Debug Info for Macros

Adrian Prantl

EuroLLVM 2024 | Apple | 4/10/2024

Macros

Macros

The "Integration" Spectrum

separate from language

integral part of language

Macros

The "Integration" Spectrum

separate from language

integral part of language

Macros

The "Integration" Spectrum

C Preprocessor

- separate language

Macros

The "Integration" Spectrum

separate from language — integral part of language

C Preprocessor

- separate language
- can be implemented outside of compiler

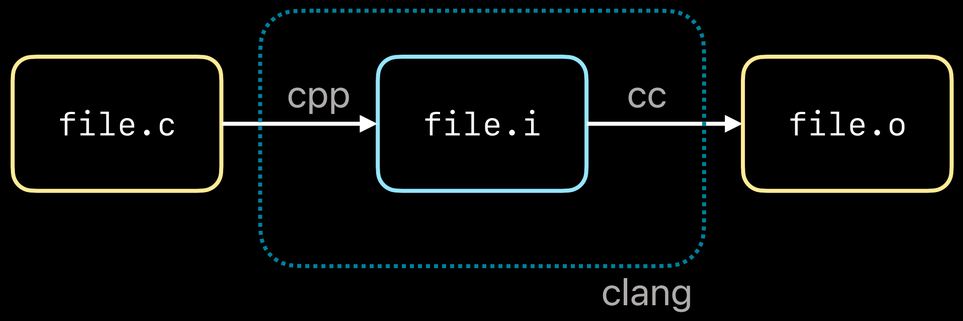
Macros

The "Integration" Spectrum

separate from language — integral part of language

C Preprocessor

- separate language
- can be implemented outside of compiler



Macros

The "Integration" Spectrum

C Preprocessor

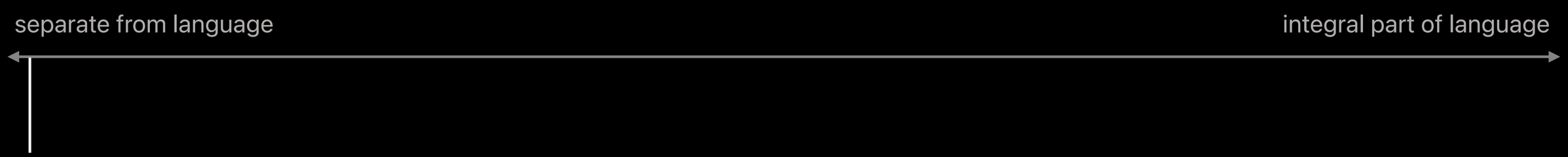
- separate language
- can be implemented outside of compiler
- simple text replacement

```
// Example from Clang sources:

const char *Type::getTypeClassName() const {
  switch (TypeBits.TC) {
#define ABSTRACT_TYPE(Derived, Base)
#define TYPE(Derived, Base) case Derived:
return #Derived;
#include "clang/AST/TypeNodes.inc"
  }

  llvm_unreachable("Invalid type class.");
}
```

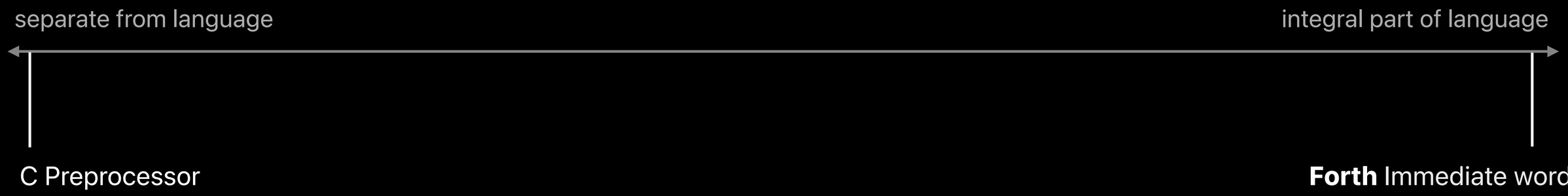
Macros



C Preprocessor

- separate language
- can be implemented outside of compiler
- simple text replacement

Macros



- separate language
- can be implemented outside of compiler
- simple text replacement

Macros

C Preprocessor

- separate language
- can be implemented outside of compiler
- simple text replacement

Forth Immediate words

- integral part of language

Macros

C Preprocessor

- separate language
- can be implemented outside of compiler
- simple text replacement

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros

C Preprocessor

- separate language
- can be implemented outside of compiler
- simple text replacement

```
: begin here ; immediate
```

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Metaprogramming

C Preprocessor

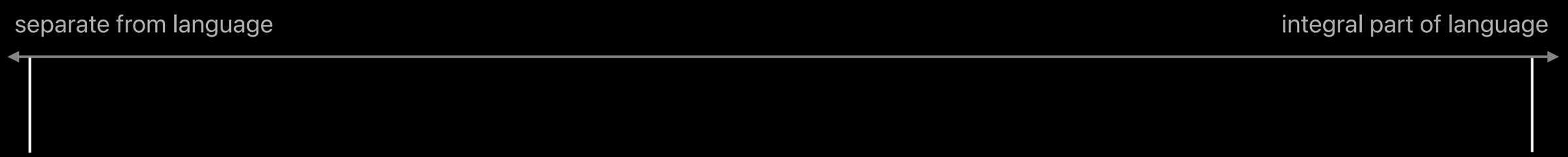
- separate language
- can be implemented outside of compiler
- simple text replacement

```
: begin here ; immediate
```

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros



C Preprocessor

- separate language
- simple text replacement
- can be implemented outside of compiler

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros

C Preprocessor

- separate language
- simple text replacement
- can be implemented outside of compiler

Swift macros

- type safe

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros

C Preprocessor

- separate language
- simple text replacement
- can be implemented outside of compiler

Swift macros

- type safe
- same language

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros

C Preprocessor

- separate language
- simple text replacement
- can be implemented outside of compiler

Swift macros

- type safe
- same language
- compiler plugins

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime

Macros

separate from language

integral part of language

C Preprocessor

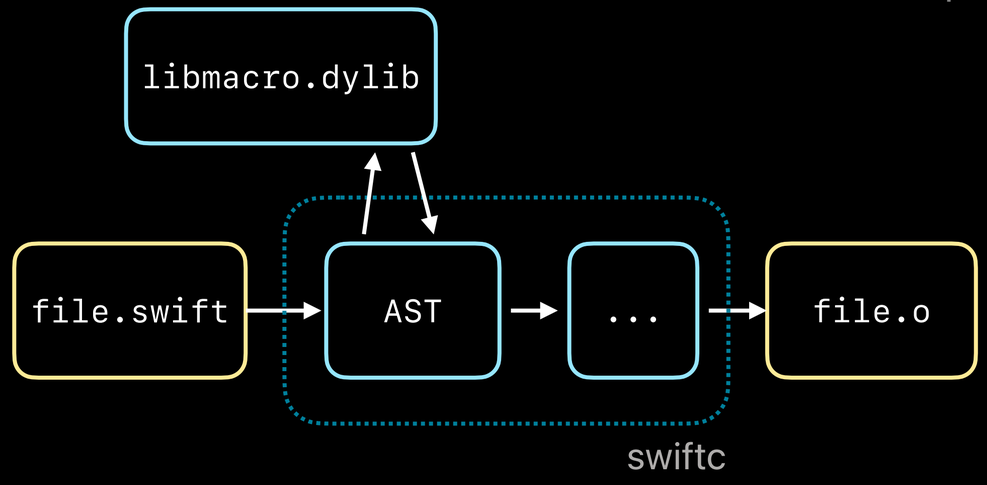
- separate language
- simple text replacement
- can be implemented outside of compiler

Swift macros

- type safe
- same language
- compiler plugins

Forth Immediate words

- integral part of language
- choose wether code is run at compile-time or runtime



C Macros & Debuggers

Short explanatory text about the topic.

Source Locations

Macros are by definition on one line

- no stepping into / through
- no column information

Source Locations

Macros are by definition on one line

- no stepping into / through
- no column information

```
* thread #1, stop reason = hit program assert
    frame #4: 0x0000000100000f38 cmacro`h at cmacro.c:10:3
   2     #define ASSERT_AND(COND, F)                          \
   3       do {                                               \
   4         assert(COND);                                    \
   5         F;                                               \
   6       } while(0)
   7
   8     void f() {}
   9     void h() {
-> 10      ASSERT_AND(0, f());
   11    }
   12
Target 0: (cmacro) stopped.
(lldb)
```

Source Locations

Macros are by definition on one line

- no stepping into / through
- no column information

```
* thread #1, stop reason = hit program assert
    frame #4: 0x0000000100000f38 cmacro`h at cmacro.c:10:3
   2     #define ASSERT_AND(COND, F)                          \
   3       do {                                               \
   4         assert(COND);                                    \
   5         F;                                               \
   6       } while(0)
   7
   8     void f() {}
   9     void h() {
-> 10      ASSERT_AND(0, f());
   11    }
   12
Target 0: (cmacro) stopped.
(lldb) bt
* thread #1, stop reason = hit program assert
    frame #0: 0x00007ff81abb812a libsystem_kernel.dylib`__pthread_kill + 10
    frame #1: 0x00007ff81abf0ebd libsystem_pthread.dylib`pthread_kill + 262
    frame #2: 0x00007ff81ab16a79 libsystem_c.dylib`abort + 126
    frame #3: 0x00007ff81ab15d68 libsystem_c.dylib`__assert_rtn + 314
  * frame #4: 0x0000000100000f38 cmacro`h at cmacro.c:10:3
    frame #5: 0x0000000100000f7b cmacro`main(argc=1, argv=0x00007ff7bfeff0d0) at cmacro.c:21:3
    frame #6: 0x00007ff81a865366 dyld`start + 1942
(lldb)
```

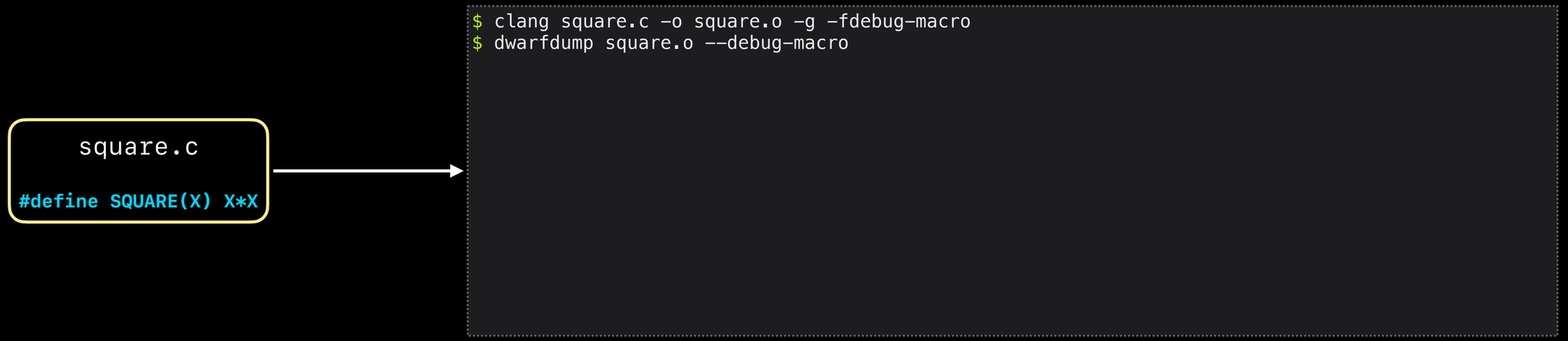
Expression evaluation

Expression evaluation

DWARF debug info can collect each macro redefinition

Debugger could re-expand macros in the source code

Can make macros available in expressions



Expression evaluation

DWARF debug info can collect each macro redefinition

Debugger could re-expand macros in the source code

Can make macros available in expressions

square.c

```
#define SQUARE(X) X*X
```

```
$ clang square.c -o square.o -g -fdebug-macro
$ dwarfdump square.o --debug-macro
.debug_macro contents:
0x00000000:
macro header: version = 0x0005, flags = 0x02, format = DWARF32, debug_line_offset = 0x00000000
DW_MACRO_start_file - lineno: 0 filenum: 0
  DW_MACRO_define_strx - lineno: 1 macro: SQUARE(X) X*X
DW_MACRO_end_file
DW_MACRO_define_strx - lineno: 0 macro: __llvm__ 1
DW_MACRO_define_strx - lineno: 0 macro: __clang__ 1
DW_MACRO_define_strx - lineno: 0 macro: __clang_major__ 19
DW_MACRO_define_strx - lineno: 0 macro: __clang_minor__ 0
DW_MACRO_define_strx - lineno: 0 macro: __clang_patchlevel__ 0
...
```

Swift Macros & Debuggers

Short explanatory text about the topic.

(Freestanding) Swift macros

(Freestanding) Swift macros

- Strongly typed declaration

macro.swift

```swift
@freestanding(expression)
public macro stringify<T>(_ value: T) -> (T, String) =
  #externalMacro(module: "MacroImpl", type: "StringifyMacro")
```

(Freestanding) Swift macros

- Strongly typed declaration `macro.swift`
- Implementation `libmacro.dylib`

```
public struct StringifyMacro: ExpressionMacro {
  public static func expansion(of macro: some FreestandingMacroExpansionSyntax,
                               in context: some MacroExpansionContext)
  -> ExprSyntax
{
    guard let argument = macro.argumentList.first?.expression else { fatalError("") }
    return "(\(argument), \(StringLiteralExprSyntax(content: argument.description)))"
  }
}
```

(Freestanding) Swift macros

- Strongly typed declaration `macro.swift`
- Implementation `libmacro.dylib`
- Expansion site `file.swift`

```
let s = #stringify(a + b)
```

(Freestanding) Swift macros

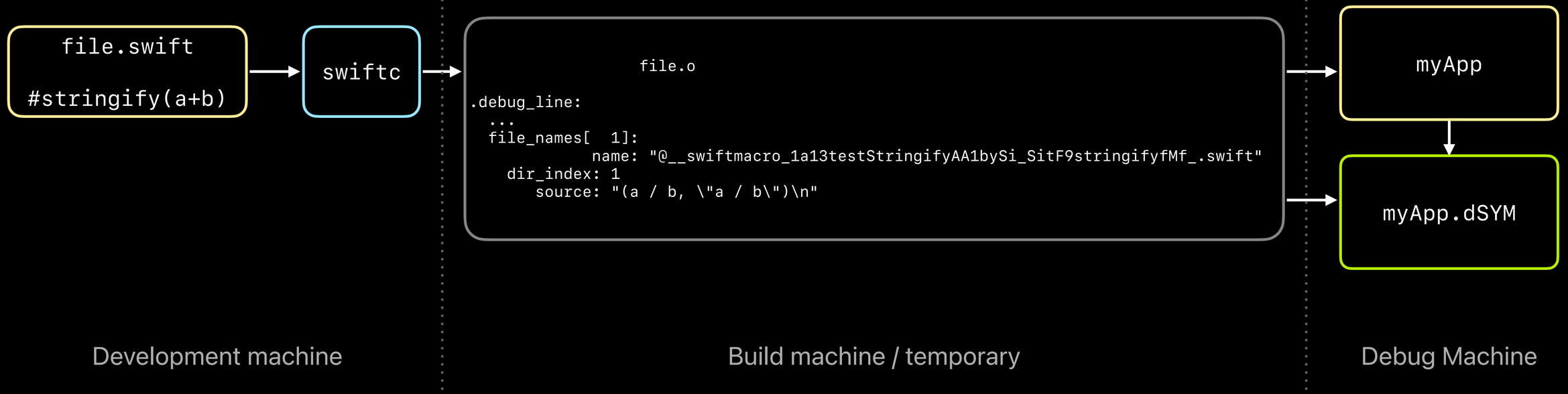
- Strongly typed declaration `macro.swift`
- Implementation `libmacro.dylib`
- Expansion site `file.swift`

Swift Compiler Plugins

Preserving Swift macro expansions

Store macro expansion at compile time in separate file

DWARF Issue 180201.1 DWARF and source text embedding



Integration with IDEs and scripting

DWARF-embedded source files:

- transparent
- LLDB produces temporary file
- LLDB API is unchanged and returns the temporary local file

Stepping & Backtraces

Stepping & Backtraces

Macro expansions are represented as inlined functions

Stepping & Backtraces

Macro expansions are represented as inlined functions

- User can decide whether to step into or over the macro

Stepping & Backtraces

Macro expansions are represented as inlined functions

- User can decide whether to step into or over the macro
- Backtraces for nested macros

```
Process 48393 stopped
* thread #1, stop reason = Fatal error: Division by zero
frame #6: 0x0000000100000bea a.out`testStringify(a:b:) [inlined] freestanding macro expansion #1 of stringify
 in a.testStringify(a: Swift.Int, b: Swift.Int) -> () at -a91222.@__swiftmacro_1a13testStringifyAA1bySi_SitF9stringifyfMf_.swift:1:4
-> 1     (a / b, "a / b")
(lldb)
```

Stepping & Backtraces

Macro expansions are represented as inlined functions

- User can decide whether to step into or over the macro
- Backtraces for nested macros

```
Process 48393 stopped
* thread #1, stop reason = Fatal error: Division by zero
frame #6: 0x0000000100000bea a.out`testStringify(a:b:) [inlined] freestanding macro expansion #1 of stringify
 in a.testStringify(a: Swift.Int, b: Swift.Int) -> () at -a91222.@__swiftmacro_1a13testStringifyAA1bySi_SitF9stringifyfMf_.swift:1:4
-> 1     (a / b, "a / b")
(lldb) up
frame #7: 0x0000000100000b8b a.out`testStringify(a=23, b=0) at main.swift:5:11
   1     import Macro
   2
   3     func testStringify(a: Int, b: Int) {
   4       print("break here")
-> 5       let s = #stringify(a / b)
   6       print(s.1)
   7     }
   8
   9     testStringify(a: 23, b: 0)
(lldb)
```

Macros in LLDB expression evaluator

Macros in LLDB expression evaluator

- LLDB embeds a Swift compiler
  - Cannot load plugins directly
    - Macro could crash!
    - Macros depend on libSwiftSyntax, potential ABI incompatibility
  - LLDB finds macros through Swift module metadata

Macros in LLDB expression evaluator

- LLDB embeds a Swift compiler
  - Cannot load plugins directly
    - Macro could crash!
    - Macros depend on libSwiftSyntax, potential ABI incompatibility
  - LLDB finds macros through Swift module metadata
- Macros are isolated via matching swift-plugin-server process

Summary

How to support new macros in debug info and debuggers

- Better debugging experience by using inline information for macros
- LLDB now supports embedded source file DWARF extension
- Compiler plugins are made available in LLDB, run in separate process