What's New in LLVM

Session 410

Bob Wilson
Manager, LLVM Core Team

Better Compiler → Better Apps

Better Compiler → Better Apps

- Performance
  - Optimized code

Better Compiler → Better Apps

- Performance
  - Optimized code
- Productivity
  - Fast builds
  - New language features

Better Compiler → Better Apps

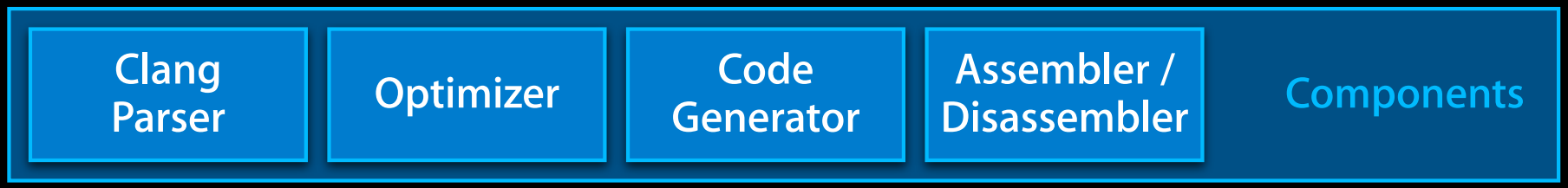
- Performance
  - Optimized code
- Productivity
  - Fast builds
  - New language features
- Quality
  - Compiler warnings
  - Static analysis

Better Compiler → Better Apps

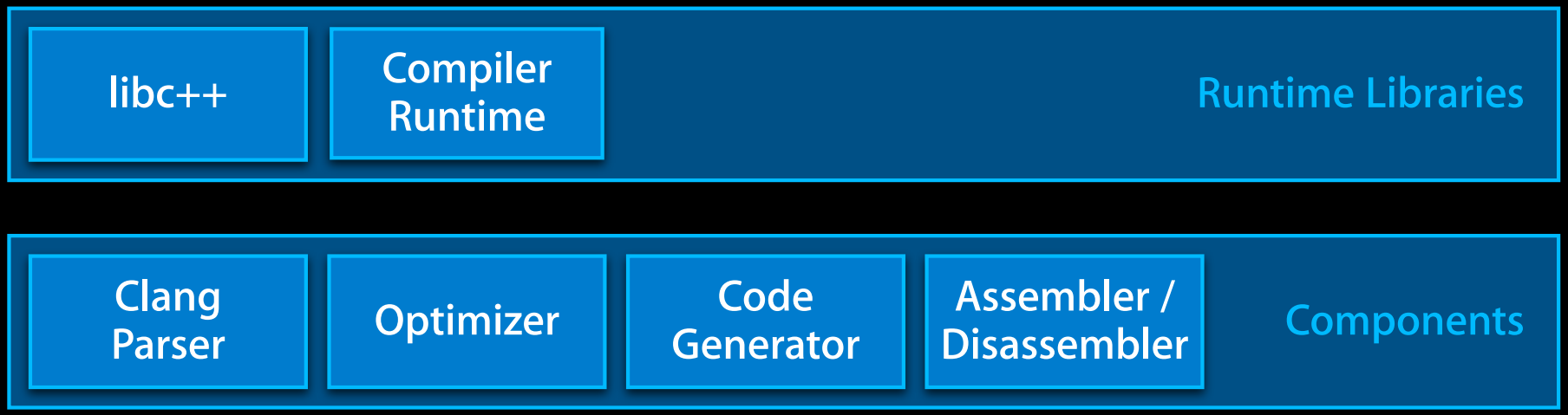
- Performance
  - Optimized code
- Productivity
  - Fast builds
  - New language features
- Quality
  - Compiler warnings
  - Static analysis

Apple LLVM Compiler 4.0

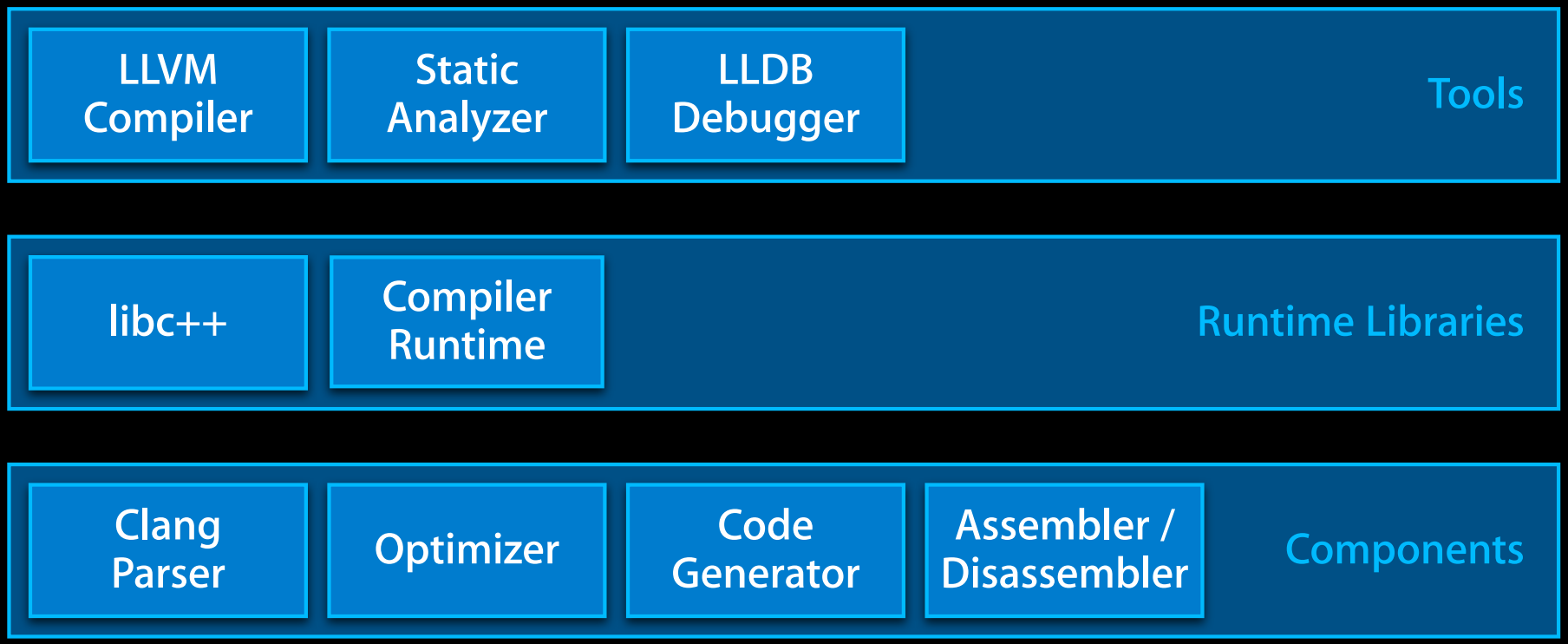
LLVM: Modular Low-Level Tools



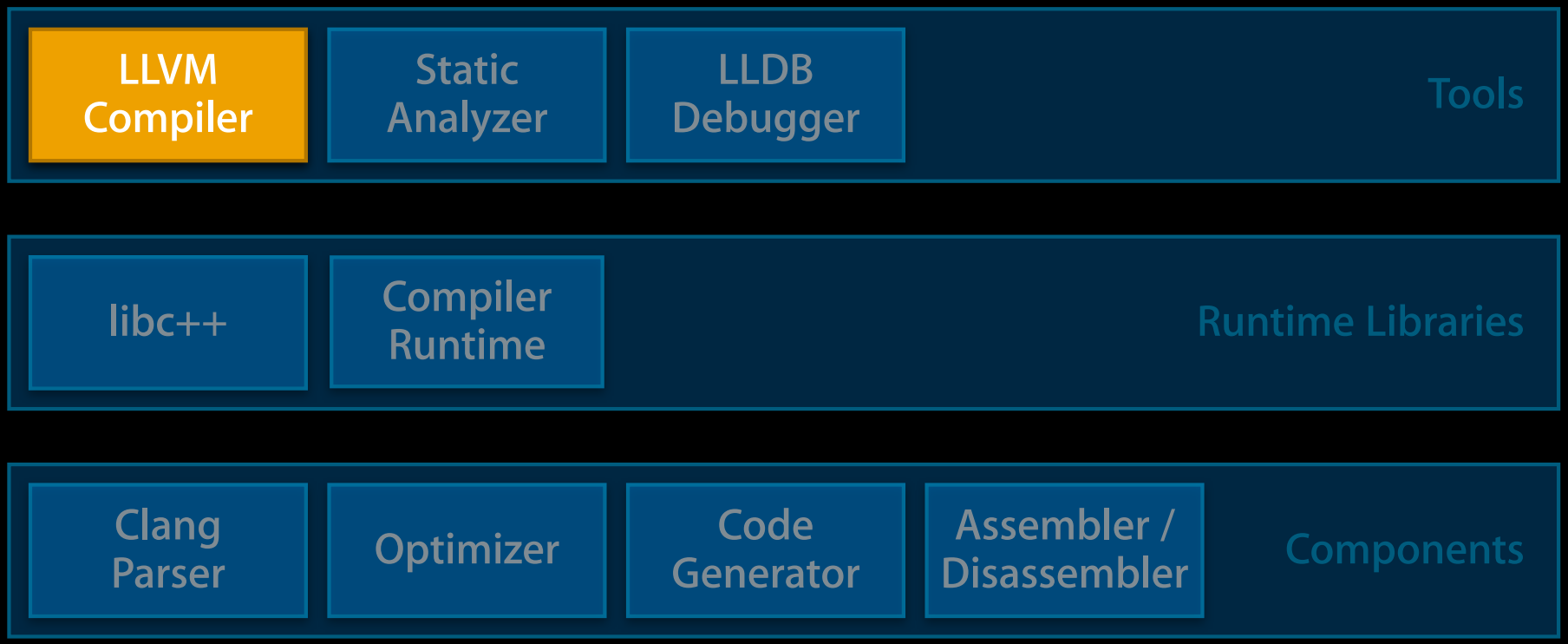
LLVM: Modular Low-Level Tools



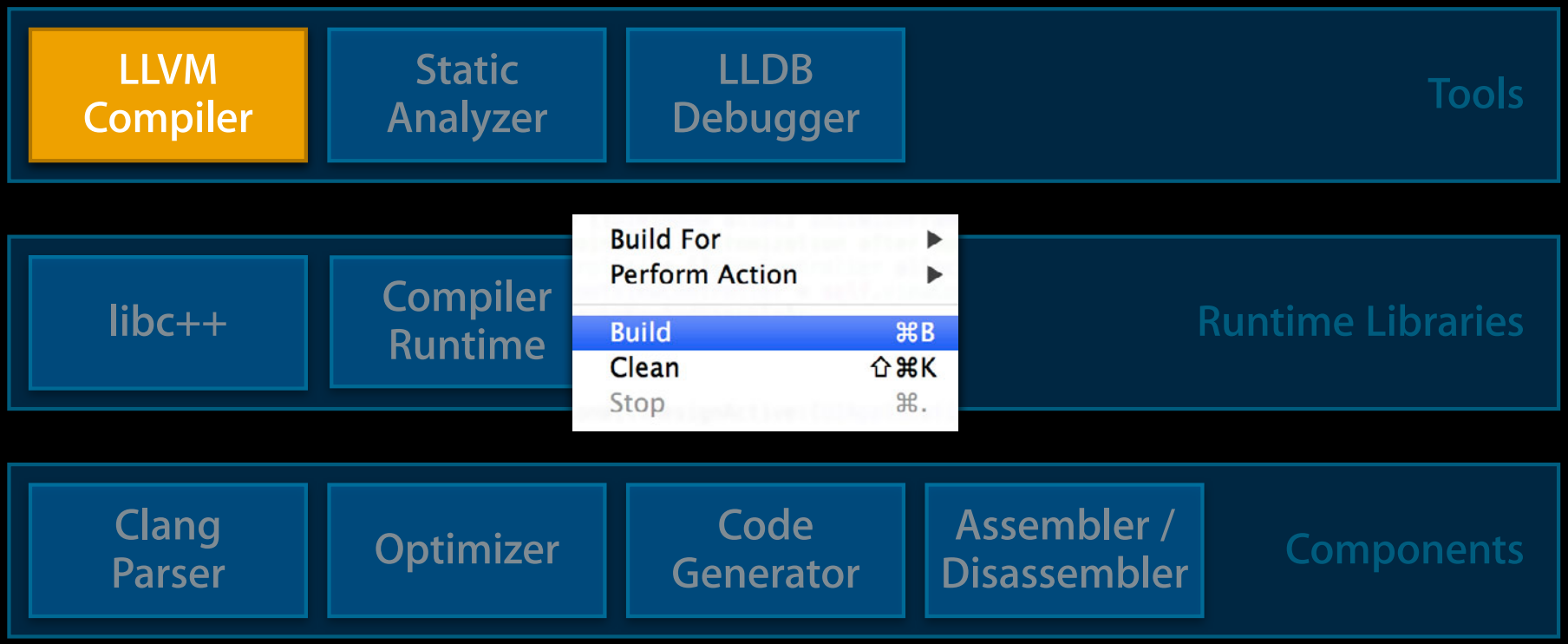
LLVM: Modular Low-Level Tools



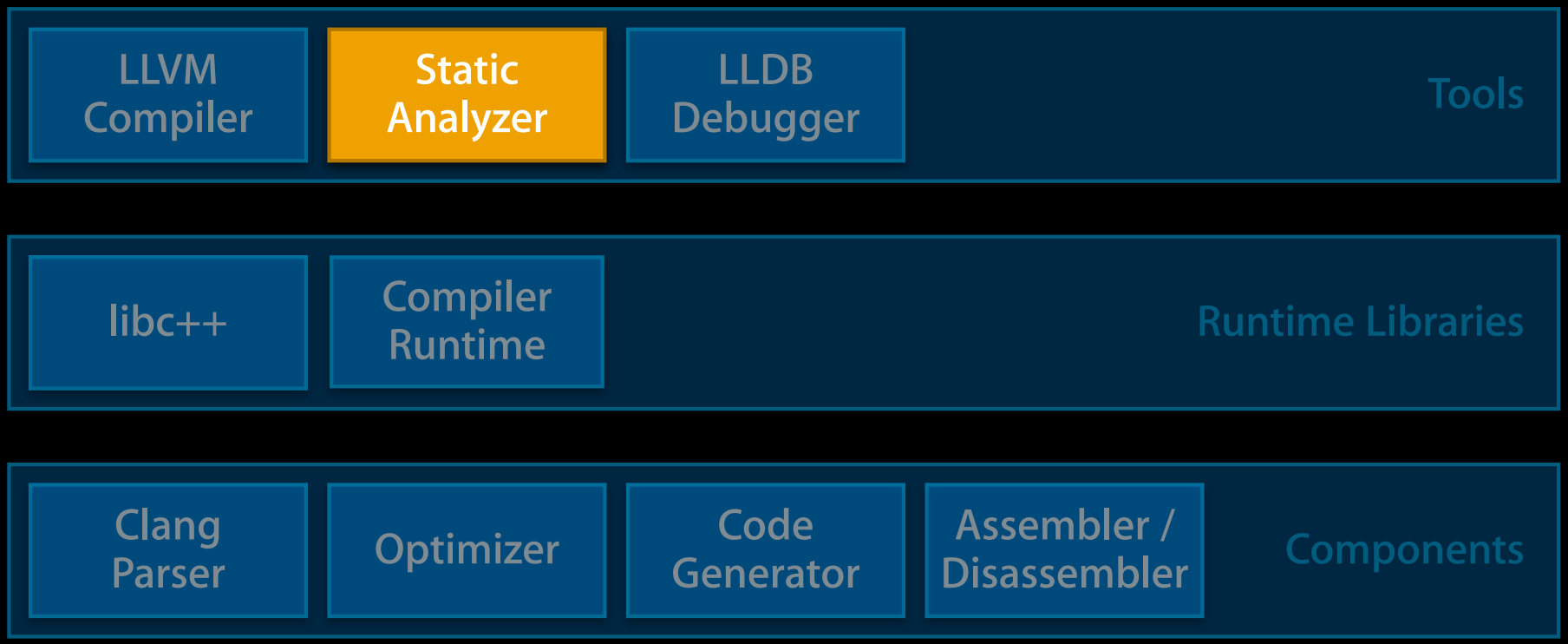
LLVM: Modular Low-Level Tools



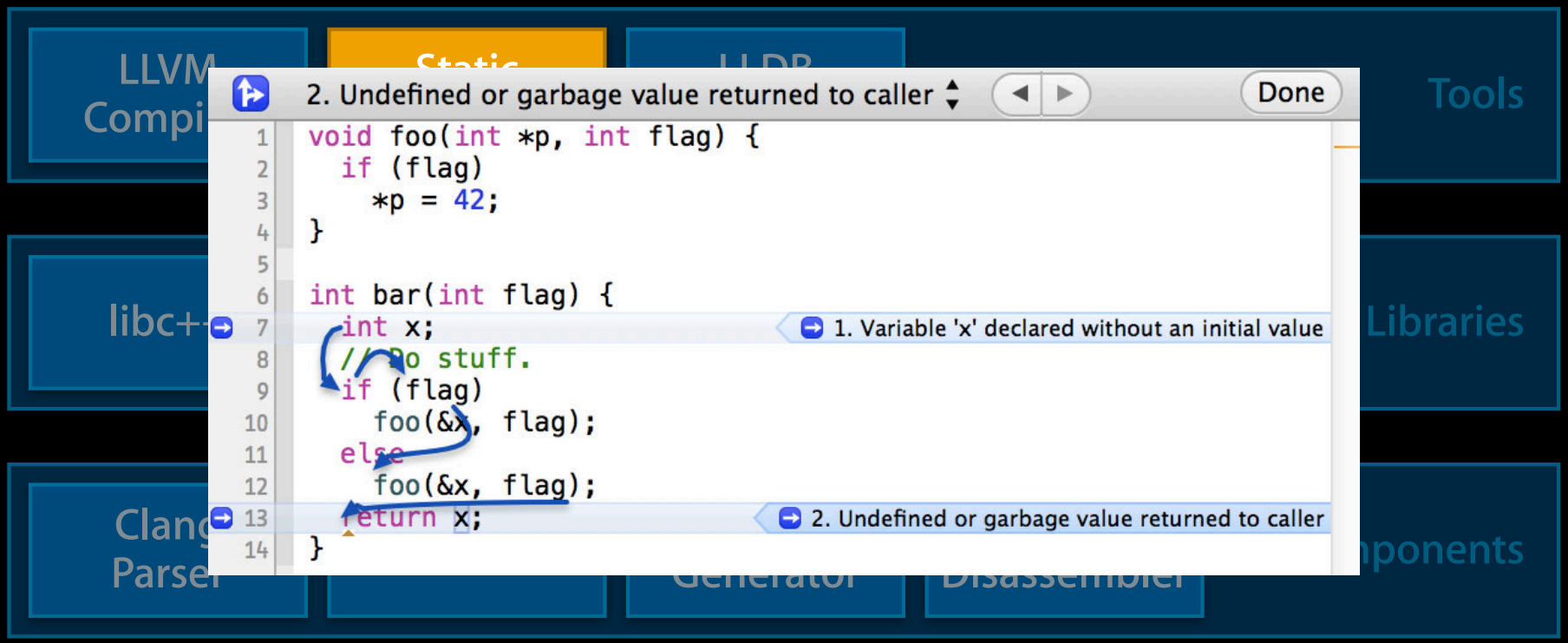
LLVM: Modular Low-Level Tools



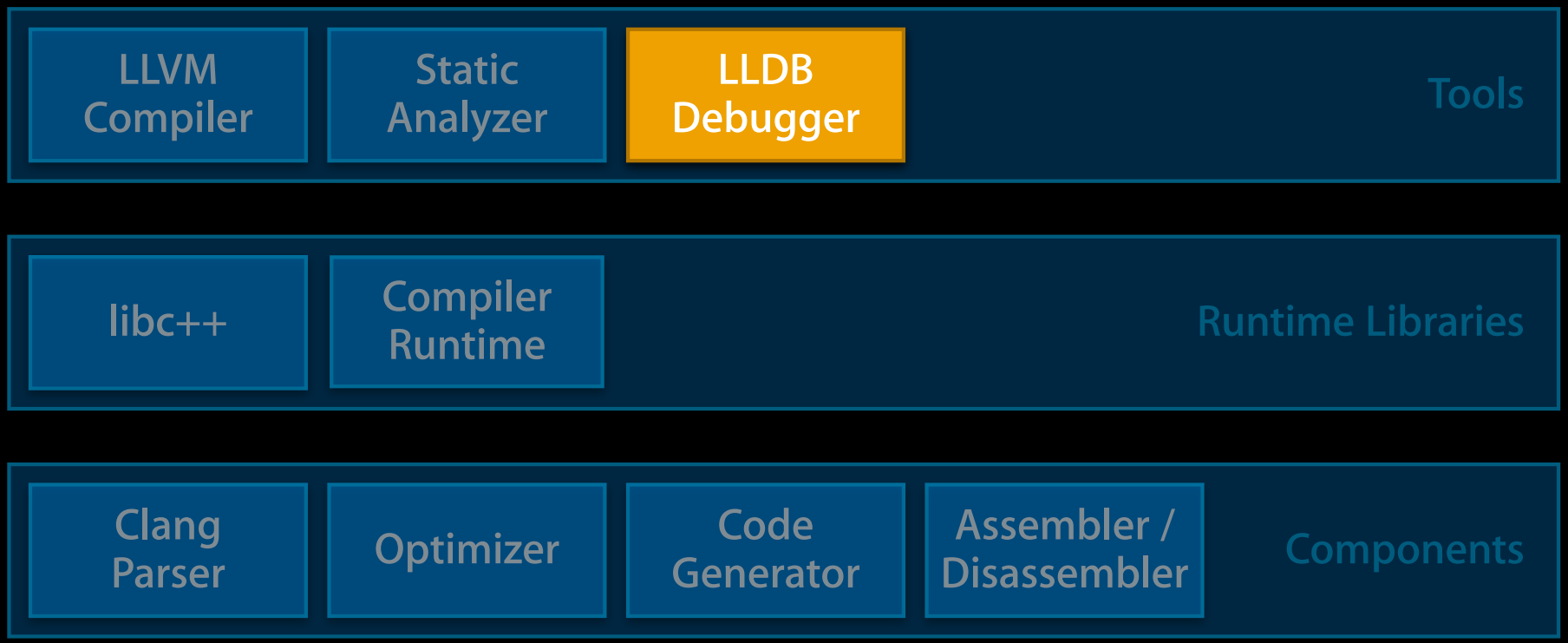
LLVM: Modular Low-Level Tools



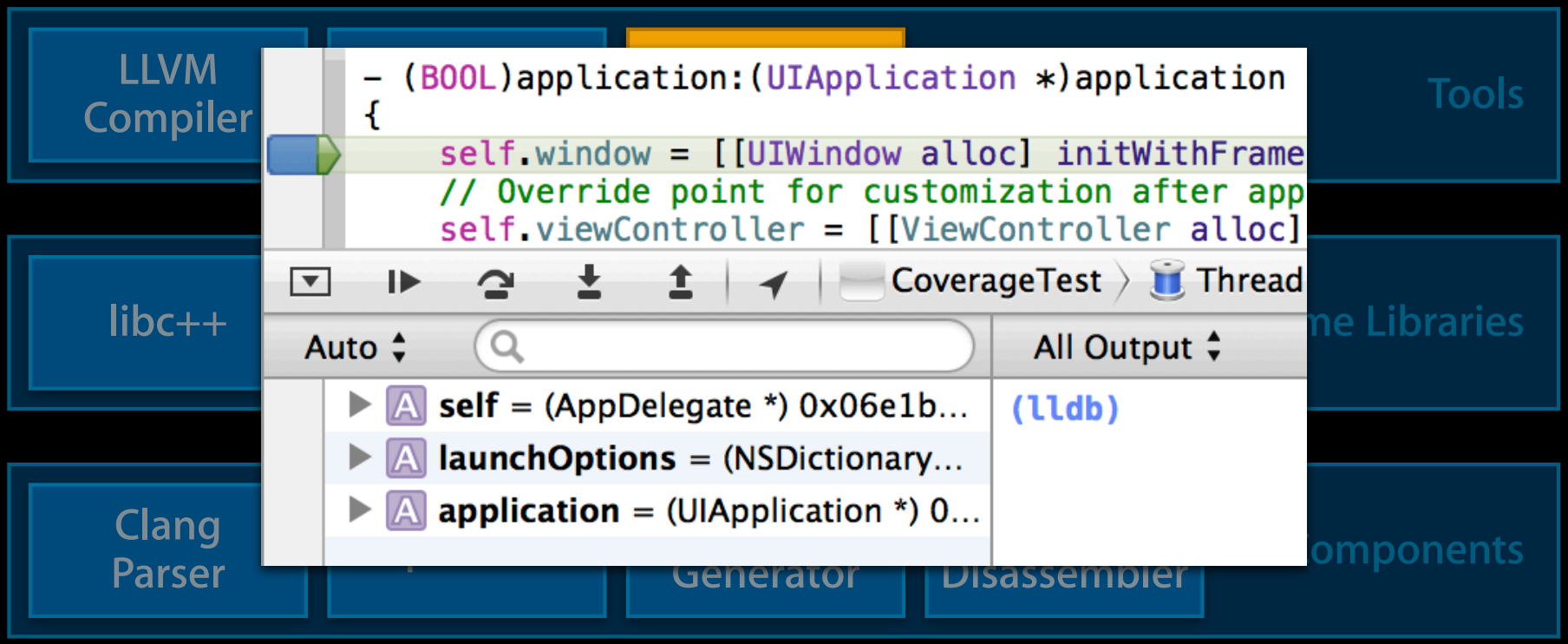
LLVM: Modular Low-Level Tools



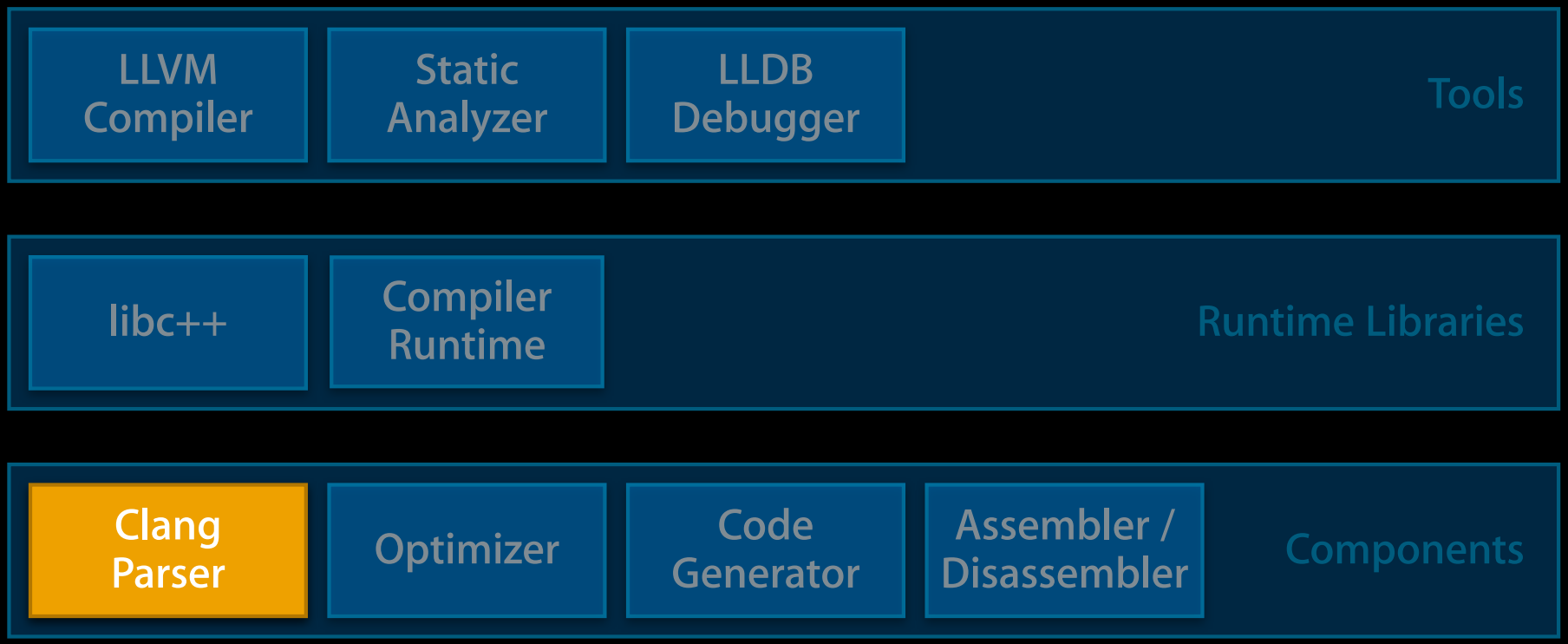
LLVM: Modular Low-Level Tools



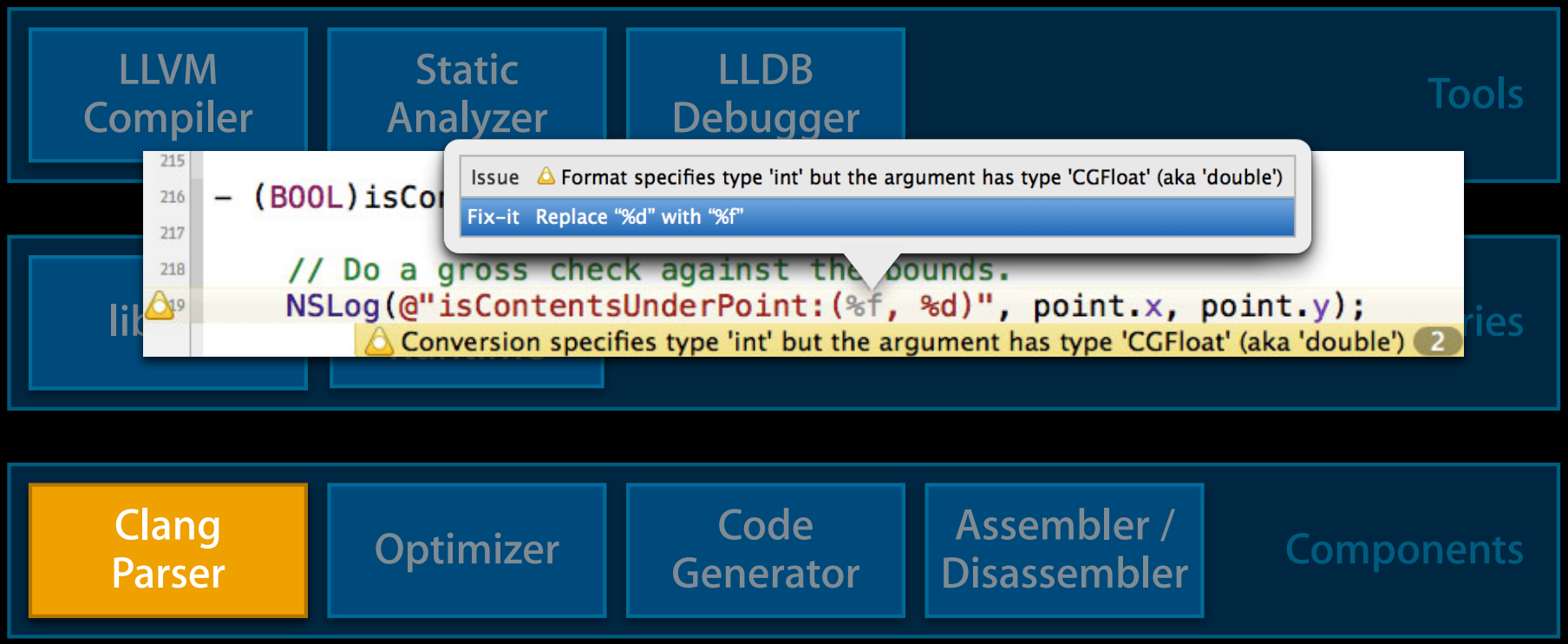
LLVM: Modular Low-Level Tools



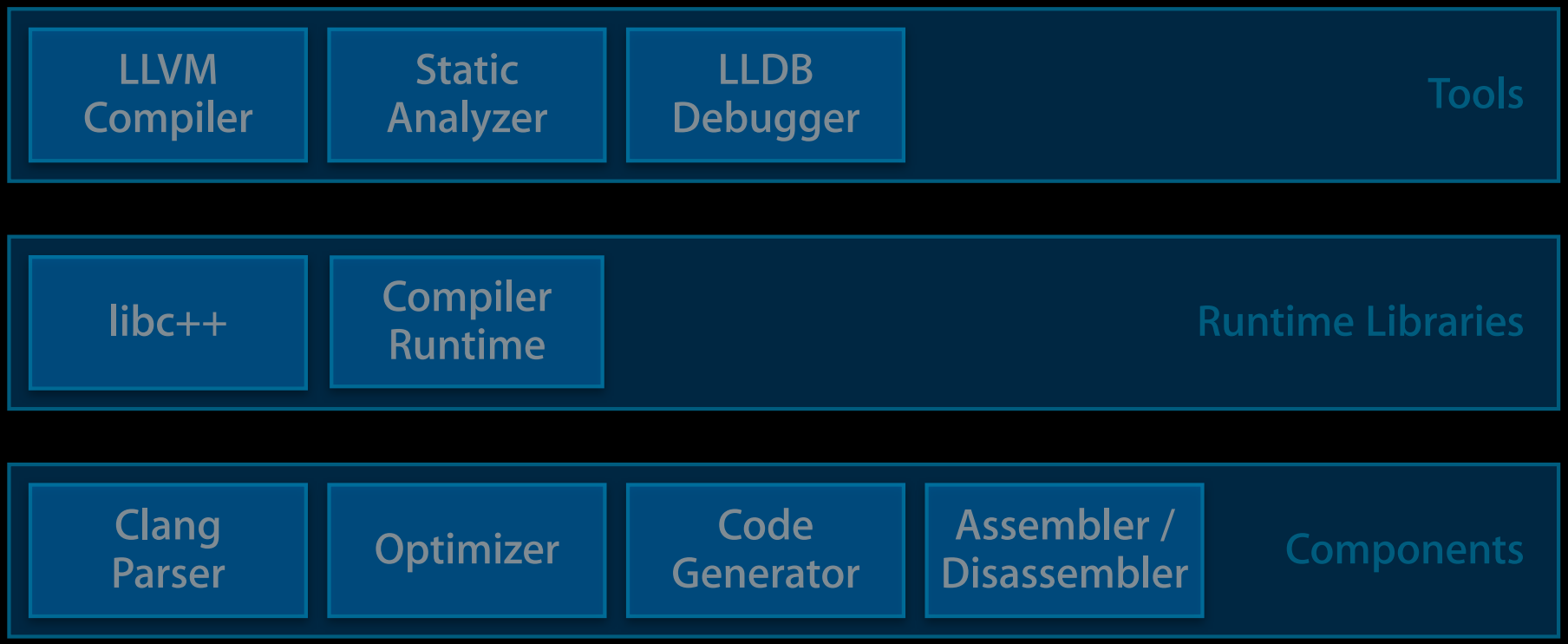
LLVM: Modular Low-Level Tools



LLVM: Modular Low-Level Tools



LLVM: Modular Low-Level Tools



LLVM-GCC: Legacy Compiler

- Backward compatibility with legacy code
  - Based on aging GCC 4.2 parser
  - Uses LLVM optimizer from 2 years ago
- Compiler is frozen at this point
  - No bug fixes
  - No new features
- You need to stop using it now

LLVM-GCC: Legacy Compiler

- Backward compatibility with legacy code
  - Based on aging GCC 4.2 parser
  - Uses LLVM optimizer from Apple branch
- Compiler is frozen at this point
  - No bug fixes
  - No new features
- You need to stop using it now

Will be removed in a future Xcode release!

Performance

ARC Optimizer

- ARC inserts retains and releases conservatively
- Optimizer removes unnecessary retain/release pairs
- A simple example:

```
- (void)debugLog:(NSString *)s {

  NSLog(“Debug: %@\n”, s);

}
```

ARC Optimizer

- ARC inserts retains and releases conservatively
- Optimizer removes unnecessary retain/release pairs
- A simple example:

```
- (void)debugLog:(NSString *)s {
  [s retain]; // inserted automatically
  NSLog("Debug: %@\n", s);
  [s release]; // inserted automatically
}
```

ARC Optimizer

- ARC inserts retains and releases conservatively
- Optimizer removes unnecessary retain/release pairs
- A simple example:

```
- (void)debugLog:(NSString *)s {

  NSLog(“Debug: %@\n”, s);

}
```

ARC Optimizer Enhancements

- Many improvements in Apple LLVM Compiler 4.0
- For example: nested retains

```
- (void)debugLog:(NSString *)s {

  NSString *t = s;

  if (loggingEnabled) {
     [self incrementLogCount]; // may release t
     NSLog("Debug: %@\n", t);
  }


}
```

ARC Optimizer Enhancements

- Many improvements in Apple LLVM Compiler 4.0
- For example: nested retains

```
- (void)debugLog:(NSString *)s {
  [s retain];
  NSString *t = s;

  if (loggingEnabled) {
     [self incrementLogCount]; // may release t
     NSLog("Debug: %@\n", t);
  }

  [s release];
}
```

ARC Optimizer Enhancements

- Many improvements in Apple LLVM Compiler 4.0
- For example: nested retains

```
- (void)debugLog:(NSString *)s {
  [s retain];
  NSString *t = s;
  [t retain];
  if (loggingEnabled) {
     [self incrementLogCount]; // may release t
     NSLog("Debug: %@\n", t);
  }
  [t release];
  [s release];
}
```

ARC Optimizer Enhancements

- Many improvements in Apple LLVM Compiler 4.0
- For example: nested retains

```
- (void)debugLog:(NSString *)s {
  [s retain];
  NSString *t = s;

  if (loggingEnabled) {
     [self incrementLogCount]; // may release t
     NSLog("Debug: %@\n", t);
  }

  [s release];
}
```

ARC Optimizer Enhancements

- Many improvements in Apple LLVM Compiler 4.0
- For example: nested retains

```
- (void)debugLog:(NSString *)s {

  NSString *t = s;
  if (loggingEnabled) {
     [s retain];
     [self incrementLogCount]; // may release t
     NSLog("Debug: %@\n", t);
     [s release];
  }

}
```

Intel AVX

New

Intel AVX

- 256-bit floating-point vector computation
  - Twice as wide as SSE vectors
  - Supported in Sandy Bridge and Ivy Bridge processors

Intel AVX

- 256-bit floating-point vector computation
  - Twice as wide as SSE vectors
  - Supported in Sandy Bridge and Ivy Bridge processors
- Good fit for certain kinds of applications
  - Floating-point intensive
  - High ratio of computation to memory bandwidth

AVX Example: Matrix Addition

- Standard AVX intrinsic functions supported

```
#include <immintrin.h>

void addAVX(int size, float *in1, float *in2, float *out) {
  for (int i = 0; i < size; i += 8) {
    __m256 a = _mm256_load_ps(in1);
    __m256 b = _mm256_load_ps(in2);
    __m256 c = _mm256_add_ps(a, b);
    _mm256_store_ps(out, c);
    in1 += 8; in2 += 8; out += 8;
  }
}
```

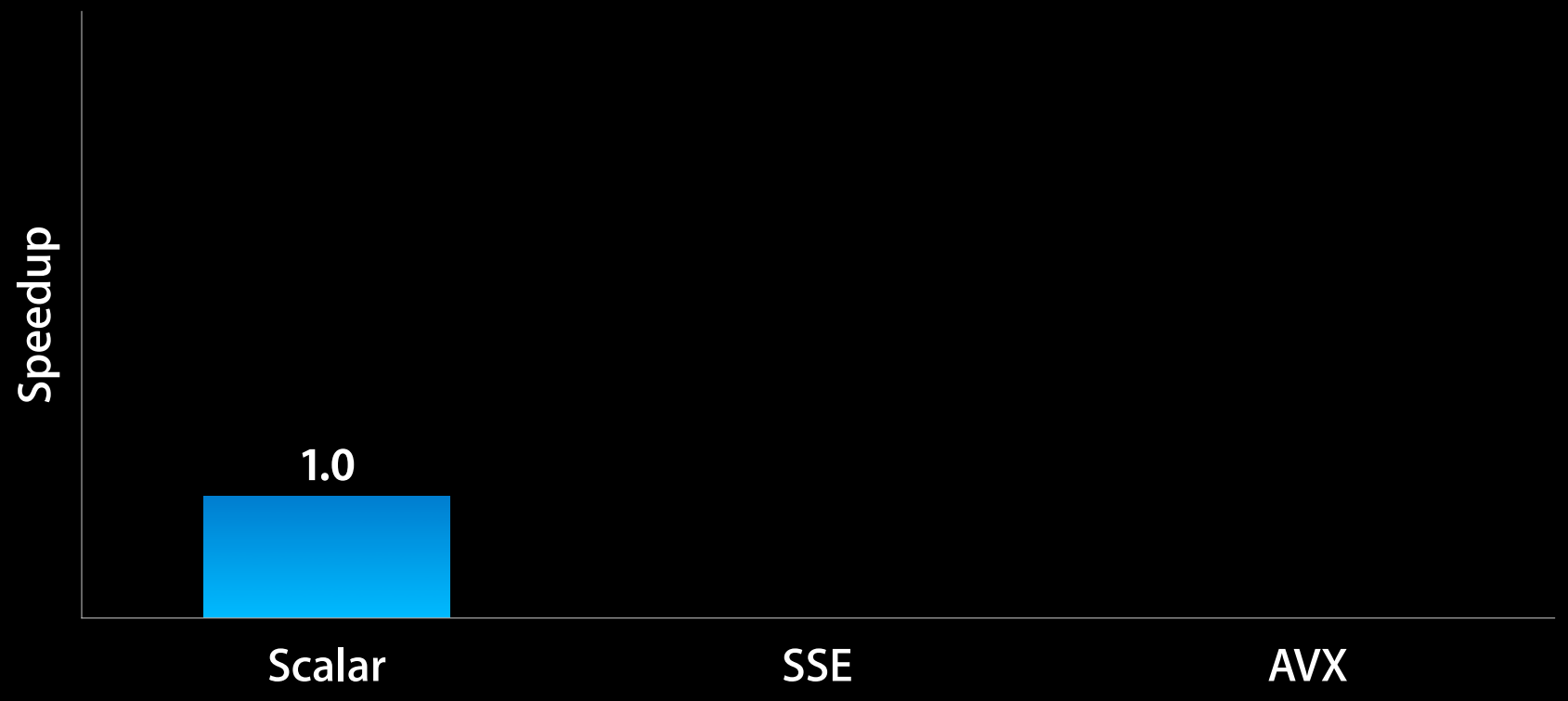
AVX Example: Matrix Addition

- Standard AVX intrinsic functions supported
- OpenCL vector syntax also works

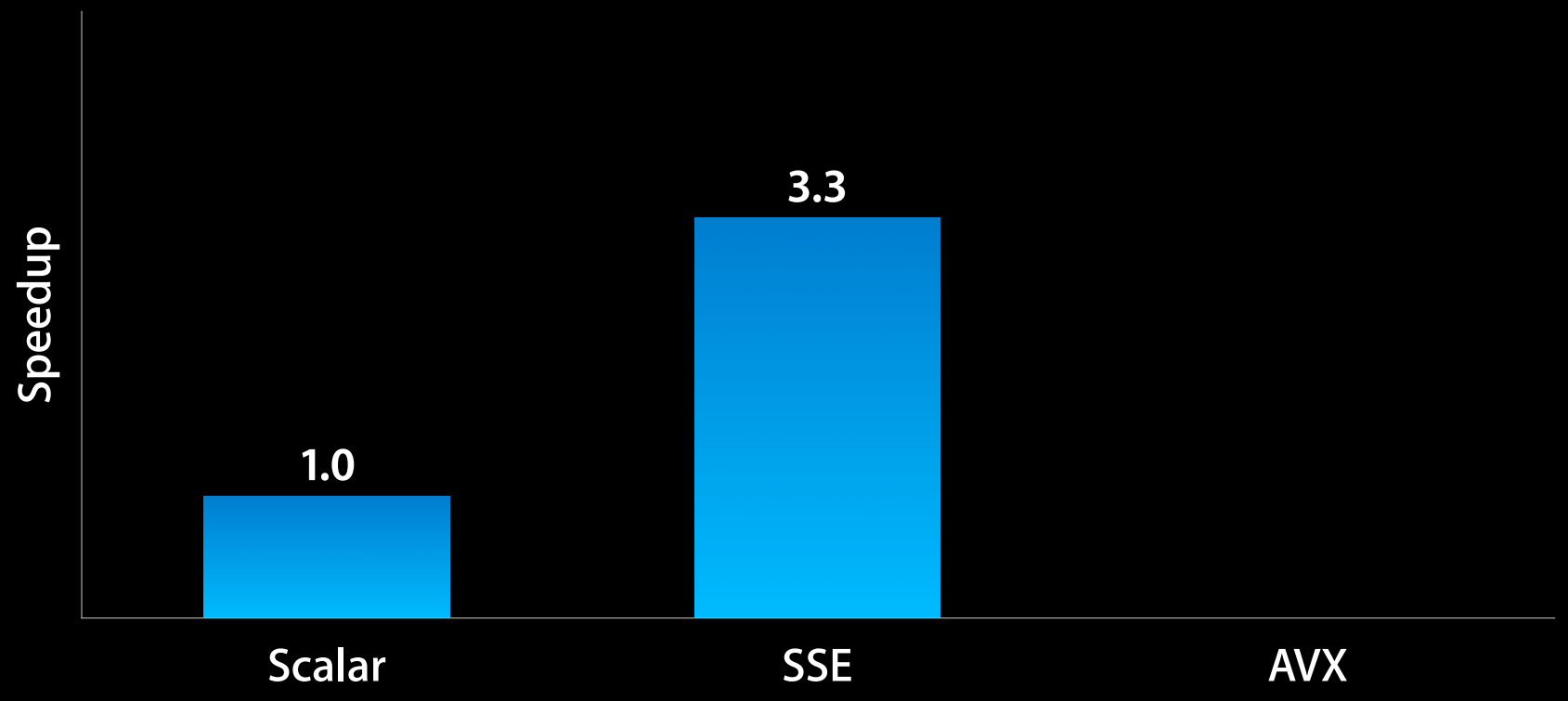
```
#include <immintrin.h>

void addAVX(int size, float *in1, float *in2, float *out) {
  for (int i = 0; i < size; i += 8) {
    __m256 a = *(__m256 *)in1;
    __m256 b = *(__m256 *)in2;
    __m256 c = a + b;
    *(__m256 *)out = c;
    in1 += 8; in2 += 8; out += 8;
  }
}
```

AVX Example Performance

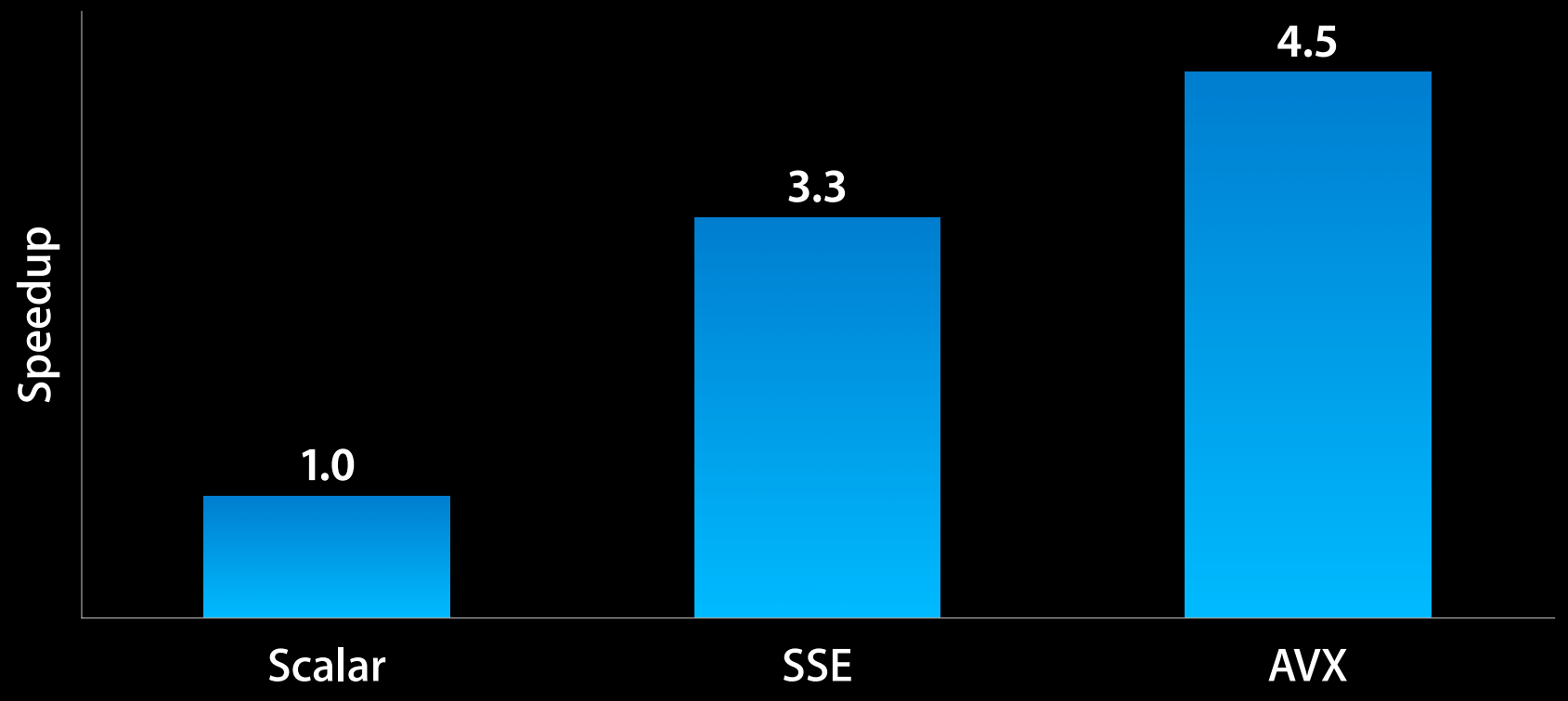
AVX Example Performance



AVX Example Performance



AVX Example Performance



Using AVX with Fallback to SSE

- Check at runtime if AVX is supported
- Put AVX code in separate files to be compiled with `-mavx` option
- Provide an alternate version using SSE

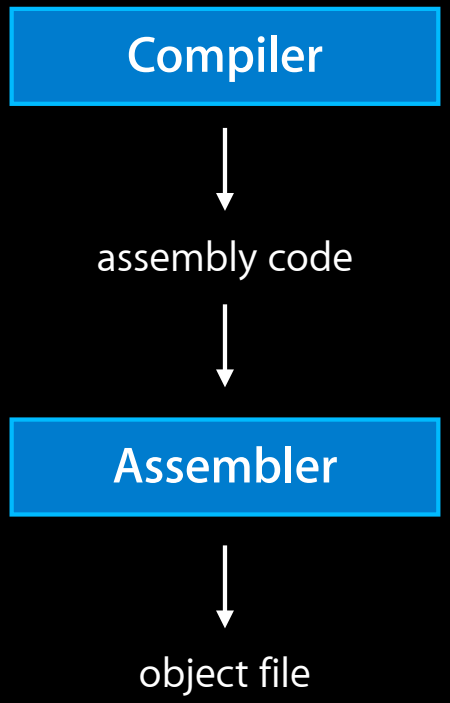
```
#include <sys/sysctl.h>

void add(int size, float *in1, float *in2, float *out) {
  int answer = 0;
  size_t length = sizeof(answer);
  if (!sysctlbyname("hw.optional.avx1_0", &answer, &length, NULL, 0) &&
      answer != 0)
    addAVX(size, in1, in2, out);
  else
    addSSE(size, in1, in2, out);
}
```

Integrated ARM Assembler

- Old way: write assembly code and invoke the assembler
- Now we generate object files directly
- Better error checking for inline assembly code
- Only supports ARM "unified syntax"

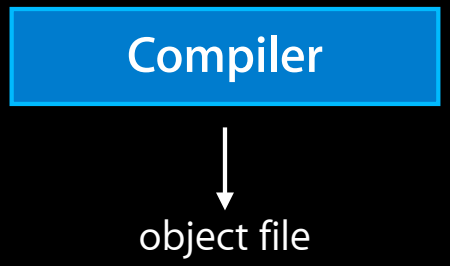
Integrated ARM Assembler

- Old way: write assembly code and invoke the assembler
- Now we generate object files directly
- Better error checking for inline assembly code
- Only supports ARM "unified syntax"



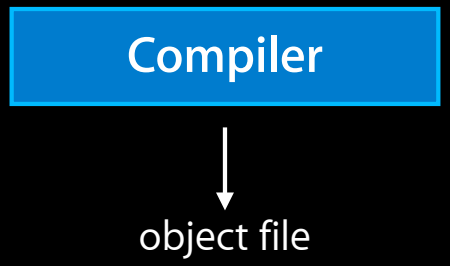
Integrated ARM Assembler

- Old way: write assembly code and invoke the assembler
- Now we generate object files directly



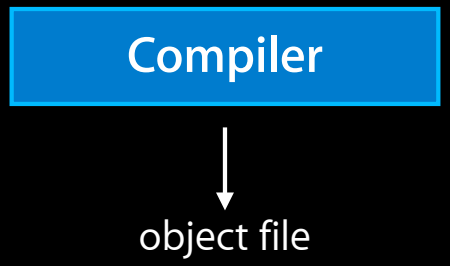
Integrated ARM Assembler

- Old way: write assembly code and invoke the assembler
- Now we generate object files directly
- Better error checking for inline assembly code



Integrated ARM Assembler

- Old way: write assembly code and invoke the assembler
- Now we generate object files directly
- Better error checking for inline assembly code
- Only supports ARM "unified syntax"



New Language Features

Doug Gregor
Senior Engineer, Compiler Frontend Team

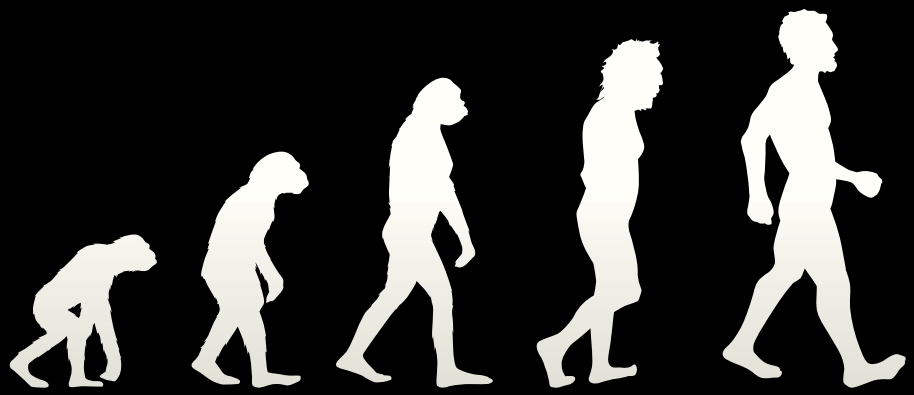
New Objective-C Language Features

- Numeric literals `@3.14159`
- Array literals `@[@1, @2]`
- Dictionary literals `@{@"Red" : @1, @"Green" : @2}`
- Boxed expressions `@(x + y)`
- Default synthesis of properties
- Order-independent `@implementation`

C++11

The 2011 C++ Standard

- Approximately 13 years of language and library evolution
  - Simplify common idioms
  - Improve performance
  - Improve support for writing libraries
- Strong focus on backward compatibility



C++11 Features: LLVM Compiler 3.0

C++11 Features: LLVM Compiler 3.0

deleted functions

override controls

static assertions

noexcept

'auto' typed variables

variadic templates

strongly typed enums

extended SFINAE

rvalue references

range-based for loop

C++11 Features: LLVM Compiler 3.1

deleted functions

override controls

static assertions

noexcept

'auto' typed variables

variadic templates

strongly typed enums

extended SFINAE

rvalue references

range-based for loop

C++11 Features: LLVM Compiler 3.1

deleted functions

override controls

delegating constructors

implicit move constructors

static assertions

noexcept

'auto' typed variables

non-static data member initializers

variadic templates

strongly typed enums

explicit conversions

extended SFINAE

defaulted functions

rvalue references

range-based for loop

C++11 Features: LLVM Compiler 4.0

deleted functions

override controls

delegating constructors

implicit move constructors

static assertions

noexcept

'auto' typed variables

non-static data member initializers

variadic templates

strongly typed enums

extended SFINAE

explicit conversions

defaulted functions

rvalue references

range-based for loop

C++11 Features: LLVM Compiler 4.0

deleted functions

override controls

generalized initializer lists

delegating constructors

implicit move constructors

static assertions

noexcept

'auto' typed variables

non-static data member initializers

atomics

variadic templates

strongly typed enums

extended SFINAE

explicit conversions

defaulted functions

lambda expressions

rvalue references

range-based for loop

generalized constant expressions

C++11 Standard Library

Needed for great C++11 support

- C++11 language features depend on library features:
  - Initializer lists
  - Move semantics
  - Generalized constant expressions
- C++11 provides new library components:
  - Smart pointers (`unique_ptr/shared_ptr/weak_ptr`)
  - Regular expressions
  - Threading and atomics
  - And much more

libc++ : C++11 Standard Library

libc++

- Standards-conformant implementation of the C++11 library
  - Backward compatible with C++98/03 applications
- Engineered from the ground up for performance
- Replaces the existing GCC standard library (libstdc++)

Migrating to libc++

libc++

- libc++ and C++11 are largely backward compatible

Migrating to libc++

libc++

- libc++ and C++11 are largely backward compatible
- C++ Library TR1 components have moved into C++11
  - Headers have moved from `<tr1/header>` to `<header>`
  - Components have moved from namespace `std::tr1` to `std`

```
#include <tr1/unordered_map>
std::tr1::unordered_map<int, int> m;
```

→

```
#include <unordered_map>
std::unordered_map<int, int> m;
```

Variable Creation Is Verbose

```
vector<NSView *> views;

/* update views... */
for (vector<NSView *>::iterator v = views.begin(), vend = views.end();
     v != vend; ++v) {
  [*v setNeedsDisplay:YES];
}
```

auto Variables Infer Type

```
vector<NSView *> views;

/* update views... */
for (auto                    v = views.begin(), vend = views.end();
     v != vend; ++v) {
  [*v setNeedsDisplay:YES];
}
```

auto Variables Infer Type

```
vector<NSView *> views;

/* update views... */
for (auto v = views.begin(), vend = views.end();
     v != vend; ++v) {
  [*v setNeedsDisplay:YES];
}
```

auto Variables Infer Type

```
vector<NSView *> views;

/* update views... */
for (auto v = views.begin(), vend = views.end();
     v != vend; ++v) {
  [*v setNeedsDisplay:YES];
}
```

- By default, `auto` variables copy the value
- Reference to `auto` creates a reference:

```
auto &first = views.front();
```

For-Range Loop

```
vector<NSView *> views;

/* update views... */
for (auto view : views) {
  [view setNeedsDisplay:YES];
}
```

- Like fast enumeration, with `:` rather than `in`
- Loop over anything with `begin` and `end` functions

Be Wary of `auto` with `id`

```
- (void)method:(NSArray *)views {
  NSView *view = [views objectAtIndex:0];
  // Use view
}
```

Be Wary of `auto` with `id`

```
- (void)method:(NSArray *)views {
  auto view = [views objectAtIndex:0];
  // Use view
}
```

Be Wary of `auto` with `id`

```
- (void)method:(NSArray *)views {
  /*auto=*/id view = [views objectAtIndex:0];
  // Use view
}
```

Be Wary of `auto` with `id`

```
- (void)method:(NSArray *)views {
  /*auto=*/id view = [views objectAtIndex:0];
  // Use view
}
```

- Expected type of `view` (`NSView *`) differs from inferred type (`id`)

Be Wary of `auto` with `id`

```
- (void)method:(NSArray *)views {
  /*auto=*/id view = [views objectAtIndex:0];
  // Use view
}
```

- Expected type of `view` (`NSView *`) differs from inferred type (`id`)
- `id` provides less static type information
  - Compiler won't warn if we convert the view to an `NSString*`
  - Code completion shows **all** known methods
- `auto` is still perfectly safe for C++ types

Initializing Containers Is Painful

```
vector<NSString *> colors;
colors.push_back(@”Red”);
colors.push_back(@”Green”);
colors.push_back(@”Blue”);
```

Generalized Initializer Lists


```
vector<NSString *> colors = {
  @"Red", @"Green", @"Blue"
};
```

Generalized Initializer Lists


```
vector<NSString *> colors = {
  @"Red", @"Green", @"Blue"
};
```

- An initializer list `{ … }` can be used with any C++ container

```
map<string, NSView *> views = {
  { "MyView", myView }
};
```

Generalized Initializer Lists


```
vector<NSString *> colors = {
  @"Red", @"Green", @"Blue"
};
```

- An initializer list `{ … }` can be used with any C++ container

```
map<string, NSView *> views = {
  { "MyView", myView }
};
```

- Also works for inserting values into maps

```
views.insert({"OtherView", otherView});
```

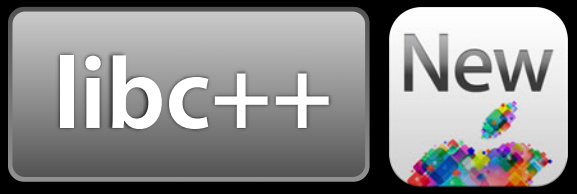
Multiple Return Values

Multiple Return Values

- Tuples make it easy to return multiple values

```
tuple<int, int> minmax(int x, int y) {
  if (x <= y) return { x, y };
  return { y, x };
}
```

Multiple Return Values



- Tuples make it easy to return multiple values

```
tuple<int, int> minmax(int x, int y) {
  if (x <= y) return { x, y };
  return { y, x };
}
```

- `tie` lets you bind multiple return values to different variables

```
int a, b;
tie(a, b) = minmax(m, n);
```

Lambda Expressions


```
vector<NSString *> strings;
sort(strings.begin(), strings.end(),
     ](NSString *x, NSString *y) -> bool {
       return [x localizedCaseInsensitiveCompare:y]
         == NSOrderedAscending;
     });
```

Lambda Expressions


```
vector<NSString *> strings;
sort(strings.begin(), strings.end(),
     [](NSString *x, NSString *y) -> bool {
       return [x localizedCaseInsensitiveCompare:y]
         == NSOrderedAscending;
     });
```

- Anonymous function objects (closures)
- Similar to blocks:
  - `[]` introduces a lambda
  - `-> type` optionally specifies return type
  - We'll talk about some of the differences and interactions

Capture Semantics in Blocks

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  ^ {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

Capture Semantics in Blocks

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  ^ {
      sort(strings.begin(), strings.end(), /* compare strings */); error!
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- `strings` variable is "captured" by the block **by value**
  - By-value captured variables are copied when the block is copied
  - By-value captured variables are `const`; `strings` cannot be sorted
- Objective-C objects captured by value are automatically retained

__block Capture Semantics in Blocks

```
__block vector<NSString *> strings;
dispatch_async(sort_queue,
  ^ {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- `strings` variable is "captured" by the block **by reference**
  - All blocks that refer to `strings` see the same value of `strings`
  - By-reference captured variables are still copied once

Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [] {
      sort(strings.begin(), strings.end(), /* compare strings */); error!
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- `[]` represents an empty capture list

By-value Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [strings] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

By-value Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [strings] {
      sort(strings.begin(), strings.end(), /* compare strings */); error!
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- List variable names to be captured by value
  - Variables will be copied into the lambda
  - Copied variables are treated as `const`
- Objective-C objects captured by value are **not** automatically retained (except under ARC)

By-reference Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [&strings] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

By-reference Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [&strings] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- Variable names preceded by `&` indicate capture by reference
  - Variables will **never be copied**
  - Referenced variables will be modified

By-reference Lambda Captures

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [&strings] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- Variable names preceded by `&` indicate capture by reference
  - Variables will **never be copied**
  - Referenced variables will be modified
- Caution: Dangling pointer if the lambda outlives its by-reference captures

Lambda Capture Defaults

```
vector<NSString *> strings;
dispatch_async(sort_queue,
  [&] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
// Do more work concurrently
dispatch_sync(sort_queue, ^{}); // wait for string sorting to finish
```

- `[&]` means capture everything by reference
- `[=]` means capture everything by value
- Can still list exceptions to the default rule:
  - `[=, &strings]`

Interoperating with Blocks in Objective-C++

```
dispatch_async(sort_queue,
  [&] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
```

Interoperating with Blocks in Objective-C++

```
dispatch_async(sort_queue,
  [&] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
```

- Lambdas can be used with blocks-based APIs
  - A lambda can be implicitly converted to a block
  - Parameter types and return type must match

Interoperating with Blocks in Objective-C++

```
dispatch_async(sort_queue,
  [&] {
     sort(strings.begin(), strings.end(), /* compare strings */);
  });
```

- Lambdas can be used with blocks-based APIs
  - A lambda can be implicitly converted to a block
  - Parameter types and return type must match
- Returned block is retain/autoreleased
  - Compiler may optimize away this retain/autorelease pair

Comparing Blocks and Lambdas

	Blocks	Lambdas
Capture by Copy	Yes (default)	Yes (`[=]` or `[var]`)
Capture by Copy Retains Objects	Yes	Requires ARC
Capture by Reference	Yes (`__block`)	Yes (`[&]`, `[&var]`)
Capture by Reference Cannot Dangle	Yes	No
Works with Block APIs	Yes	Requires Objective-C++

When Should I Use Lambdas?

- For Objective-C++ code, you should generally use blocks
  - Succinct and well understood by Objective-C developers
  - Safer (retains objects, does not let by-reference captures dangle)
- Use lambdas if:
  - You are in a portable C++11 code base
  - You need precise control over how variables are captured
  - You are using templates and want to eliminate call overhead

C++11 Deployment

C++11 Deployment

- C++11 available now
  - Deploys back to iOS 5.0, OS X v10.7

C++11 Deployment

- C++11 available now
  - Deploys back to iOS 5.0, OS X v10.7
- Defaults moving toward C++11
  - C++11 language default for new projects in Xcode 4.4
  - libc++ default for new projects in Xcode 4.5

C++11 Deployment

- C++11 available now
  - Deploys back to iOS 5.0, OS X v10.7
- Defaults moving toward C++11
  - C++11 language default for new projects in Xcode 4.4
  - libc++ default for new projects in Xcode 4.5

Apple LLVM compiler 4.0 – Language	
'char' Type Is Unsigned	No
Allow 'asm', 'inline', 'typeof'	Yes
C Language Dialect	Compiler Default
C++ Language Dialect	**GNU++11 [-std=gnu++11]**
C++ Standard Library	**libc++ (LLVM C++ standard library with C++11 support)**

```
xcrun clang++ -std=gnu++11 -stdlib=libc++ hello.cpp
```

Finding Bugs Early

Ted Kremenek
Manager, Compiler Frontend Team

Users Want Quality Apps

Users Want Quality Apps

“routinely crashes”

“full of bugs”

“really frustrated”

“really unstable”

Finding Issues Early with Warnings

Finding Issues Early with Warnings

- Warnings help find bugs early
  - Clear and explanatory diagnostics
  - Provides suggestions for fixes

Finding Issues Early with Warnings

- Warnings help find bugs early
  - Clear and explanatory diagnostics
  - Provides suggestions for fixes
- Improvements in Xcode 4.4:
  - Deeper static analysis
  - New compiler warnings and analyzer checks
  - New ways to control warnings

Trading CPU Resources for Finding Issues

Trading CPU Resources for Finding Issues

- Always available
- Catches bugs early
- Fast and shallow analysis

Trading CPU Resources for Finding Issues

- Always available
- Catches bugs early
- Fast and shallow analysis

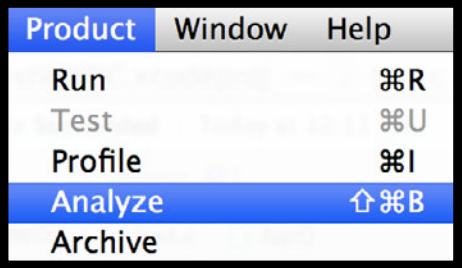
Trading CPU Resources for Finding Issues

Compiler

- Always available
- Catches bugs early
- Fast and shallow analysis

Static Analyzer

- Run on demand



- Takes longer with deeper analysis
- Finds hard-to-detect bugs
- Understands common APIs

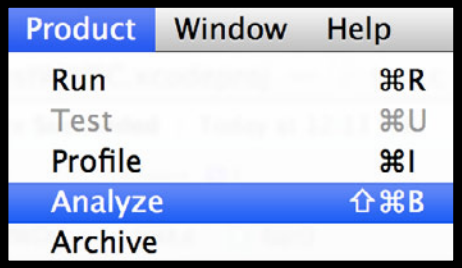
Trading CPU Resources for Finding Issues

Compiler

- Always available
- Catches bugs early
- Fast and shallow analysis

Static Analyzer

- Run on demand



- Takes longer with deeper analysis
- Finds hard-to-detect bugs
- Understands common APIs

Differences in Analysis Power

```
int bar(int flag) {
  int x;
  // Do stuff, but forget to assign to ‘x’.
  return x;
}
```

Differences in Analysis Power

```
int bar(int flag) {
  int x;
  // Do stuff, but forget to assign to ‘x’.
  return x;
}
```


```
warning: variable 'x' is uninitialized when used here [-Wuninitialized]
    return x;
           ^
note: initialize the variable 'x' to silence this warning
    int x;
         ^
          = 0
```

Differences in Analysis Power

```
void foo(int *p, int flag) {
  if (flag) {
    *p = 42;
  }
  return;
}

int bar(int flag) {
  int x;
  // Do stuff.
  if (flag) {
    foo(&x, flag);
  }
  else {
    foo(&x, flag);
  }
  return x;
}
```

Differences in Analysis Power

```
void foo(int *p, int flag) {
  if (flag) {
    *p = 42;
  }
  return;
}

int bar(int flag) {
  int x;
  // Do stuff.
  if (flag) {
    foo(&x, flag);
  }
  else {
    foo(&x, flag);
  }
  return x;
}
```


```
$ xcrun clang -c -Wall test.c
$
```

No issue found

Differences in Analysis Power

```
void foo(int *p, int flag) {
  if (flag) {
    *p = 42;
  }
  return;
}

int bar(int flag) {
  int x;
  // Do stuff.
  if (flag) {
    foo(&x, flag);
  }
  else {
    foo(&x, flag);
  }
  return x;
}
```

Differences in Analysis Power

```
void foo(int *p, int flag) {
  if (flag) {
    *p = 42;
}

in

}
```

6. Undefined or garbage value returned to caller

Done

```
void foo(int *p, int flag) {
  if (flag) {
    *p = 42;
  }
  return;
}

int bar(int flag) {
  int x;
  // Do stuff.
  if (flag) {
    foo(&x, flag);
  }
  else {
    foo(&x, flag);
  }
  return x;
}
```

4. Entered call from 'bar'

1. Variable 'x' declared without an initial value

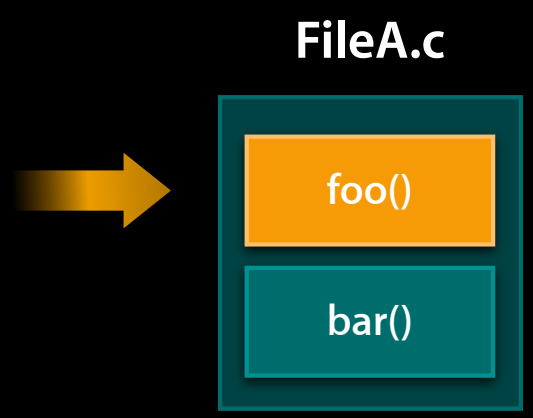
2. Assuming 'flag' is 0

3. Calling 'foo' 2

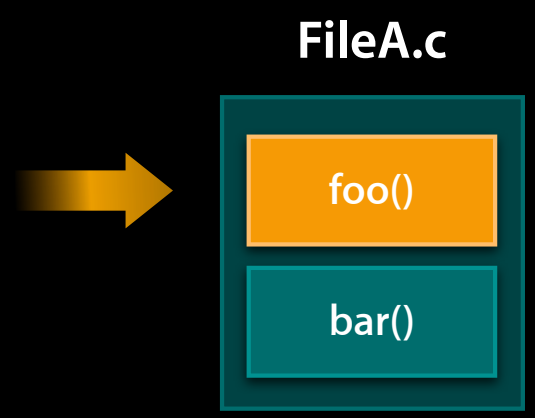
6. Undefined or garbage value returned to caller

Single Function Analysis in Xcode 4.3

Single Function Analysis in Xcode 4.3

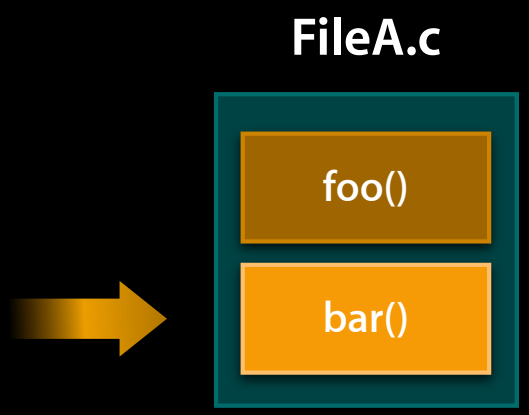


Single Function Analysis in Xcode 4.3




```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

Single Function Analysis in Xcode 4.3



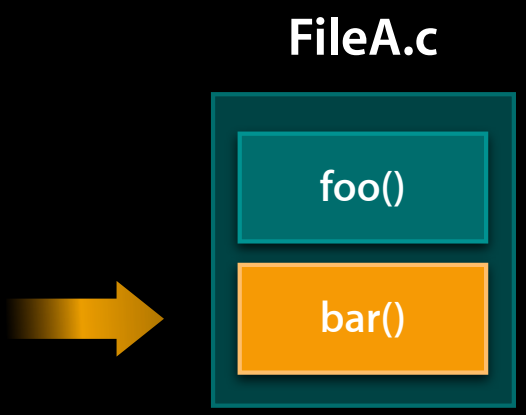

```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

Cross Function Analysis in Xcode 4.4

FileA.c

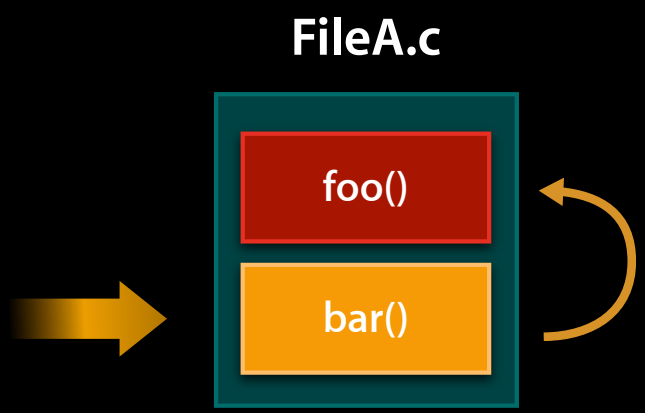

```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

Cross Function Analysis in Xcode 4.4




```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

Cross Function Analysis in Xcode 4.4



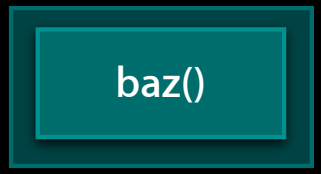

```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

Cross Function Analysis in Xcode 4.4

FileA.c


```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

FileB.c




```
void baz() {
  foo(NULL);
}
```

Cross Function Analysis in Xcode 4.4

FileA.c


```
void foo(int *p) {
  *p = 0xDEADBEEF;
}

void bar() {
  foo(NULL);
}
```

FileB.c


```
void baz() {
  foo(NULL);
}
```

New Warnings

Memory Safety and Security

Objective-C Format String Checking

```
- (NSString)fontDescription:(NSTextField *)fontField size:(NSInteger)size {
  return [NSString stringWithFormat:@"Name: %s, size: %ld pt",
                   fontField.stringValue, size];
}
```

Objective-C Format String Checking

```
- (NSString)fontDescription:(NSTextField *)fontField size:(NSInteger)size {
  return [NSString stringWithFormat:@”Name: %s, size: %ld pt”,
                   fontField.stringValue, size];
}
```


```
warning: format specifies type 'char *' but the argument has type
'NSString *' [-Wformat]

  ...stringWithFormat:@"Font: %s, size: %d point", fontField.stringValue,
                              ~^                   ~~~~~~~~~~~~~~~~~~~~~~
                              %@
```

memcpy Size Checking

```
int copyRect(NSRect *to, NSRect *from) {
  memcpy(to, from, sizeof(to));
}
```

memcpy Size Checking

```
int copyRect(NSRect *to, NSRect *from) {
  memcpy(to, from, sizeof(to));
}
```


```
warning: 'memcpy' call operates on objects of type 'NSRect' while the size
is based on a different type 'NSRect *'[-Wsizeof-pointer-memaccess]
  memcpy(to, from, sizeof(to));
         ~~              ^~
note: did you mean to dereference the argument to 'sizeof' (and multiply
it by the number of elements)?
  memcpy(to, from, sizeof(to));
                          ^~
```

memcpy Size Checking

```
int copyRect(NSRect *to, NSRect *from) {
  memcpy(to, from, sizeof(*to));
}
```

memset Issues with C++ Objects

```
void clear_Y(Y* y) {
  memset(y, 0, sizeof(*y));
}
```

memset Issues with C++ Objects

```
void clear_Y(Y* y) {
  memset(y, 0, sizeof(*y));
}
```


```
warning: destination for this 'memset' call is a pointer to dynamic class 'Y';
vtable pointer will be overwritten [-Wdynamic-class-memaccess]

  memset(y, 0, sizeof(*y));
  ~~~~~~ ^
```

CF Containers and Non-Pointer Sized Values

```
int x[] = { 1, 2, 3 };
// Be super clever and pretend ‘x’ is an array of pointers
// so I can stuff it inside a CF container.
CFSetRef set = CFSetCreate(NULL, (const void **)x, 3, &kCFTypeSetCallBacks);
```

CF Containers and Non-Pointer Sized Values

```
int x[] = { 1, 2, 3 };
// Be super clever and pretend ‘x’ is an array of pointers
// so I can stuff it inside a CF container.
CFSetRef set = CFSetCreate(NULL, (const void **)x, 3, &kCFTypeSetCallBacks);
```


```
int x[] = { 1, 2, 3 };
// Be super clever and pretend ‘x’ is an array of pointers
// so I can stuff it inside a CF container.
CFSetRef set = CFSetCreate(NULL, (const void **)x, 3, &kCFTypeSetCallBacks);
```

The first argument to 'CFSetCreate' must be a C array of pointer-sized values, not 'int [3]'

`malloc` and `free`

- ARC automates Objective-C memory management
  - Manual memory management is still your responsibility
- `malloc` and `free` checking
  - Find potential leaks
  - Find potential use-after-releases
  - Will not find all issues

malloc and free

```
static MyStack *allocMyStack(void)
{
    MyStack *s = (MyStack *)malloc(sizeof(MyStack));

    if (!s) {
      return 0;
    }

    s->data = setupData();

    if (!s->data) {
        return 0;
    }

    return s;
}
```

malloc and free

3. Memory is never released; potential leak of memory pointed to by 's' Done

```
static MyStack *allocMyStack(void)
{
    MyStack *s =(MyStack *)malloc(sizeof(MyStack));        1. Memory is allocated

    if (!s) {                                              2. Assuming 's' is non-null
      return 0;
    }

    s->data = setupData();

    if (!s->data) {
      return 0;                                            3. Memory is never released; potential leak of memory pointed to by 's'
    }

    return s;
}
```

15. Memory is never released; potential leak of memory point... Done

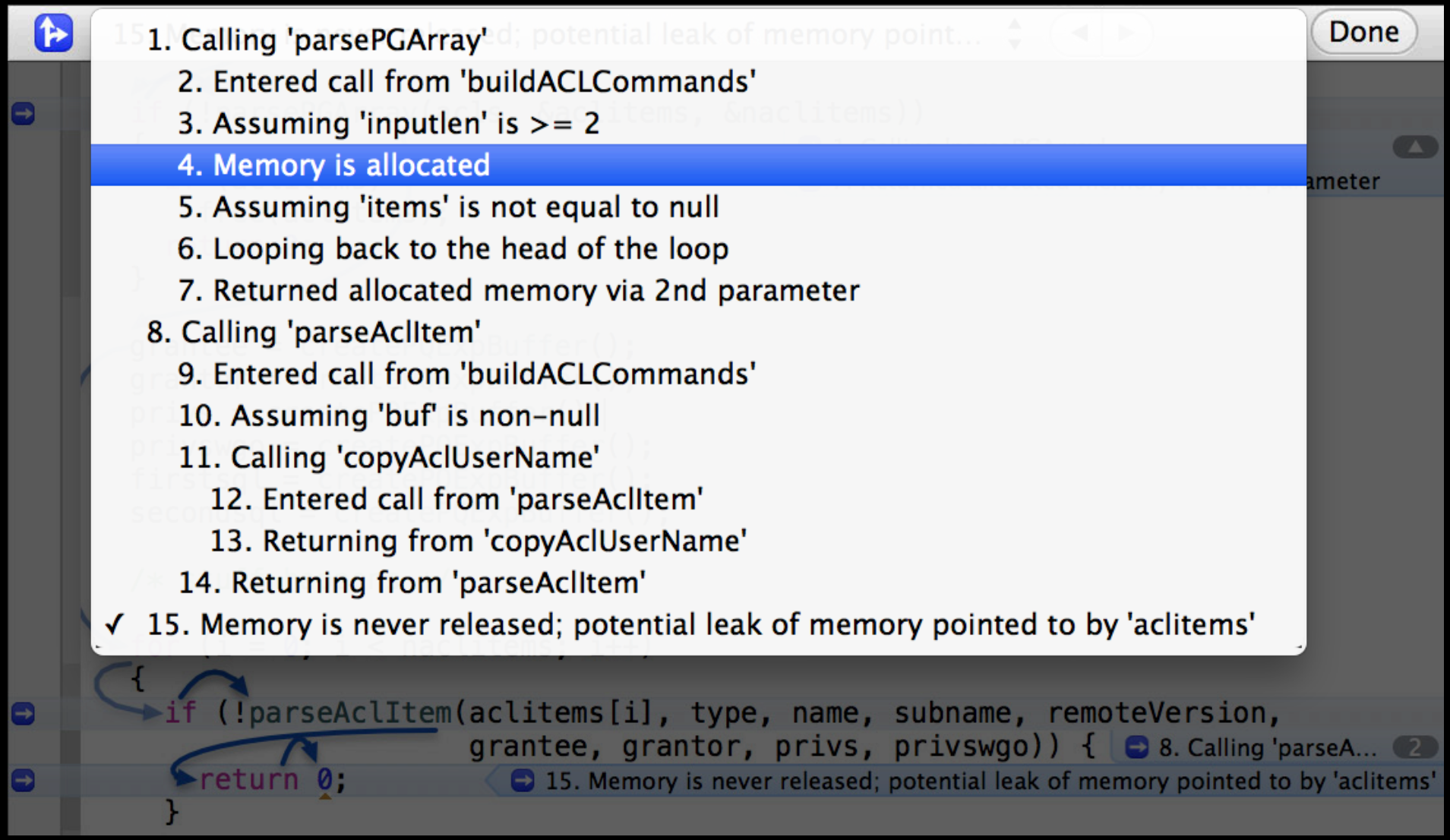
```
if (!parsePGArray(acls, &aclitems, &naclitems))
{
  if (aclitems)
    free(aclitems);
  return 0;
}

grantee = createPQExpBuffer();
grantor = createPQExpBuffer();
privs = createPQExpBuffer();
privswgo = createPQExpBuffer();
firstsql = createPQExpBuffer();
secondsql = createPQExpBuffer();

/* stuff happens */

for (i = 0; i < naclitems; i++)
{
  if (!parseAclItem(aclitems[i], type, name, subname, remoteVersion,
                    grantee, grantor, privs, privswgo)) {
    return 0;
  }
```

1. Calling 'parsePGArray'
7. Returned allocated memory via 2nd parameter
8. Calling 'parseA... 2
15. Memory is never released; potential leak of memory pointed to by 'aclitems'

15. Memory is never released; potential leak of memory point... Done

```
if (!parsePGArray(acls, &aclitems, &naclitems))
{
  if (aclitems)
    free(aclitems);
  return 0;
}

grantee = createPQExpBuffer();
grantor = createPQExpBuffer();
privs = createPQExpBuffer();
privswgo = createPQExpBuffer();
firstsql = createPQExpBuffer();
secondsql = createPQExpBuffer();

/* stuff happens */

for (i = 0; i < naclitems; i++)
{
  if (!parseAclItem(aclitems[i], type, name, subname, remoteVersion,
                    grantee, grantor, privs, privswgo)) {
    return 0;
  }
```

1. Calling 'parsePGArray'
7. Returned allocated memory via 2nd parameter
8. Calling 'parseA... 2
15. Memory is never released; potential leak of memory pointed to by 'aclitems'


Done
1. Calling 'parsePGArray'
2. Entered call from 'buildACLCommands'
3. Assuming 'inputlen' is >= 2
4. Memory is allocated
5. Assuming 'items' is not equal to null
6. Looping back to the head of the loop
7. Returned allocated memory via 2nd parameter
8. Calling 'parseAclItem'
9. Entered call from 'buildACLCommands'
10. Assuming 'buf' is non-null
11. Calling 'copyAclUserName'
12. Entered call from 'parseAclItem'
13. Returning from 'copyAclUserName'
14. Returning from 'parseAclItem'
✓ 15. Memory is never released; potential leak of memory pointed to by 'aclitems'
if (!parseAclItem(aclitems[i], type, name, subname, remoteVersion,
grantee, grantor, privs, privswgo)) {
return 0;
}
8. Calling 'parseA...
15. Memory is never released; potential leak of memory pointed to by 'aclitems'

```
bool parsePGArray(const char *atext,
                  char ***itemarray, int *nitems)
{
  int inputlen;
  char **items;
  char *strings;
  int curitem;
  *itemarray = 0;
  *nitems = 0;
  inputlen = strlen(atext);

  if (inputlen < 2 || atext[0] != '{' || atext[inputlen - 1] != '}') {
    return 0;
  }

  items = (char **) malloc(inputlen * (sizeof(char *) + sizeof(char)));

  if (items == 0)
    return 0;

  *itemarray = items;
```

2. Entered call from 'buildACLCommands'

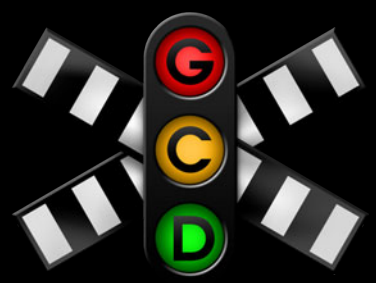
3. Assuming 'inputlen' is >= 2

4. Memory is allocated

5. Assuming 'items' is not equal to null

Many Other Checks

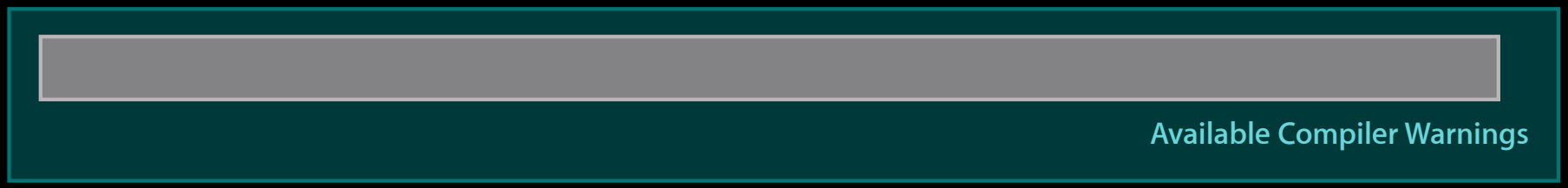
Many Other Checks



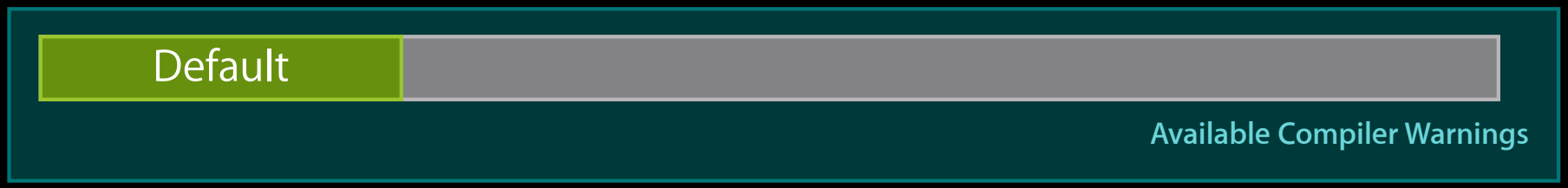
Controlling Warnings

Additive Approach to Warnings

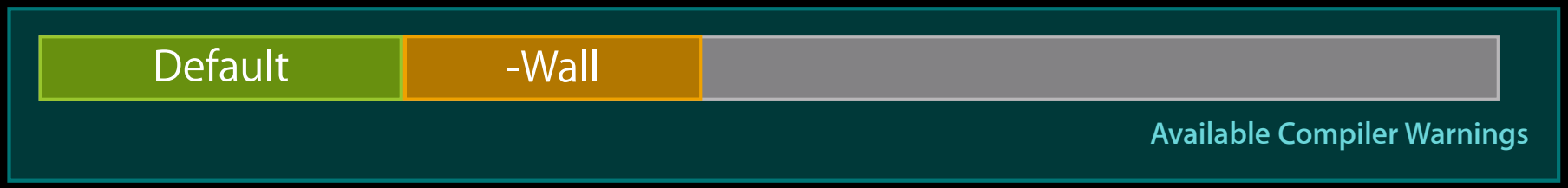
Additive Approach to Warnings



Additive Approach to Warnings

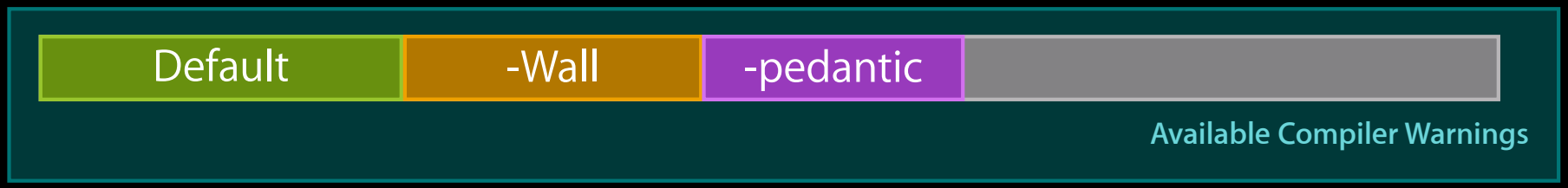


Additive Approach to Warnings



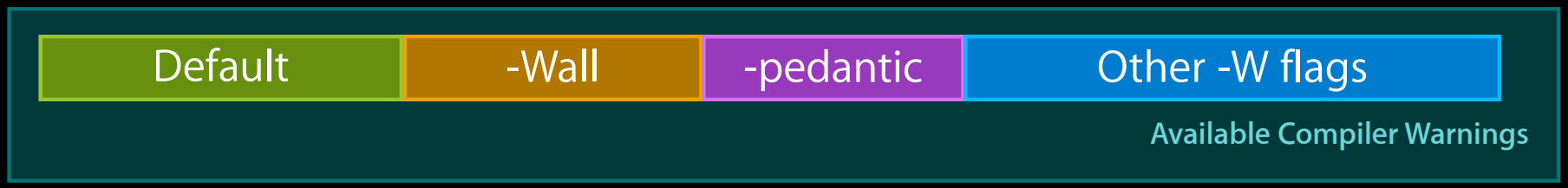
- `-Wall` is not "all warnings" because of historical expectations
  - Frequently paired with `-Werror`
  - Warnings added to `-Wall` are done with care

Additive Approach to Warnings



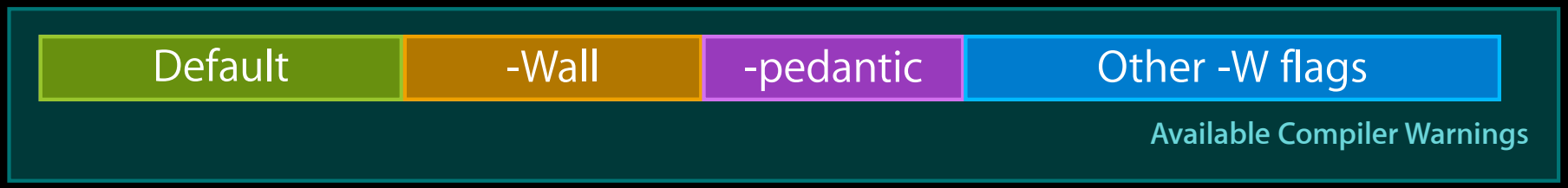
- `-Wall` is not "all warnings" because of historical expectations
  - Frequently paired with `-Werror`
  - Warnings added to `-Wall` are done with care

Additive Approach to Warnings



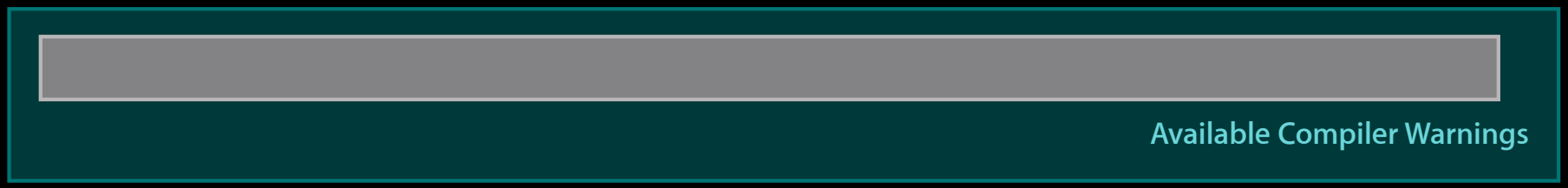
- `-Wall` is not "all warnings" because of historical expectations
  - Frequently paired with `-Werror`
  - Warnings added to `-Wall` are done with care

Additive Approach to Warnings



- `-Wall` is not "all warnings" because of historical expectations
  - Frequently paired with `-Werror`
  - Warnings added to `-Wall` are done with care
- Want an additional warning? Need to know the `-W` flag

Subtractive Approach to Warnings



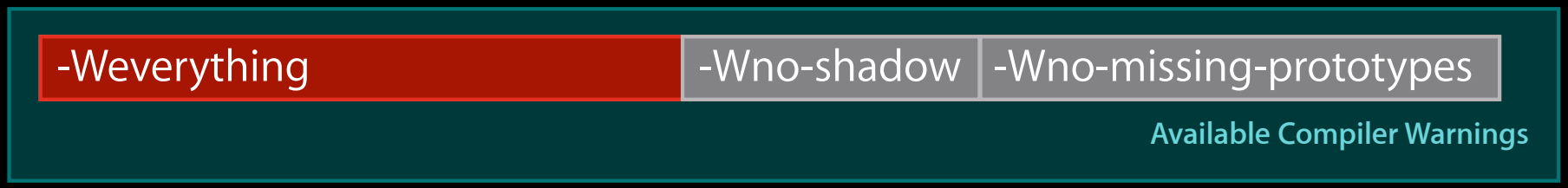
Subtractive Approach to Warnings

-Weverything

Available Compiler Warnings

- `-Weverything` is truly **all** warnings
  - Will change over time
  - Expect build failures with `-Werror`

Subtractive Approach to Warnings



- `-Weverything` is truly **all** warnings
  - Will change over time
  - Expect build failures with `-Werror`
- Disable warnings you don't want with `-Wno-XXX`
  - Compiler tells you the flag when you get a warning

Fine-Grain Control of Compiler Warnings

- Control compiler warnings within a single file using pragmas:

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"

char b = 'df'; // no warning.

#pragma clang diagnostic pop
```

- Can map warnings to `ignored` or `error`

Controlling Analyzer Issues

- No matching mechanism for the static analyzer
- Checkers can be enabled/disabled in Xcode's build settings

Static Analyzer – Checkers	
Dead stores	Yes
Misuse of GCD	Yes
Misuse of malloc/free	Yes
Static Analyzer – Checkers – Objective-C	
'@synchronized' with 'nil' mutex	Yes
Improper handling of CFError and NSError	Yes
Method signatures mismatch	Yes
Misuse of CFNumberCreate	Yes
Misuse of collections API	Yes
Unused ivars	Yes
Violation of 'self = [super init]' rule	Yes
Violation of reference counting rules	Yes
Static Analyzer – Checkers – Security	
Floating point value used as loop counter	No
Misuse of Keychain Services API	Yes
Unchecked return values	Yes
Use of 'getpw', 'gets' (buffer overflow)	Yes
Use of 'mktemp' or predictable 'mktemps'	Yes
Use of 'rand' functions	No
Use of 'strcpy' and 'strcat'	No
Use of 'vfork'	Yes

Better Compiler → Better Apps

Better Compiler → Better Apps

Summary

Apple LLVM Compiler 4.0

Summary

- Faster performance
  - ARC optimizer
  - AVX vector extensions (OS X)
  - Integrated ARM assembler
- Language improvements
  - Objective-C enhancements
  - C++11 support
- Find problems early
  - Intelligent compiler warnings
  - Vastly improved static code analyzer

Apple LLVM Compiler 4.0

More Information

Michael Jurewitz
Developer Tools Evangelist
jury@apple.com

LLVM Project
Open-Source LLVM Project Home
http://llvm.org

Clang Static Analyzer
Open-Source Clang Static Analyzer
http://clang-analyzer.llvm.org

Apple Developer Forums
http://devforums.apple.com

Labs

Objective-C and Automatic Reference Counting Lab	Developer Tools Lab C Thursday 2:00PM
LLVM Lab	Developer Tools Lab A Thursday 2:00PM

WWDC2012