Modules

Doug Gregor, Apple

Roadmap

- The fundamental brokenness of headers
- A module system for the C family
- Building better tools

On the Fundamental Brokenness of Headers

C Preprocessing Model

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

C Preprocessing Model

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

```
// stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

C Preprocessing Model

```
#include <stdio.h>

int main() {
  printf(“Hello, world.
}
```

```
// stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

C Preprocessing Model

```
// from stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...

int main() {
  printf(“Hello, world!\n”);
}
```

Problems with the Model

```
// from stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...

int main() {
  printf(“Hello, world!\n”);
}
```

- Fragility
- Performance

Header Fragility

```
#define FILE “MyFile.txt”
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

Header Fragility

```
#define FILE “MyFile.txt”
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

```
// stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

Header Fragility

```
#define FILE “MyFile.txt”
#include <stdio.h>

int main() {
  printf(“Hello, world!
}
```

```
// stdio.h

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

Header Fragility

```
// from stdio.h

typedef struct {
  ...
} “MyFile.txt”;

int printf(const char*, ...);
int fprintf(“MyFile.txt” *,
      const char*, ...);
int remove(const char*);

// on and on...

int main() {
  printf(“Hello, world!\n”);
}
```

Conventional Workarounds

Conventional Workarounds

- LLVM_WHY_PREFIX_UPPER_MACROS

Conventional Workarounds

- LLVM_WHY_PREFIX_UPPER_MACROS
- LLVM_CLANG_INCLUDE_GUARD_H

Conventional Workarounds

- LLVM_WHY_PREFIX_UPPER_MACROS
- LLVM_CLANG_INCLUDE_GUARD_H
- ```
  template<class _Tp>
  const _Tp& min(const _Tp &__a,
                 const _Tp &__b);
  ```
  ```

Conventional Workarounds

- LLVM_WHY_PREFIX_UPPER_MACROS
- LLVM_CLANG_INCLUDE_GUARD_H
- ```
  template<class _Tp>
  const _Tp& min(const _Tp &__a,
                 const _Tp &__b);
  ```
- ```
  #include <windows.h>
  #undef min // because #define NOMINMAX
  #undef max // doesn't always work
  ```

How Big is a Source File?

How Big is a Source File?

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

How Big is a Source File?

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

	C Hello
Source	64
Headers	11,072

How Big is a Source File?

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

```
#include <iostream>

int main() {
  std::cout << “Hello, world!”
            << std::endl;
}
```

	C Hello
Source	64
Headers	11,072

How Big is a Source File?

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

```
#include <iostream>

int main() {
  std::cout << “Hello, world!”
            << std::endl;
}
```

	C Hello	C++ Hello
Source	64	81
Headers	11,072	1,161,033

How Big is a Source File?

```
#include <stdio.h>

int main() {
  printf(“Hello, world!\n”);
}
```

```
#include <iostream>

int main() {
  std::cout << “Hello, world!”
            << std::endl;
}
```

	C Hello	C++ Hello	SemaOverload
Source	64	81	469,939
Headers	11,072	1,161,033	3,824,521

Inherently Non-Scalable

- M headers with N source files
  - → M x N build cost
- C++ templates exacerbate the problem
- Precompiled headers are a terrible solution

A Module System for the C Family

What Is a Module?

- A module is a package describing a library
  - Interface of the library (API)
  - Implementation of the library

Module Imports

```
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

- ‘import’ makes the API of the named module available

Module Imports

```
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module (includes stdio)

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

- ‘import’ makes the API of the named module available

Module Imports

```
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module (includes stdio)

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

- ‘import’ makes the API of the named module available

Import Resilience

- ‘import’ ignores preprocessor state within the source file

Import Resilience

```
#define FILE “MyFile.txt”
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

- ‘import’ ignores preprocessor state within the source file

Import Resilience

```
#define FILE “MyFile.txt”
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module (includes stdio)

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

- ‘import’ ignores preprocessor state within the source file

Import Resilience

```
#define FILE “MyFile.txt”
import std;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module (includes stdio)

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

- ‘import’ ignores preprocessor state within the source file

Selective Import

```
// std module
  // stdio submodule
  typedef struct {
    ...
  } FILE;

  int printf(const char*, ...);
  int fprintf(FILE *,
        const char*, ...);
  int remove(const char*);

  // on and on...

  // stdlib submodule

  void abort(void);
  int rand(void);

  // on and on...
```

Selective Import

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module

// stdio submodule

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

```
// stdlib submodule

void abort(void);
int rand(void);

// on and on...
```

Selective Import

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

```
// std module

// stdio submodule

typedef struct {
  ...
} FILE;

int printf(const char*, ...);
int fprintf(FILE *,
      const char*, ...);
int remove(const char*);

// on and on...
```

```
// stdlib submodule

void abort(void);
int rand(void);

// on and on...
```

What Does `import` Import?

- Functions, variables, types, templates, macros, etc.
- Only public API -- everything else can be hidden.
- No special namespace mechanism.

Writing a Module

Futuristic Version

Writing a Module

```
// stdio.c
export std.stdio:

public:
typedef struct {
  ...
} FILE;

int printf(const char*, ...) {
  // ...
}

int fprintf(FILE *,
            const char*, ...) {
  // ...
}

int remove(const char*) {
  // ...
}
```

Writing a Module

```
// stdio.c
export std.stdio:

public:
typedef struct {
  ...
} FILE;

int printf(const char*, ...) {
  // ...
}

int fprintf(FILE *,
            const char*, ...) {
  // ...
}

int remove(const char*) {
  // ...
}
```

- Specify module name in source file

Writing a Module

```
// stdio.c
export std.stdio:

public:
typedef struct {
  ...
} FILE;

int printf(const char*, ...) {
  // ...
}

int fprintf(FILE *,
            const char*, ...) {
  // ...
}

int remove(const char*) {
  // ...
}
```

- Specify module name in source file
- Public access describes API

Writing a Module

```
// stdio.c
export std.stdio:

public:
typedef struct {
  ...
} FILE;

int printf(const char*, ...) {
  // ...
}

int fprintf(FILE *,
            const char*, ...) {
  // ...
}

int remove(const char*) {
  // ...
}
```

- Specify module name in source file
- Public access describes API
- No headers!

Problems With This Future

- Transitioning existing header-based libraries
- Interoperability with compilers that don't implement modules
- Requires tools that understand modules

Writing a Module

Transitional Version

Embracing Headers

- Build modules directly from the headers
- Headers remain “the truth”
  - Good for interoperability
  - Doesn’t change the programmer model

Module Maps

```
// /usr/include/module.map

module std {
  module stdio { header “stdio.h” }
  module stdlib { header “stdlib.h” }
  module math { header “math.h” }
}
```

- `module` defines a named (sub)module
- `header` includes the contents of the named header in the current (sub)module

Umbrella Headers

```
// clang/include/clang/module.map

module ClangAST {
  umbrella header “AST/AST.h”
  module * { }
}
```

- An umbrella header includes all of the headers in its directory

Umbrella Headers

```
// clang/include/clang/module.map

module ClangAST {
  umbrella header “AST/AST.h”
  module * { }
}
```

- An umbrella header includes all of the headers in its directory
- Wildcard submodules (`module *`) create a submodule for each included header
  - `AST/Decl.h` -> `ClangAST.Decl`
  - `AST/Expr.h` -> `ClangAST.Expr`

Module Map Features

Module Map Features

- Umbrella directories

```
module LLVMADT {
  umbrella “llvm/ADT”
  module * { export * }
}
```

Module Map Features

- Umbrella directories

```
module LLVMADT {
  umbrella “llvm/ADT”
  module * { export * }
}
```

- Submodule requirements

```
module _Builtin {
  module avx {
    requires avx
    header “avxintrin.h”
  }
}
```

Module Map Features

- Umbrella directories

```
module LLVMADT {
  umbrella “llvm/ADT”
  module * { export * }
}
```

- Submodule requirements

```
module _Builtin {
  module avx {
    requires avx
    header “avxintrin.h”
  }
}
```

- Excluded headers

```
module std {
  exclude header “assert.h”
}
```

Compilation Model

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

Compilation Model

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

1. Find a module map for the named module

Compilation Model

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

1. Find a module map for the named module
2. Spawn a separate instance of the compiler:
   - Parse the headers in the module map
   - Write the module file

Compilation Model

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

1. Find a module map for the named module
2. Spawn a separate instance of the compiler:
   - Parse the headers in the module map
   - Write the module file
3. Load the module file at the ‘import’ declaration

Compilation Model

```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

1. Find a module map for the named module
2. Spawn a separate instance of the compiler:
   - Parse the headers in the module map
   - Write the module file
3. Load the module file at the ‘import’ declaration
4. Cache module file for later re-use

Adopting Modules: Libraries

- Eliminate non-modular behavior:
  - Multiply-defined structs, functions, macros, must be consolidated
  - Headers should import what they depend on
- Write module maps covering the library

Adopting Modules: Users

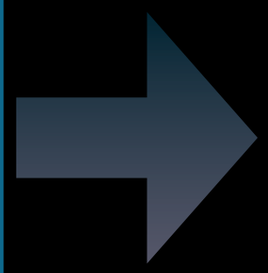
```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```

Adopting Modules: Users

```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```



```
import std.stdio;

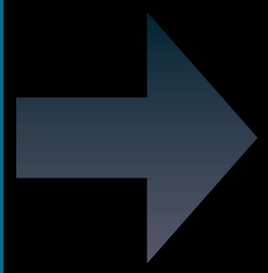
int main() {
  printf(“Hello, World!\n”);
}
```

- “Simply” rewrite each #include as an import

Translating #include to import

```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```



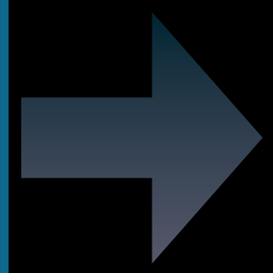
```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

Translating #include to import

```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```



```
import std.stdio;

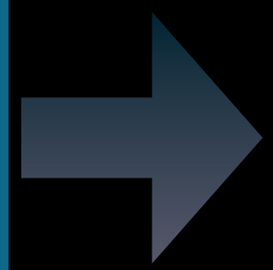
int main() {
  printf(“Hello, World!\n”);
}
```

- Use module maps to determine (sub)module corresponding to an #include’d header

Translating #include to import

```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```



```
import std.stdio;

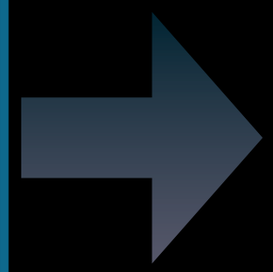
int main() {
  printf(“Hello, World!\n”);
}
```

- Use module maps to determine (sub)module corresponding to an #include’d header
- Optional Fix-Its, tooling to finalize the rewrite

Translating #include to import

```
#include <stdio.h>

int main() {
  printf(“Hello, World!\n”);
}
```



```
import std.stdio;

int main() {
  printf(“Hello, World!\n”);
}
```

- Use module maps to determine (sub)module corresponding to an #include’d header
- Optional Fix-Its, tooling to finalize the rewrite
- Enabling modules is transparent to the user

Building Better Tools

(For the Future)

Compilation Performance

- *Algorithmic* improvement for parsing time
  - Module headers parsed once, cached
  - M x N → M + N
- Benefits for all source-based tools

Automatic Linking

```
// clang/include/clang/module.map

module ClangAST {
  umbrella header “AST/AST.h”
  module * { }
  link “-lclangAST”
}
```

- Drastically simplifies use of a library

Automatic Import

```
int main() {
  std::vector<int> v;
}
```

- Forgotten `#include` → terrible diagnostic

Automatic Import

```
int main() {
  std::vector<int> v;
}
```

- Forgotten `#include` → terrible diagnostic
- **vector.cpp:2:6: error: ‘vector‘ template is not available**

```
  std::vector<int> v;
            ^
```

note: import ‘std.vector‘ to use ‘std::vector’

Debugging Flow

Debugging Flow

<iostream> → foo.cpp

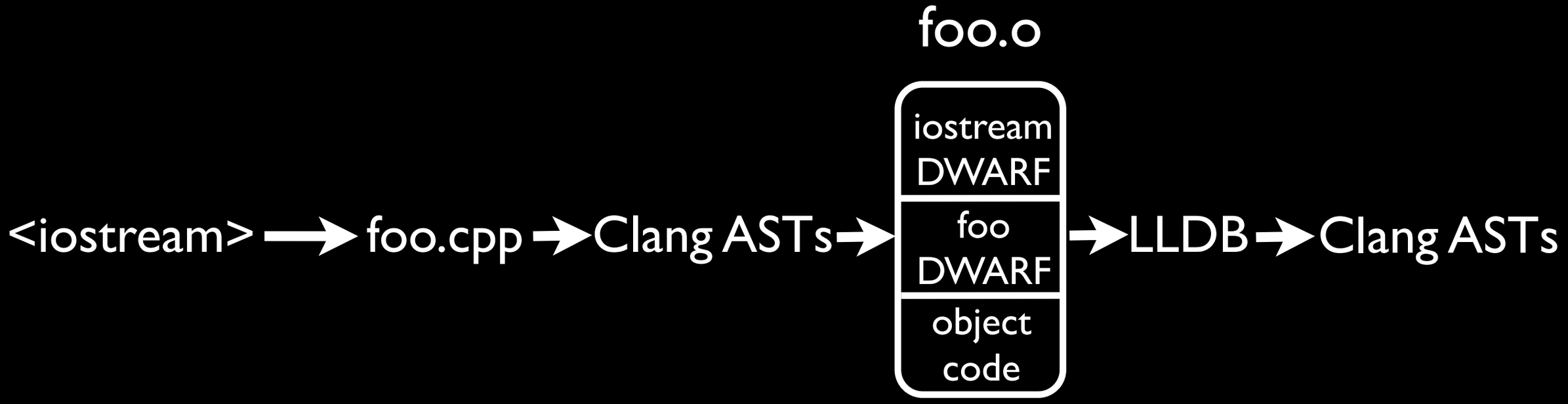
Debugging Flow

<iostream> → foo.cpp → Clang ASTs

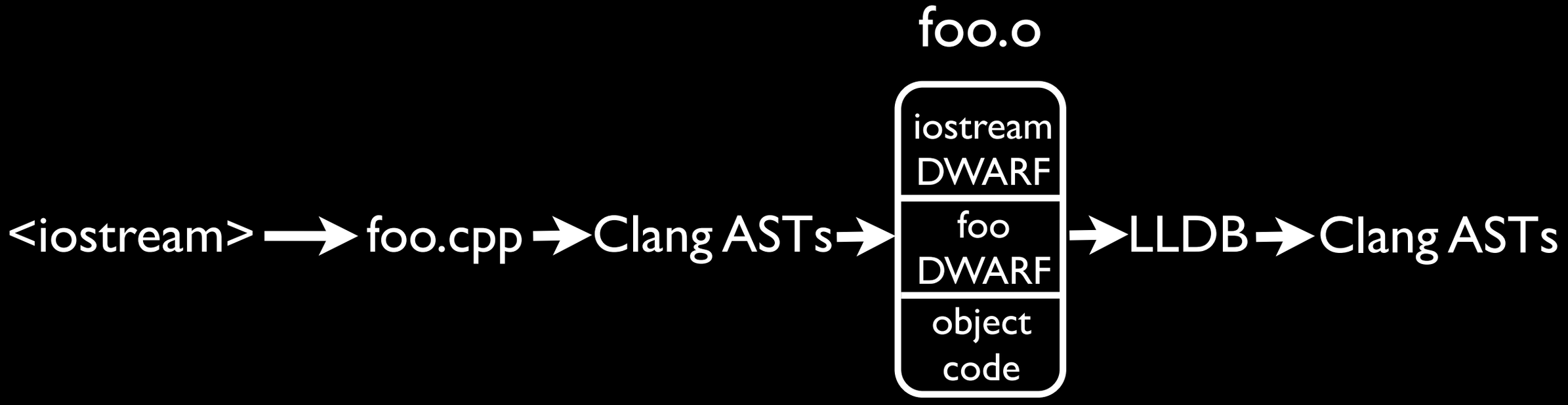
Debugging Flow

Debugging Flow

Debugging Flow

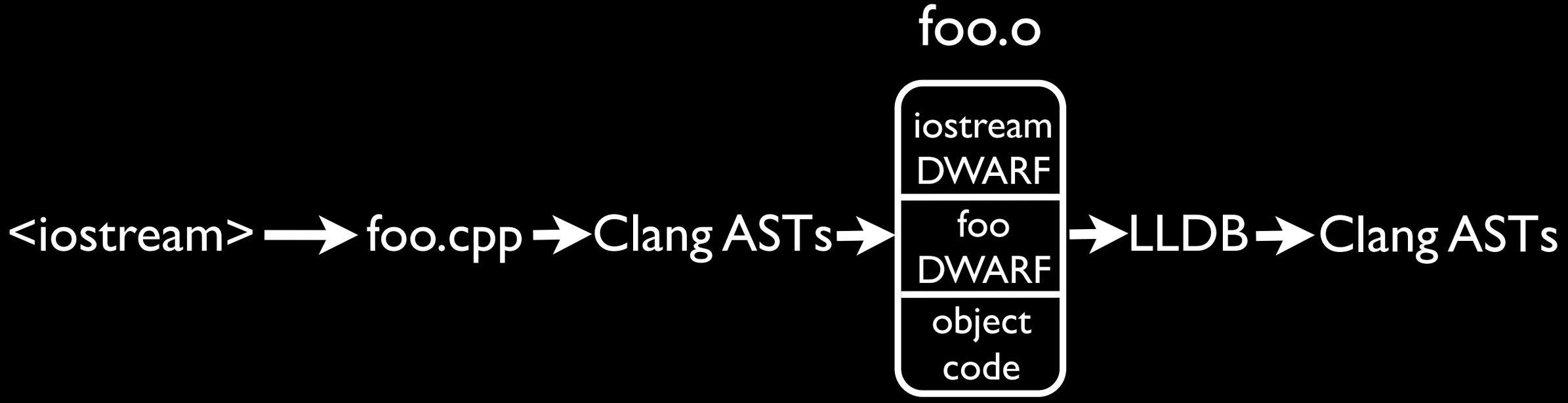


Debugging Flow

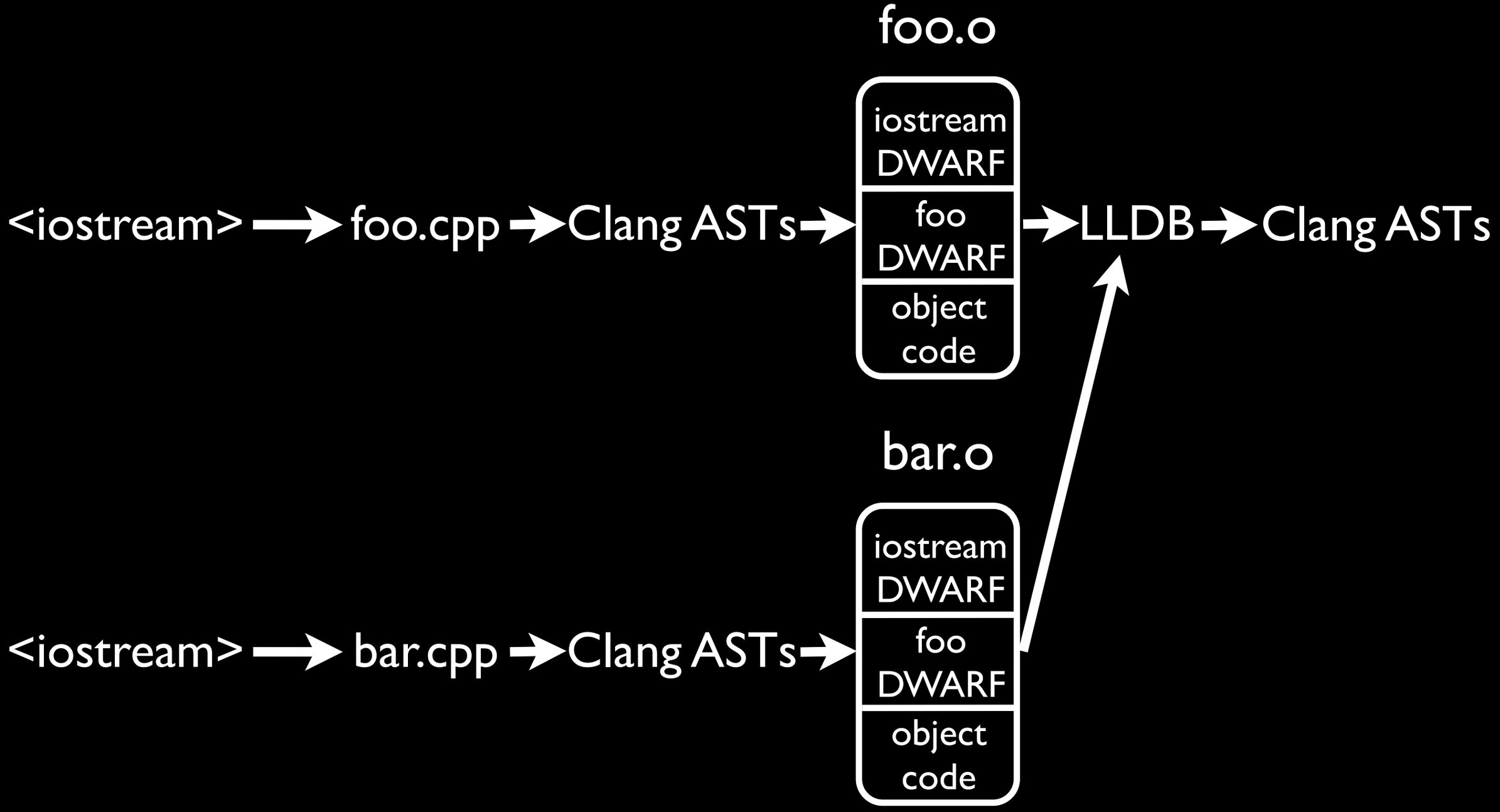


- Round-trip through DWARF is lossy
  - Only ‘used’ types, functions available
  - Inline functions, template definitions lost

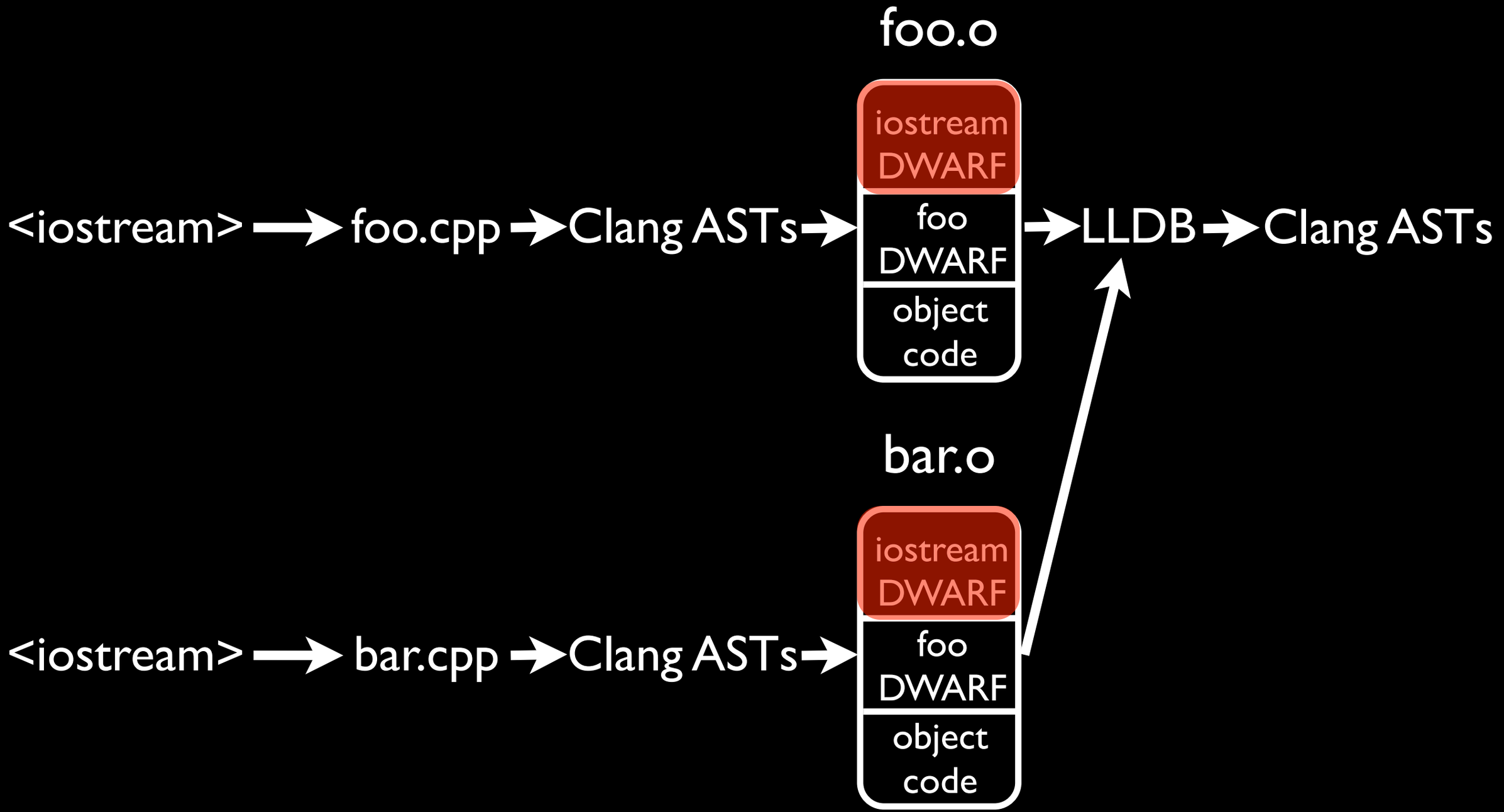
Redundant Debug Info



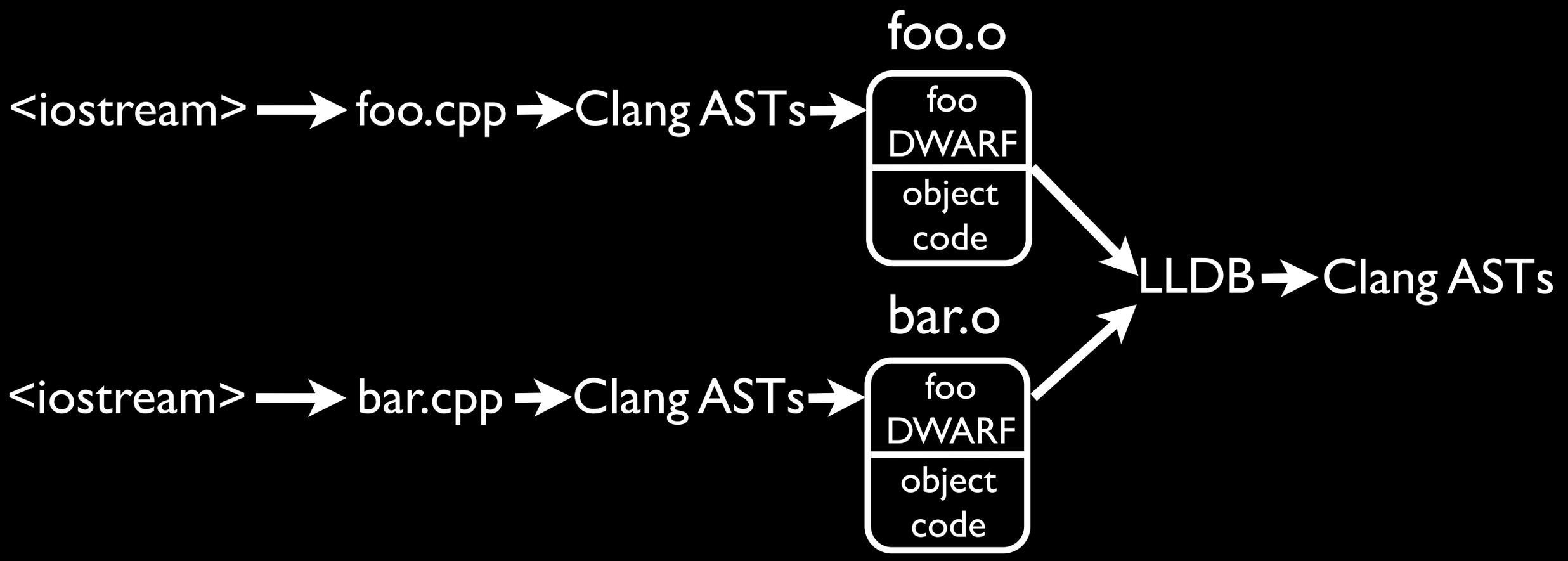
Redundant Debug Info



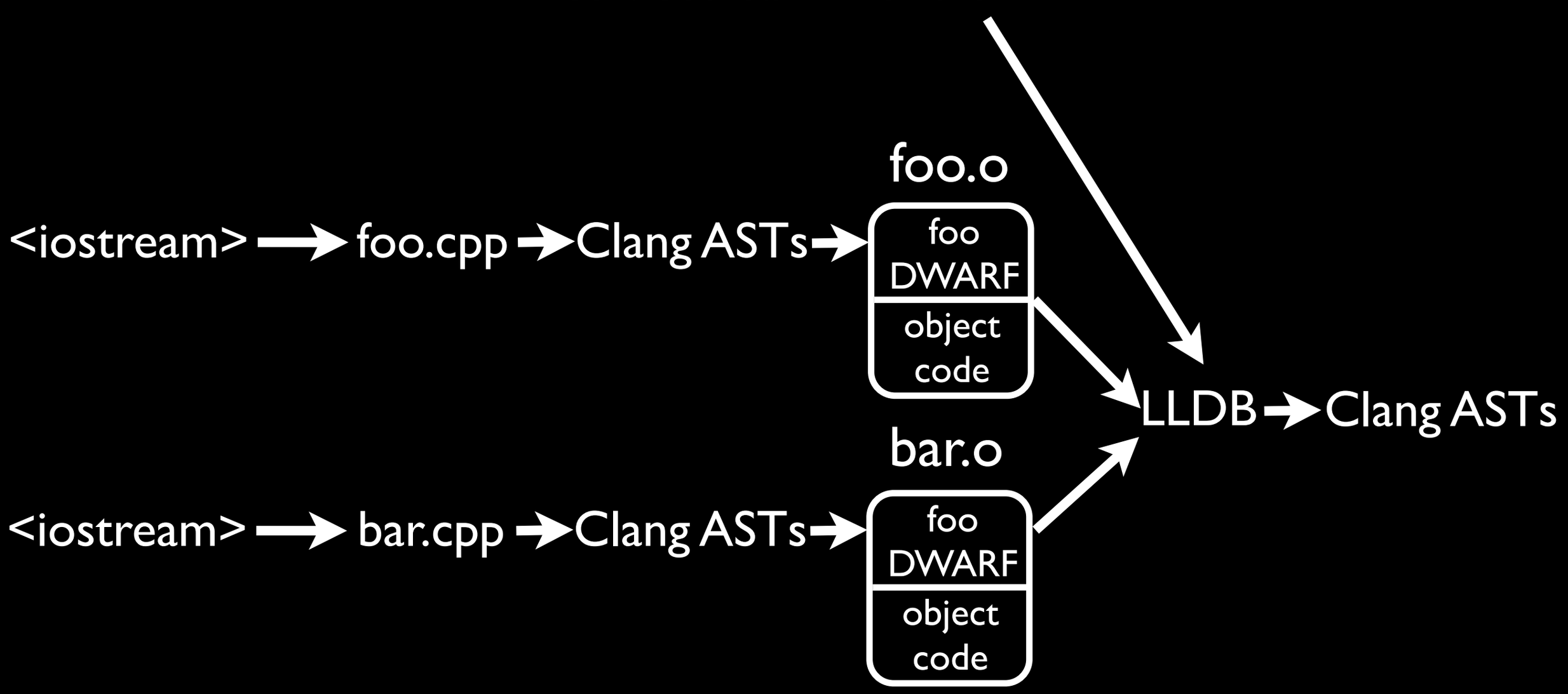
Redundant Debug Info



Debugging with Modules



Debugging with Modules



Debugging with Modules

- Improved build performance
  - Compiler emits less DWARF
  - Linker de-duplicates less DWARF

Debugging with Modules

- Improved build performance
  - Compiler emits less DWARF
  - Linker de-duplicates less DWARF
- Improved debugging experience
  - Perfect AST fidelity in debugger
  - Debugger doesn't need to search DWARF

Modules Summary

Modules Summary

- Modules are a huge potential win for C(++)
  - Compile/build time improvements
  - Fix various preprocessor problems
  - Far better tool experience

Modules Summary

- Modules are a huge potential win for C(++)
  - Compile/build time improvements
  - Fix various preprocessor problems
  - Far better tool experience
- Design enables smooth transition path

Modules Summary

- Modules are a huge potential win for C(++)
  - Compile/build time improvements
  - Fix various preprocessor problems
  - Far better tool experience
- Design enables smooth transition path
- Clang implementation underway