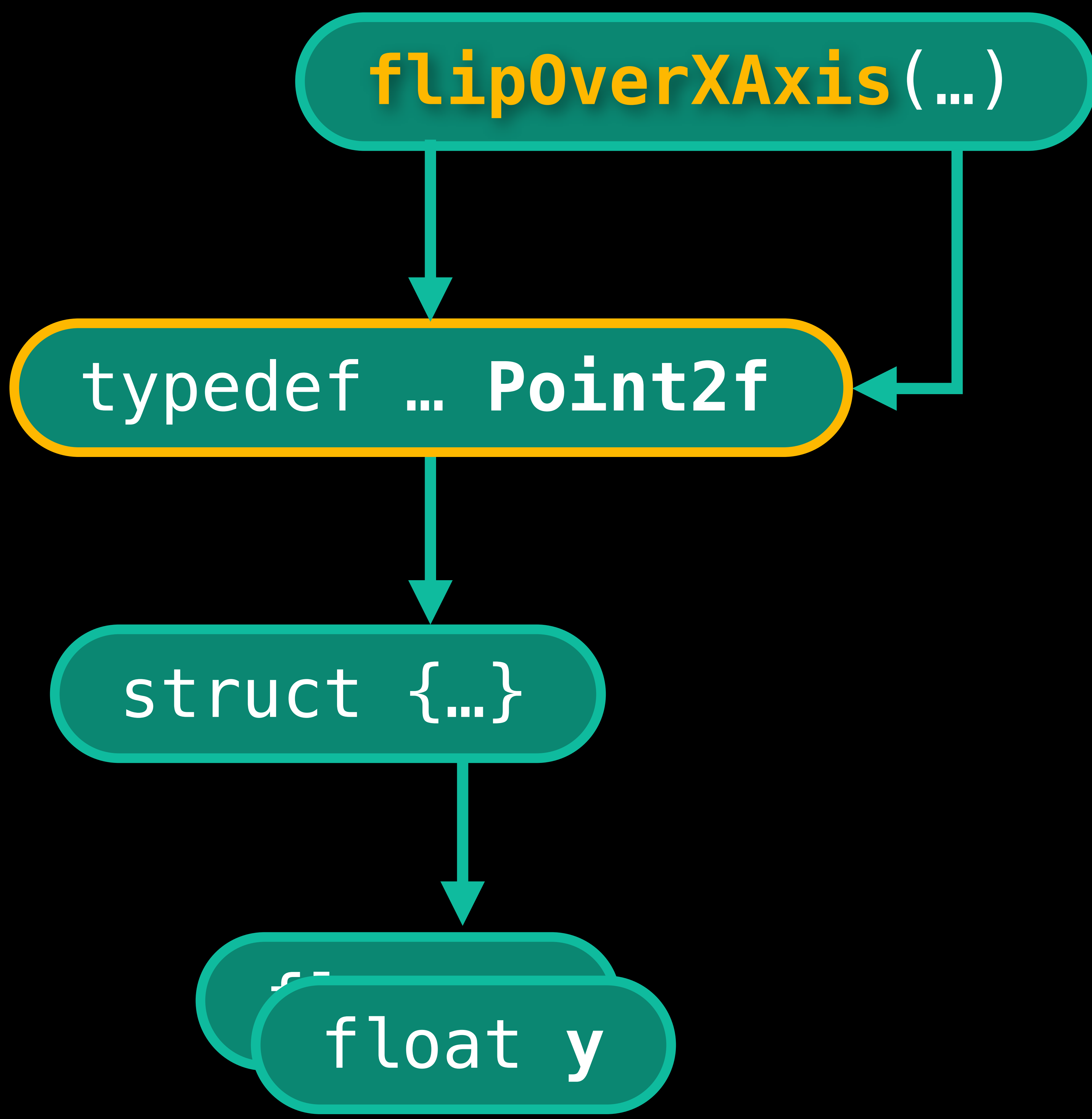
Importing Declarations

...using `clang::ASTVisitor`



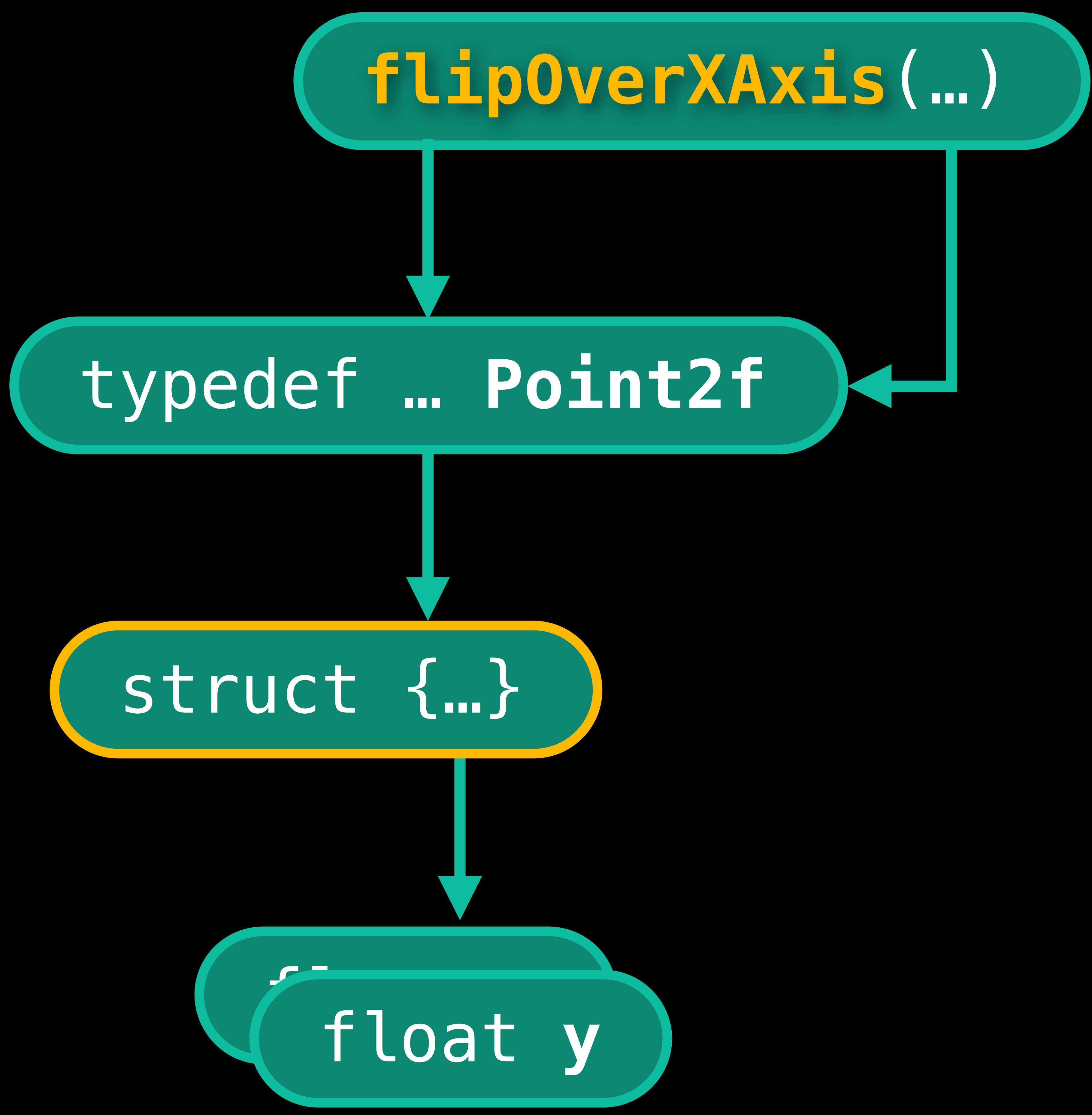
Importing Declarations

...using clang::ASTVisitor



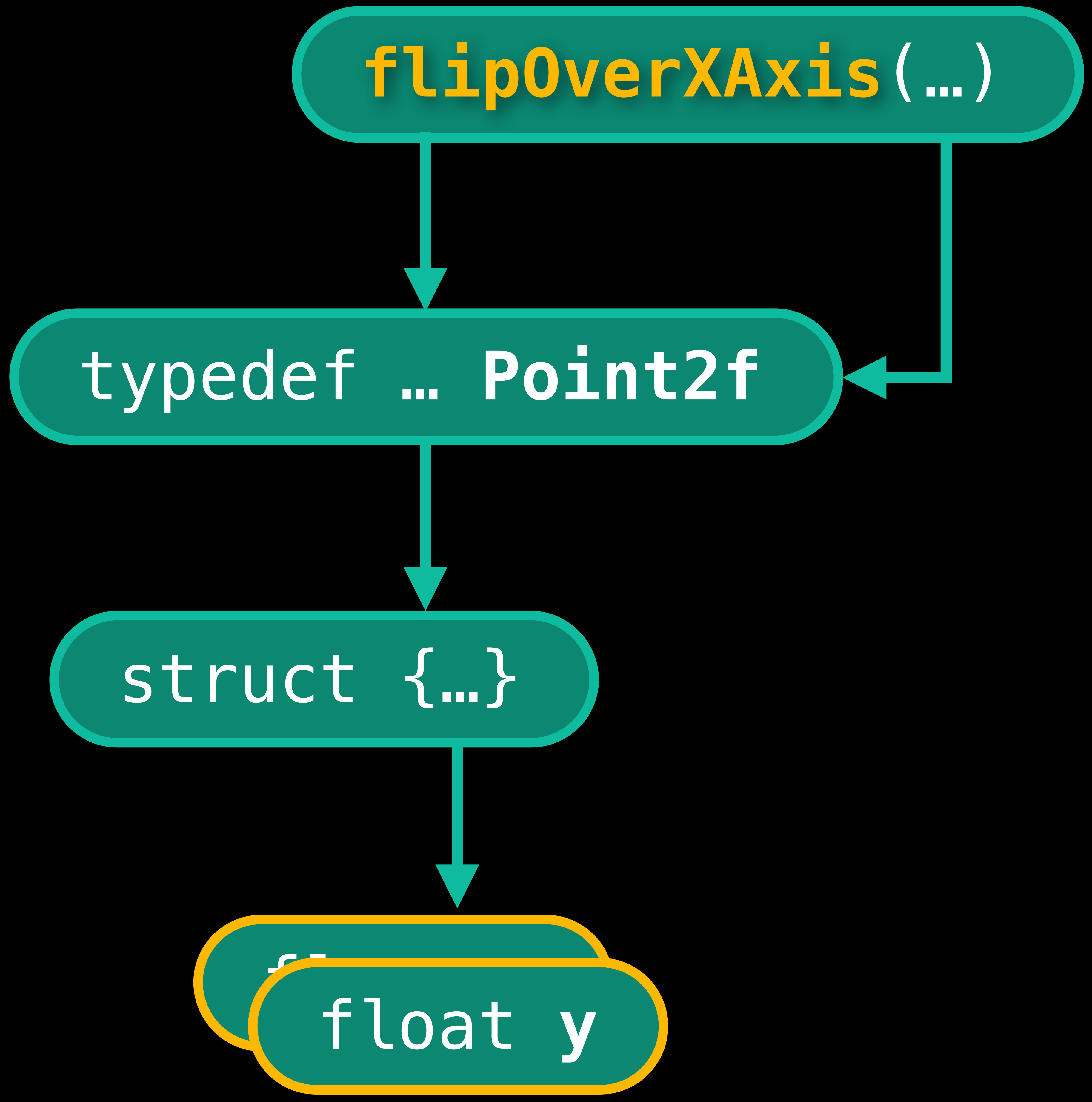
Importing Declarations

...using `clang::ASTVisitor`



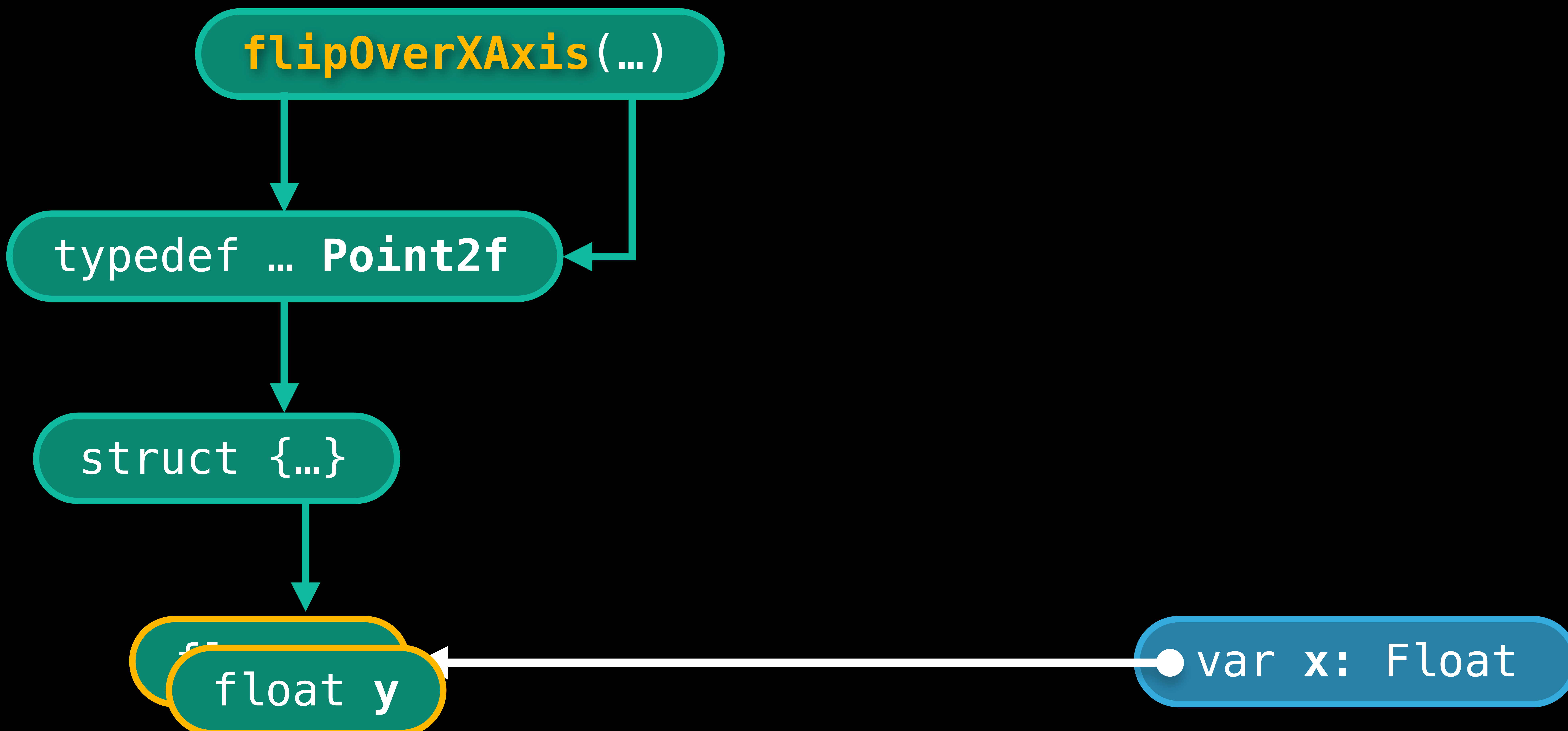
Importing Declarations

...using clang::ASTVisitor



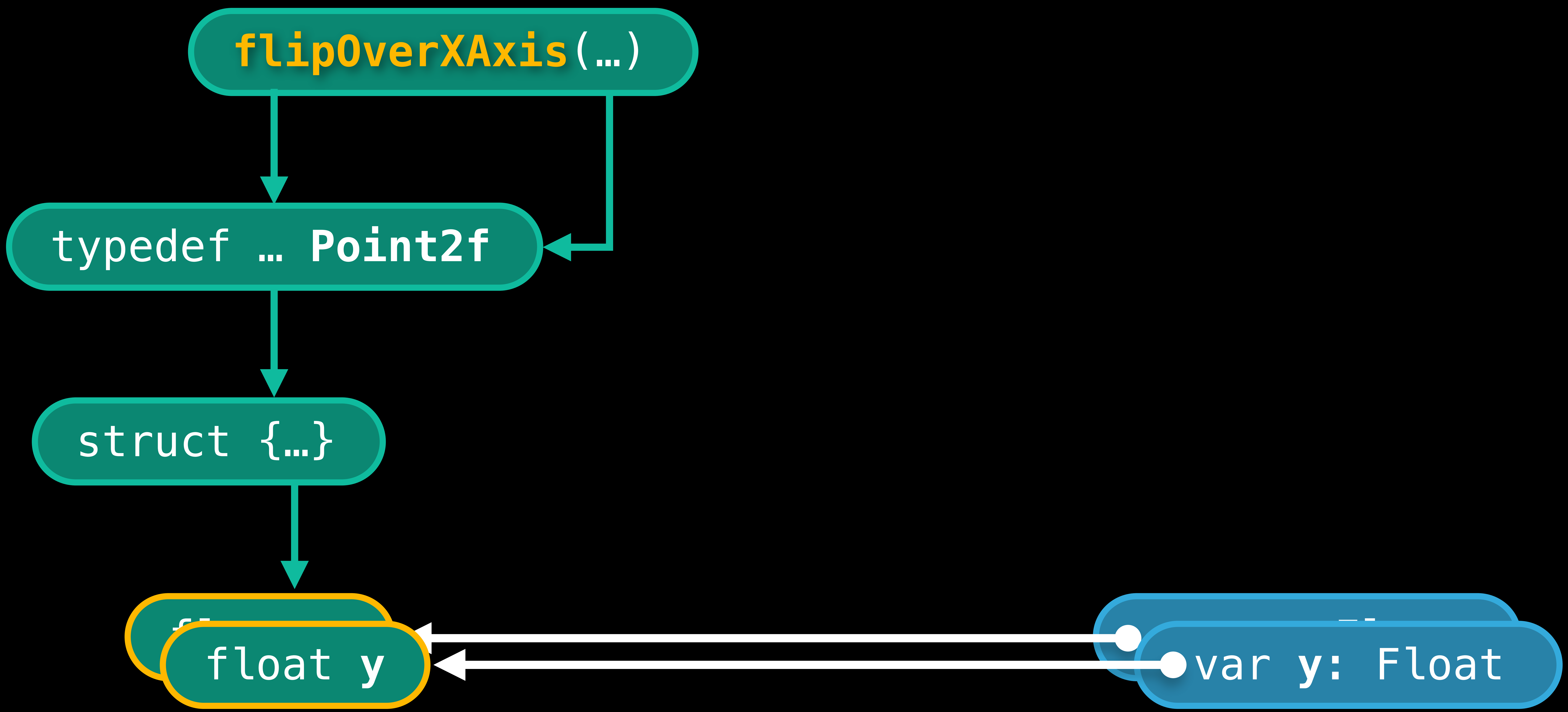
Importing Declarations

...using `clang::ASTVisitor`



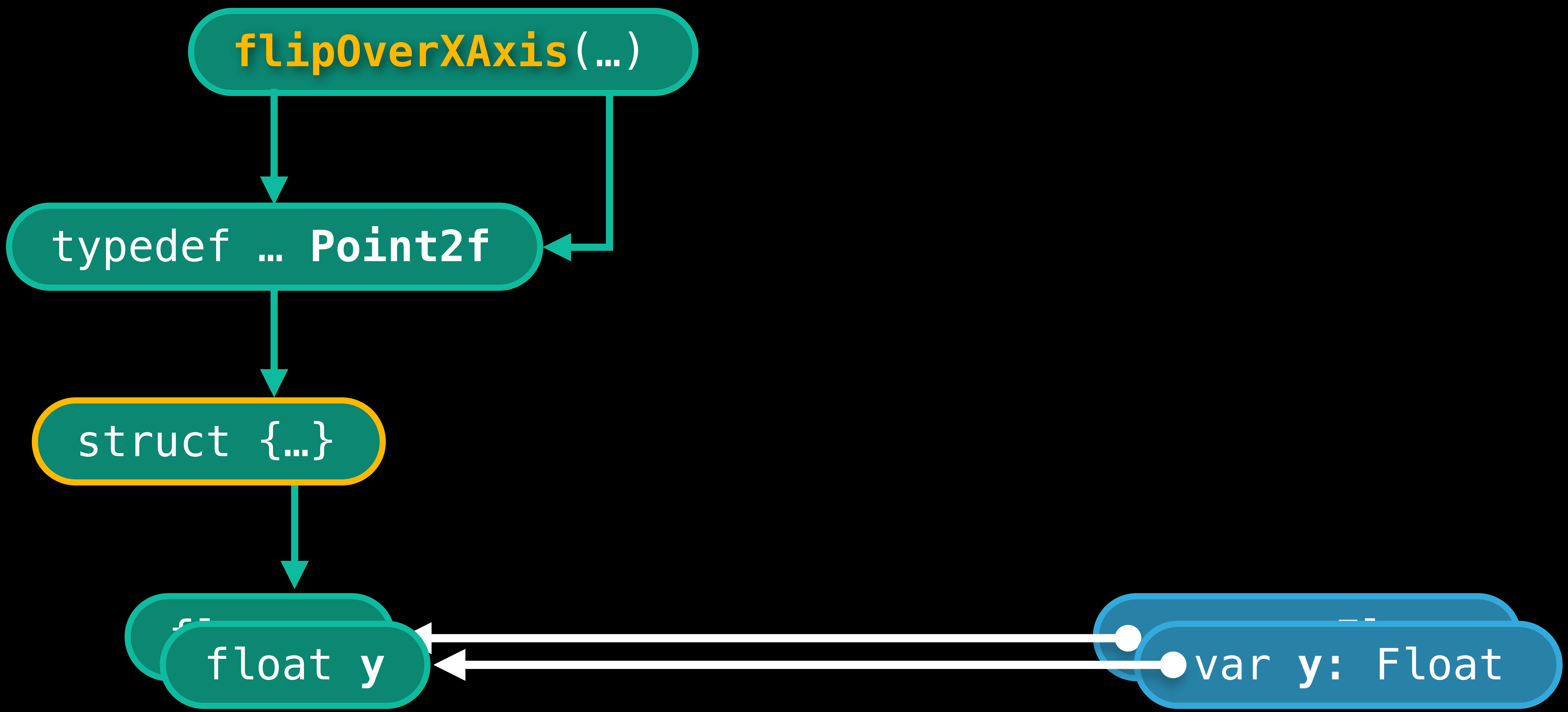
Importing Declarations

...using `clang::ASTVisitor`



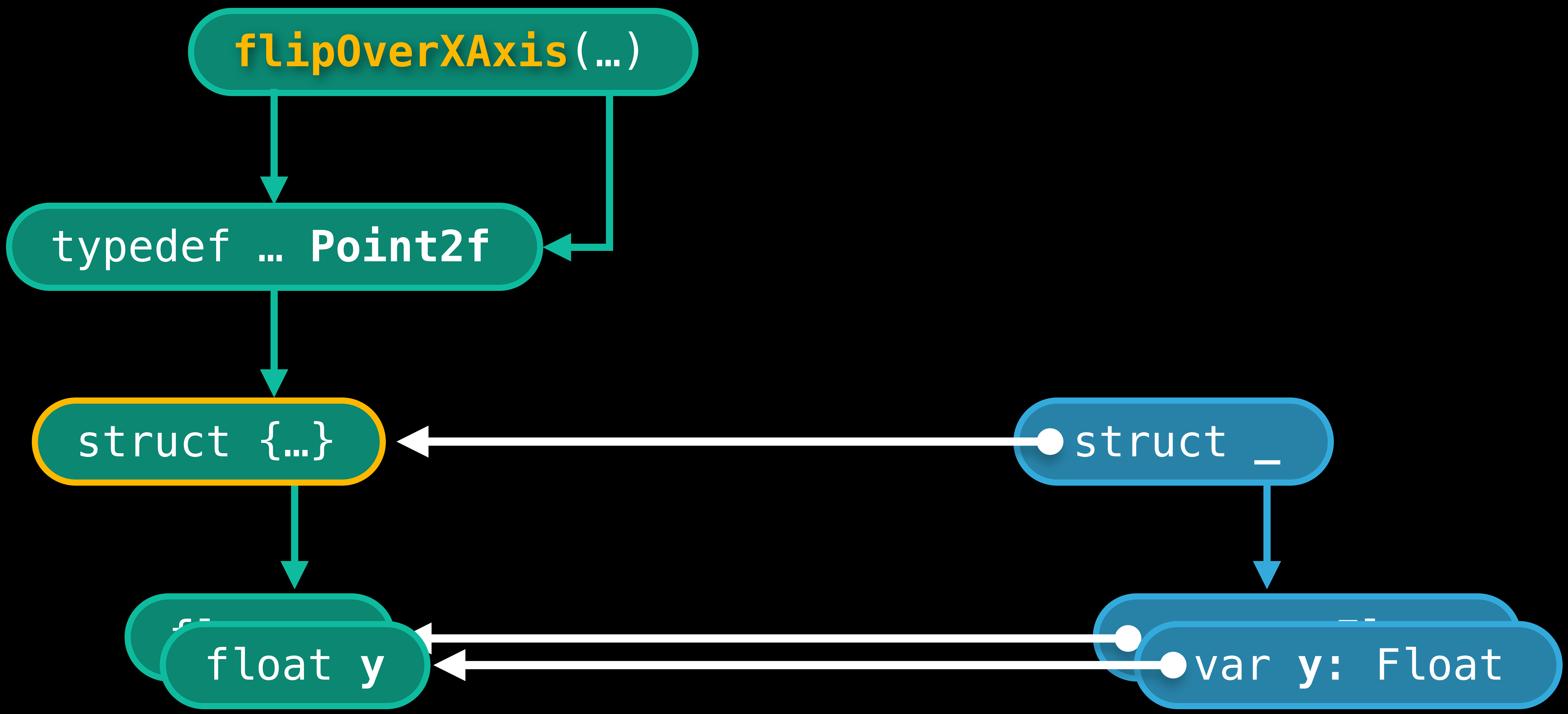
Importing Declarations

...using `clang::ASTVisitor`



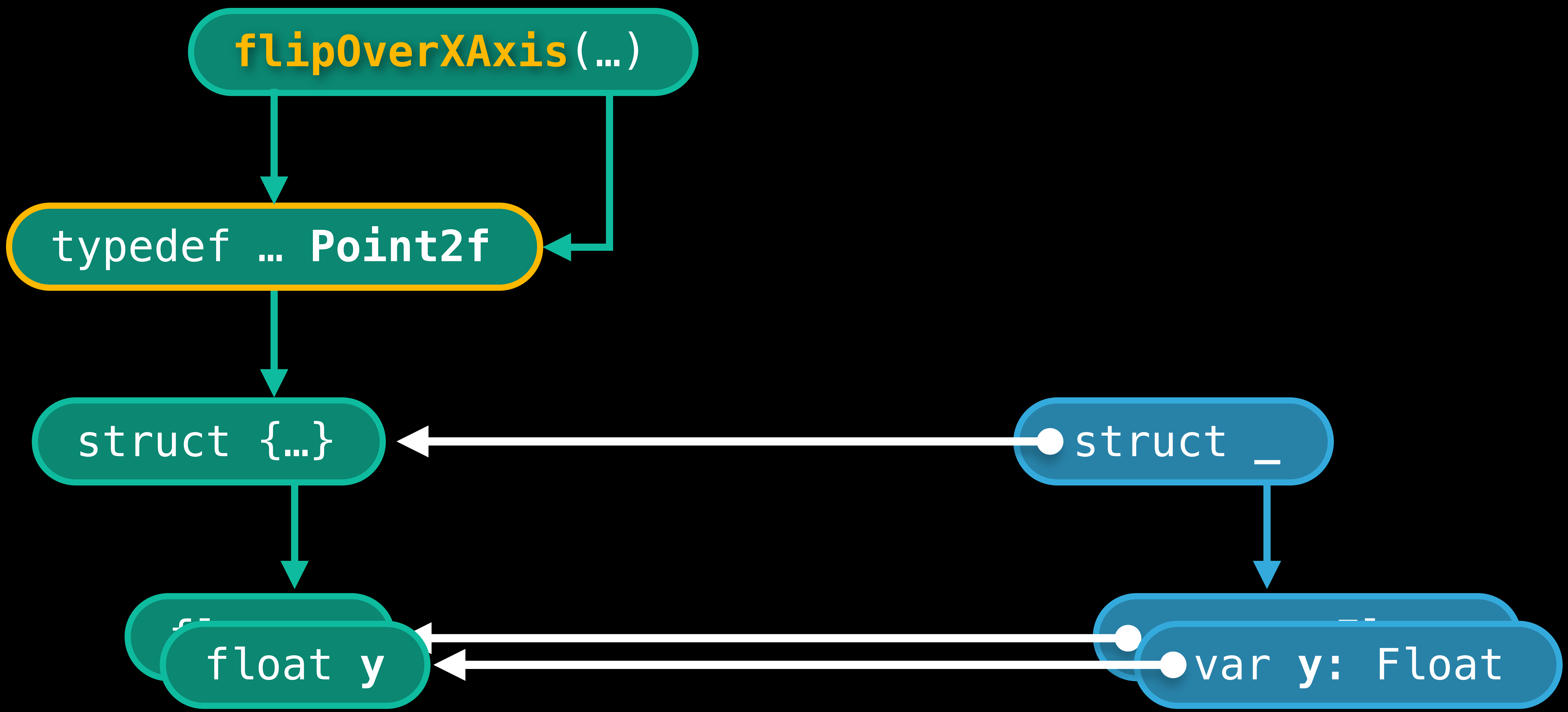
Importing Declarations

...using clang::ASTVisitor



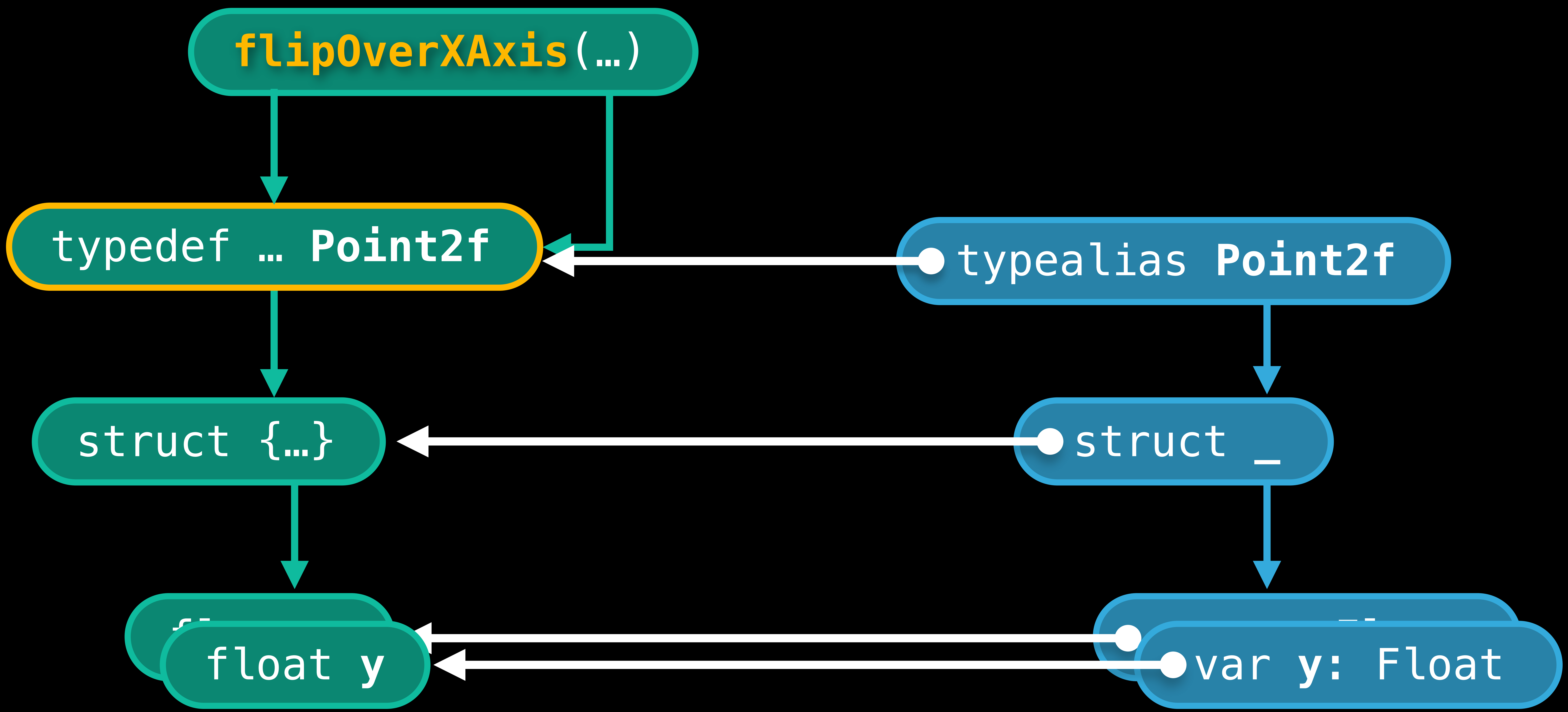
Importing Declarations

...using clang::ASTVisitor



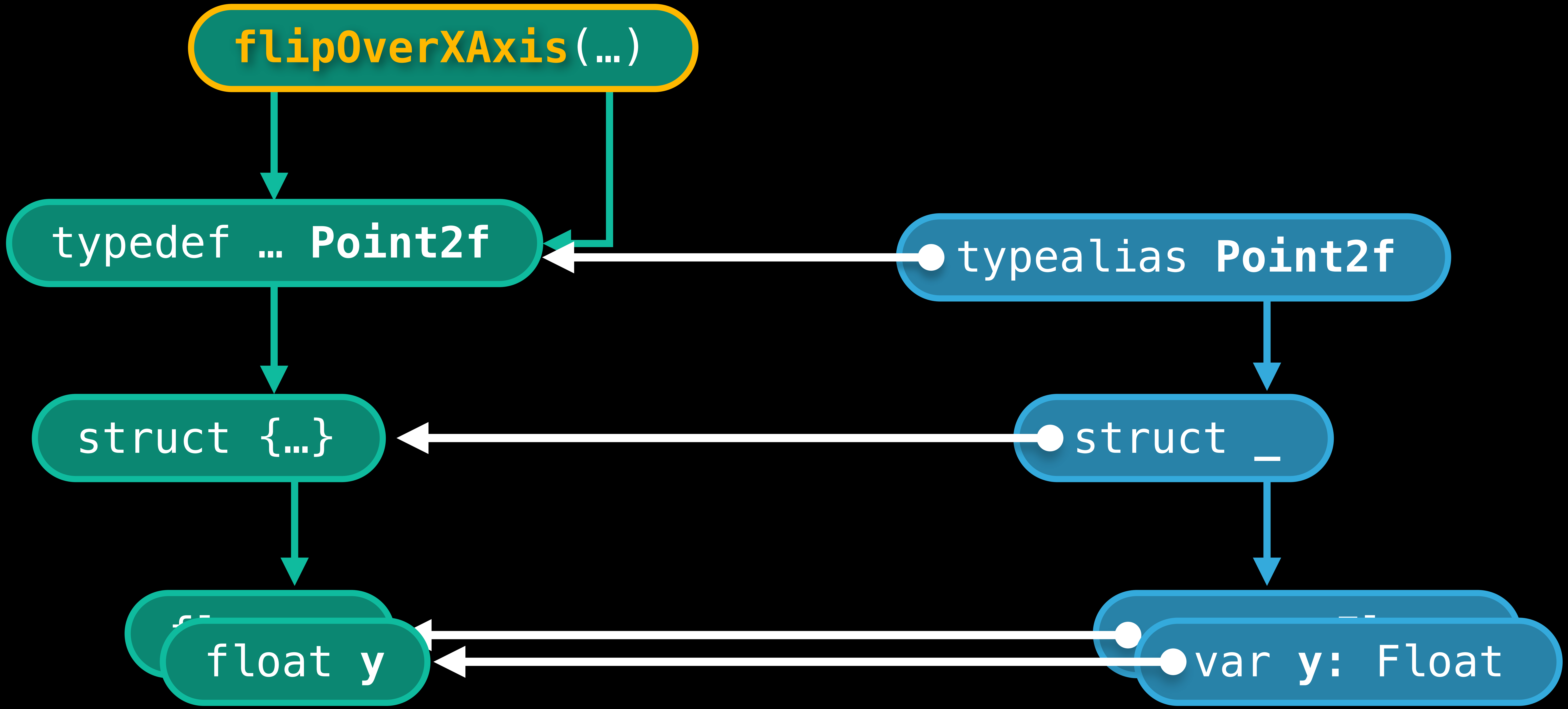
Importing Declarations

...using `clang::ASTVisitor`



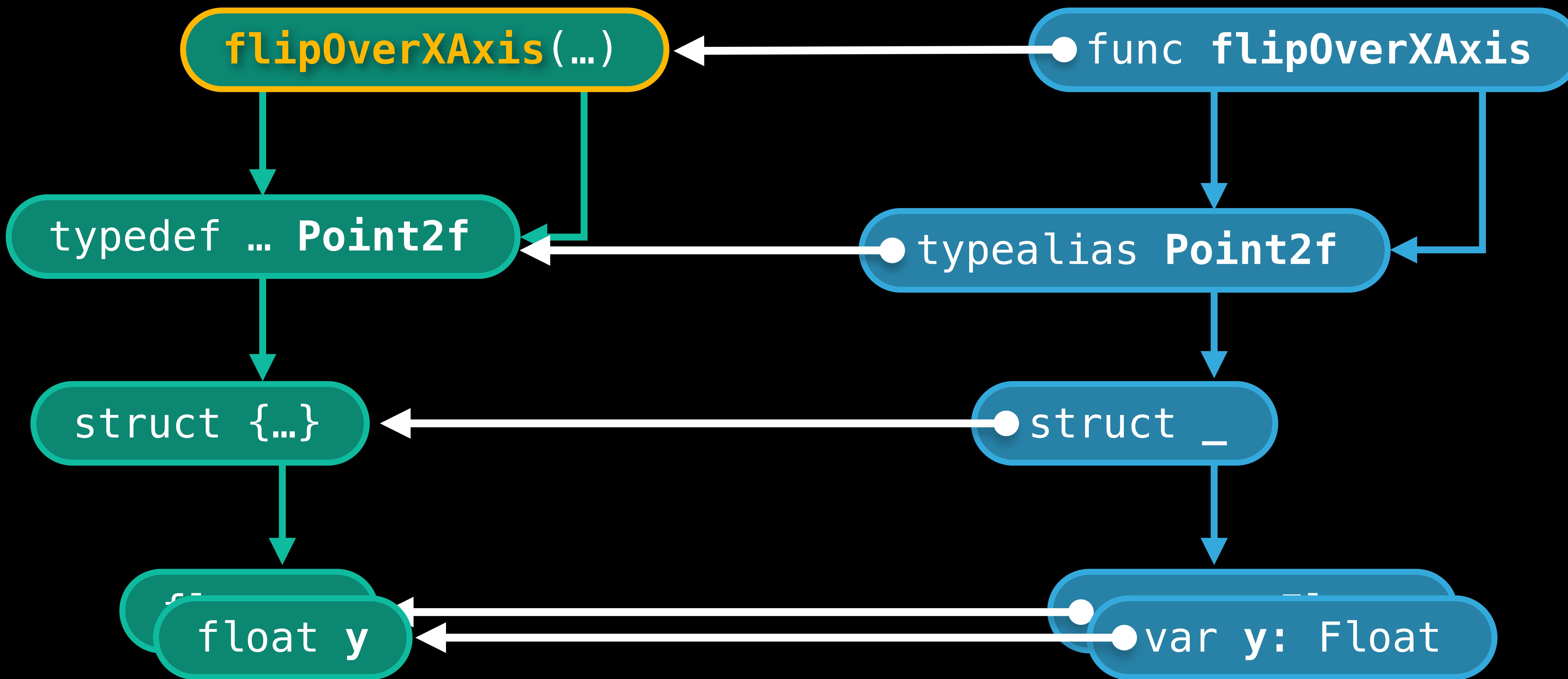
Importing Declarations

...using clang::ASTVisitor



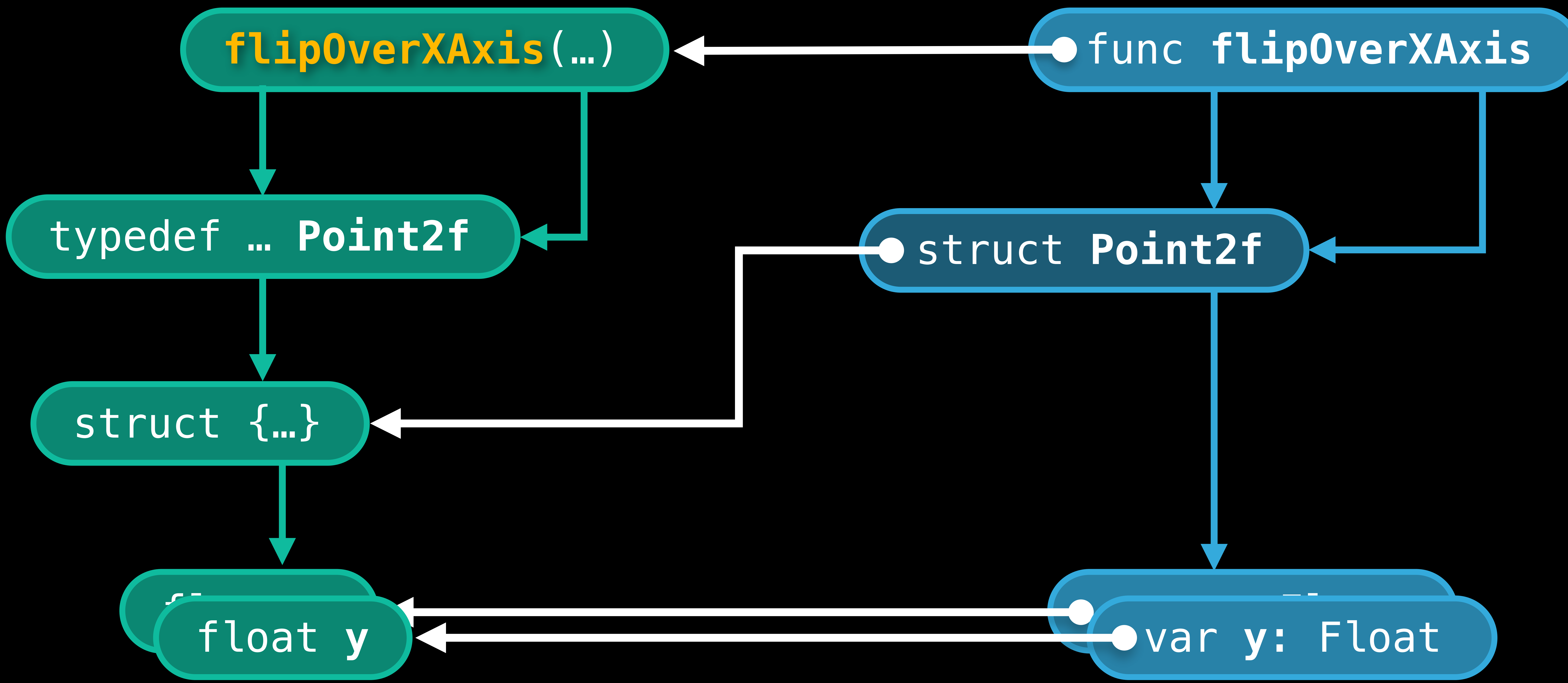
Importing Declarations

...using clang::ASTVisitor

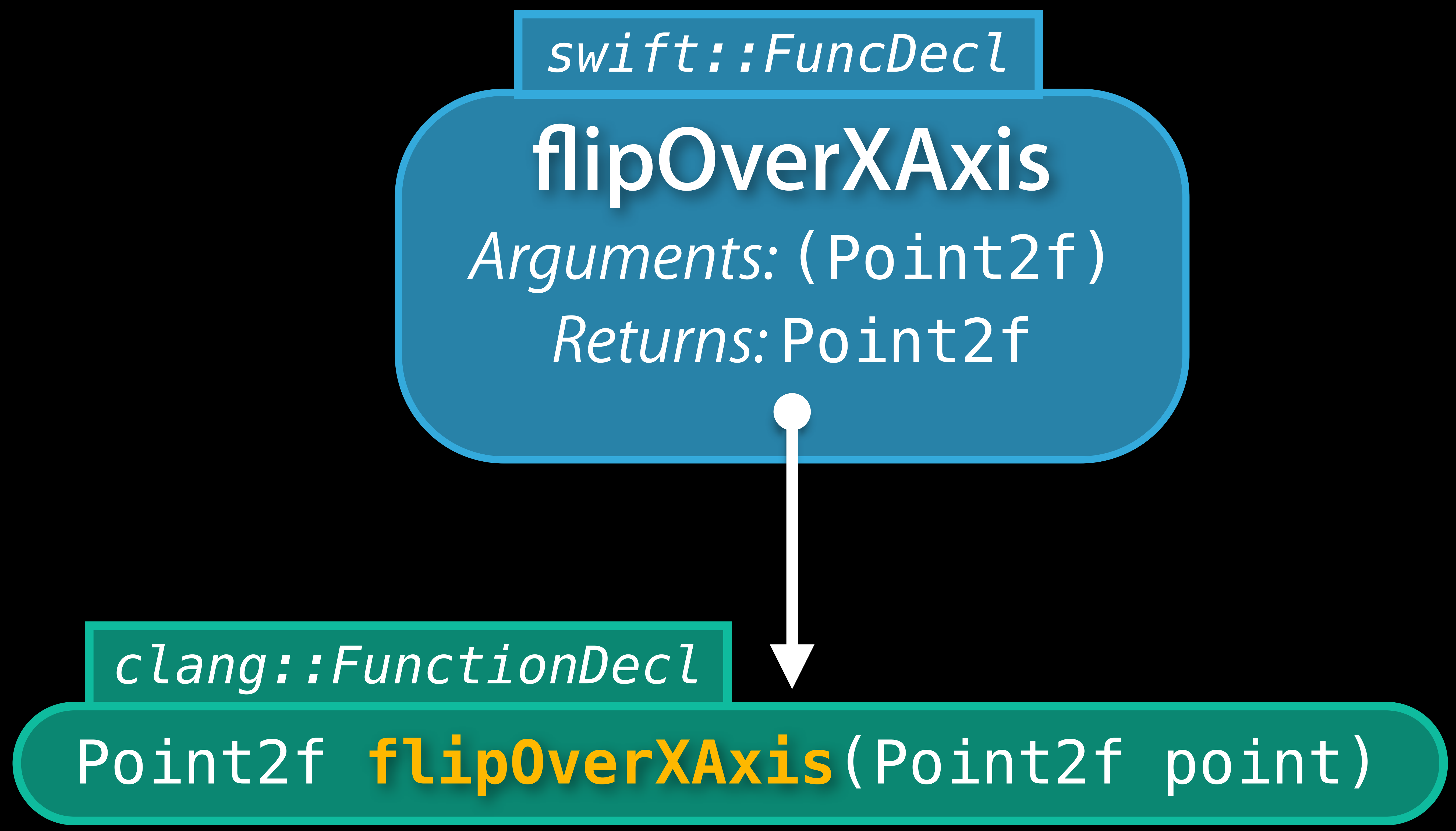


Importing Declarations

...using clang::ASTVisitor



Success!


swift::FuncDecl
flipOverXAxis
Arguments: (Point2f)
Returns: Point2f
clang::FunctionDecl
Point2f flipOverXAxis(Point2f point)

…and back to C

ABIs

Platforms and ABIs

Every language/platform combination forms an ABI

ABI defines how the language is implemented on that platform

Necessary for interoperation:

- ...between compilers offered by different vendors
- ...between different versions of the same compiler
- ...between compiled code and hand-written code (e.g. in assembly)
- ...between compiled code and various inspection/instrumentation tools

ABIs for other languages

All languages/extensions supported by Clang have ABIs defined mostly in terms of C

- Caveat: often require additional linker support
- Caveat: sometimes use slightly different calling conventions

"Itanium" C++ ABI: weak linkage

Visual Studio C++ ABI: weak linkage, different CC for member functions

GNUStep Objective-C ABI: pure C

Apple Objective-C ABI: some Apple-specific linker behavior

Objective-C Blocks ABI: pure C

ABIs for C

Often written by the architecture vendor and then tweaked by the OS vendor

Includes:

- Stack alignment rules
- Calling conventions and register use rules
- Size/alignment of fundamental types
- Layout rules for structs and unions
- Existence of various extended types
- Object file structure and linker behavior
- Guaranteed runtime facilities
- ...and a whole lot more

ABIs and undefined behavior

An ABI doesn't mean language-specific restrictions aren't still in effect!

```
struct A {
    virtual void foo();
};
void *loadVTable(A *a) { return *reinterpret_cast<void**>(a); }
```

Still undefined behavior

Memory

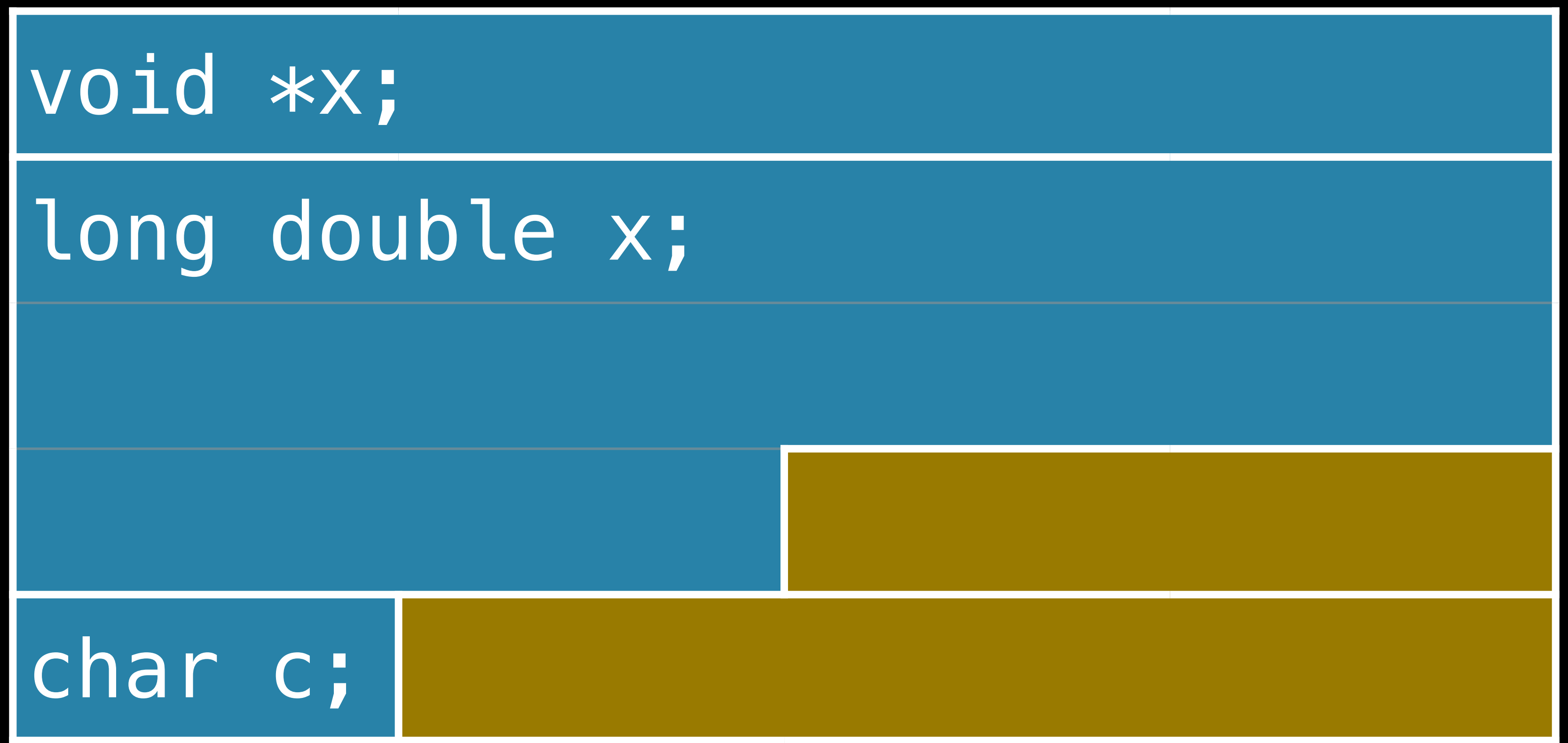
Working with C values in memory

- Often need to allocate storage for C values
- All complete types in C have an ABI size and alignment:
  - getASTContext().getTypeInfoInChars(someType)
- For normal types, sizeof(T) is always a multiple of alignof(T)
  - ...but attributes on typedefs can arbitrarily change alignment requirements

Storage Padding

For many types, sizeof includes some extra storage:

```
struct Foo {
    void *x;
    long double d;
    char c;
};
```



Contents are undefined: not required to preserve those bits

If you share pointers with C code, it won't promise to preserve them either

Special case: C99 _Bool / C++ bool are always stored as 0 or 1 (not necessarily 1 byte)

struct/union Layout

Often tempting to do your own C struct layout:

```
struct Foo {
   void *x;
   long double d;
   char c;
};
```

```
%struct.Foo = {
  opaque*,
  x86_fp80,
  i8
}
```

struct/union Layout

Often tempting to do your own C struct layout:

```
struct Foo {
   void *x;
   long double d;
   char c;
};
```

```
%struct.Foo = {
  opaque*,
  x86_fp80,
  i8
}
```

It's a trap!

struct/union Layout

C/C++ language guarantees:

- All union members have same address
- First struct member has same address as struct
- Later struct member addresses > earlier struct member addresses

Universal C Layout Algorithm

```
struct.size = 0, struct.alignment = 1
for field in struct.fields:
    struct.size = roundUpToAlignment(struct.size, field.alignment)
    struct.alignment = max(struct.alignment, field.alignment)
    offsets[field] = struct.size
    struct.size += field.size
struct.size = roundUpToAlignment(struct.size, alignment)
```

Not guaranteed, but might as well be

Universal C Layout Algorithm?

Bitfield rules differ massively between platforms

Many different attributes and pragmas affect layout

C++...

Use Clang

Type info for struct/union types reflects results of layout

Can get offsets of individual members:

- ASTContext::getASTRecordLayout(const RecordDecl *D)

IRGen provides interfaces for:

- lowering types to IR
- projecting the address of an ordinary field
- loading and storing to a bitfield

Calls

Calls

Calls

Lowering from Clang function types to LLVM function types

Calls

Lowering from Clang function types to LLVM function types

Inputs: AST calling convention, parameter types, return type

Calls

Lowering from Clang function types to LLVM function types

Inputs: AST calling convention, parameter types, return type

Outputs: LLVM calling convention, parameter types, return type, parameter attributes

Why not just use the C type system?

Things that affect CC lowering:

- Exact structure of unions
- Existence and placement of bitfields
- Attributes
- Special cases for types that structurally resemble others
- Everything!

Would have to render entire C type system in LLVM, including all extensions

Frontend/backend mutual aggression pact

- Backend figures out how to represent different ways to pass arguments, results
  - Specific IR types
  - Specific attributes on call site
- Frontend contrives to mutilate arguments into that form

Examples

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}

typedef struct {
  float x, y;
} Point2f;
```

Examples

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}

typedef struct {
  float x, y;
} Point2f;
```

```
// aarch64-apple-ios
define %struct.Point2f @flipOverXAxis(float, float)
```

Examples

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}

typedef struct {
  float x, y;
} Point2f;
```

```
// i386-apple-macosx
define i64 @flipOverXAxis(float, float)
```

Examples

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}

typedef struct {
  float x, y;
} Point2f;
```

```
// thumbv7-apple-ios
define void @flipOverXAxis(%struct.Point2f* sret, [2 x i32])
```

Examples

```
static inline
Point2f flipOverXAxis(Point2f point) {
  // ...
}

typedef struct {
  float x, y;
} Point2f;
```

```
// x86_64-apple-macosx
define <2 x float> @flipOverXAxis(<2 x float>)
```

Relief

LLVM does make an informal ABI guarantee:

A type is "register-filling" if it's a pointer or pointer-sized integer. If:

1) all the arguments are register-filling and

2) the return value is either register-filling or void

Then the obvious type lowering will match the C ABI

Relief

Guaranteed by all the normal CPU backends

Does not apply to floats, structs, vectors, too-small integers, too-large integers, etc.

Extremely useful for free-coding calls to known functions in your language runtime

Breakdown in negotiations

The current situation is pretty gross and increasingly untenable

Backends feel the need to be pretty heroic about what types they accept

Difficult for frontends to tweak CCs, which is often useful when moving beyond C

Entente

Representing whole C type system is unworkable

We should consider going the other way:

- Allow frontends more explicit control of registers and stack
- Make consistent rules about how different IR types are passed otherwise

Use Clang

IRGen provides an interface for examining function type lowering

Extremely detailed, poorly documented

Not a good combination!

Still better than doing it yourself

In progress: extracting better interfaces to do this lowering

Sharing a Module with Clang

Types and global declarations

Your frontend's IR types and Clang's can coexist in a module

Your frontend and Clang will sometimes both need to refer to the same entity

The types won't always match

Global declarations

IRGen is pretty forgiving about the type of a *declaration*

Feel free to emit your own declaration with its own type

Those code paths are well-covered in IRGen because of incomplete types

If Clang has to emit the *definition*, it may have to change the type

This will invalidate your own references to that declaration

...unless you hold onto them with a ValueHandle

...which is best practice anyway

Lazy declaration emission

IRGen only emits certain entities if they're actually used:

- static or inline functions
- certain v-tables

To get IRGen to emit it, you simply:

- tell IRGen that it has a definition (by adding it)
- ask IRGen for a declaration
- ensure that all deferred declarations are emitted

Better APIs for this are in progress

Summary

Summary

You can use Clang to import C types and declarations directly into your language

Let Clang handle the ABI rules for you instead of reinventing them

Most of the APIs for this could be improved