FUNCTIONS WITH DATA - CLOSURES IN C

AN N3780 OVERVIEW

N3781 - BJÖRKUS "THEPHD" DORKUS

https://thephd.dev | https://soasis.org

HTML Link: https://thephd.dev/_presentations/standards/C/2026%20February%20Virtual/n3781/Functions%20with%20Data%20-%20Closures%20in%20C.html

CLOSURES HAVE A RICH HISTORY IN C

FOUR DISTINCT EXISTING PRACTICES

- Plain C
- GNU Nested Functions (GCC-specific, Clang refuses to implement)
- Apple Blocks (Clang-specific, removed from GCC after NeXT Support died off slowly)
- Lambdas

FOUR DISTINCT EXISTING PRACTICES

- Plain C
  - `static` data transport subtype
  - `thread_local` data transport subtype
  - `T*` (specific)/`void*` (general) user data function parameter modification subtype

FOUR DISTINCT EXISTING PRACTICES

- GNU Nested Functions (GCC-specific, Clang refuses to implement)
  - Declaration-based
  - Trampolines (stack or heap) control behavior
  - `__builtin_with_static_chain` compatible (but not usually used this way)
  - By-reference/by-name stack data subtype
  - Forward-declarable with `auto`

FOUR DISTINCT EXISTING PRACTICES

- Apple Blocks (Clang-specific, removed from GCC after NeXT Support died off slowly)
  - Expression-based, can be used immediately in-line with other function arguments
  - blocks-rt / Blocks Runtime, Automatic Reference Counted (ARC)-based lifetime control
  - Manual trampoline creation subtype
  - By-reference/by-name stack data AND by-value copy stack data subtype

HONORABLE MENTION

Borland C's closure function pointer annotation

OTHER RELEVANT EXISTING PRACTICE

- C++ Lambdas
- C++ `struct` + function call operator

SHORTHAND AND LONGFORM DEFINITIONS / EXPRESSIONS?

C (GNU) and C++ solutions in this space are only ones to pick a "long form" solution or "short hand" solution but not have both.

- Lua, JavaScript, Haskell/OCaml/most functional languages: syntax for shorthand is same for long form: just sprinkle in an identifier, have both.
- C++, C (Apple Blocks), Rust, Python, Java, C#, etc.: syntax for shorthand lambdas is different from syntax for long form function definitions, have both.
- C (GNU Nested Functions): long form definition only.

PRIAMRY REASON FOR HAVING BOTH?

SMALL-FUNCTION-INSIDE-OTHER-FUNCTION-CALL IS AN IMPORTANT USECAE

(e.g. a call to `sort(...)`)

(Secondary reason: a second kind of "statement expression")

CURRENT PROPOSALS

ISO C TRACK RECORD

- Apple Blocks - Blaine Garst & Apple - n1451, n1457, final n2030
- Nested Functions & Closure Contexts - Martin Uecker - n2661, n3654
- Lambdas - Jens Gustedt - n2982, n2893
  - n2924: Type-generic (`auto` parameters)
- Function Literals and Local Functions - Thiago Adams - n3678, n3679
- Capture Functions / Gustedt-style Lambdas - JeanHeyd Meneide & Shepherd - n3780

DESIGN COMPONENTS

- **Capture By-Name**: can refer to an object directly, as if working with it as an l-value or dereferencing a pointer to that object. Sometimes also called "capture by reference".
- **Capture By-Value**: can refer to a copy of an object from within the function, at some fixed point in time. The lifetime of the copied object from the capture is tied to the closure rather than the scope it comes from.
- **Selective Capture**: can pick and choose what to capture and how it gets captured.
- **Safe to Return Closure**: part of the dynamic lifetime problem, but is there a way to write this closure type such that it is safe to return?

DESIGN COMPONENTS

- **Relocatable to Heap (Lifetime Management)**: if it is possible to extend or otherwise change the lifetime of a closure so that it can last longer, either by copying or relocating the closure.
- **Usable Directly as Expression**: whether or not the closure can appear as a function argument or something else.
- **Forward-Declarable**: whether or not the closure can be forward-declared and perhaps used in some ways (e.g., callable as a function but perhaps any related object definition might not be usable).
- **Immediately Invokable**: whether or not the closure can be immediately invoked, usually without naming it.

DESIGN COMPONENTS

- **Convertible to Function Pointer**: whether or not the closure can be converted to a function pointer of an identical signature to be called.
- **Convertible to "Wide" Function Type**: whether or not the closure can be converted to a "wide" function pointer type, now or into the future.
- **Access to Non-Erased Object/Type**: can use and store the closure object without erasing the type or the object first.
- **Access to Captures through Object/Type**: access the closure’s captures outside of the closure’s invocable body.
- **Recursion Possible**: can refer to itself in order to create a recursive algorithm in the normal fashion, without needing a special feature like C++'s Deducing This or the proposed `__self_func`.

Feature	GNU Nested Functions	Apple Blocks	C++-Style Lambdas in C	Function Literals	Local Functions
Capture By-Name	✅ (default, use-based)	✅ (`__block ident;`)	✅ (`[&]`, `[&ident]`)	❌	❌
Capture By-Value	❌	✅ (default, use-based)	✅ (`[=]`, `[ident]`)	❌	❌
Selective Capture	❌ (use-based, by-name only)	✅ (for by-name) ❌ (for by-value, use-based)	✅	❌	❌
Safe to Return Closure	❌	⚠️ (requires `Block_copy`)	✅	✅ (never unsafe)	✅ (never unsafe)
Relocatable to Heap (Lifetime Management)	❌	✅ (`Block_copy`/`Block_release`)	✅ (`malloc`/`memcpy`/`free`)	✅ (not needed)	✅ (not needed)

Feature	GNU Nested Functions	Apple Blocks	C++-Style Lambdas in C	Function Literals	Local Functions
Usable Directly as Expression	❌	✅	✅	✅	❌
Forward-Declarable	✅	❌	❌	❌	✅
Immediately Invokable	❌	✅	✅	✅	❌
Convertible to Function Pointer	✅	❌	⚠️ (only capture-less)	✅	✅
Convertible to "Wide" Function Type	✅	✅	✅	✅	✅

Feature	GNU Nested Functions	Apple Blocks	C++-Style Lambdas in C	Function Literals	Local Functions
Access to Non-Erased Object/Type	❌ ((wide) function pointer only)	❌ (Block type/wide function pointer only)	✅ (unique type/ size)	❌ (no object to access)	❌
Access to Captures through Object/Type	❌	❌	❌	❌	❌
Recursion Possible	✅ (use the identifier of the nested function)	❌ (`__self_func` required)	❌ (`__self_func` required)	❌ (`__self_func` required)	✅ (use the identifier of the local function)

IRRECONCILABLE DIFFERENCES - NESTED FUNCTIONS

Why not Nested Functions (Uecker, GCC)?

- Lack of control (always by-name, not by-value?)
- Lifetime issues -- cannot elevate data to higher lifetime
- Performance issues with existing design (partly solvable?)
- Locked-in ABI/Source Code (it must down-convert to a function pointer trampoline)

```
typedef void(fn_t)(void);

fn_t* use_later;

void f (fn_t* f) {
    use_later = f;
}

void g (void) {
    use_later();
}
```

IRRECONCILABLE DIFFERENCES - NESTED FUNCTIONS

```
extern int computed_answer;

int main (int argc, char* argv[]) {
    { // New scope
        int val = argc + 1;
        void compute (void) {
            computed_answer = val;
        }
        f(compute);
        // end of scope, end of "compute" and "val" lifetime
    }
    g(); // uh oh
    return computed_answer;
}
```

IRRECONCILABLE DIFFERENCES - APPLE BLOCKS

Why not Apple Blocks (Garst, Apple)?

- Type-erased and uses an Reference Counting implementation to boost lifetime
- Requires a "runtime" -- mandatory overhead as a basic implementation
- Pre-existing ABI -- locked-in details can prevent growth or change

IRRECONCILABLE DIFFERENCES - FUNCTION LITERALS / LOCAL FUNCTIONS

Why not Function Literals/Local Functions (Adams, Cake)?

- Does not engage with 90% of code that uses Nested Functions / Apple Blocks (accesses a variable from the local context)
- Simplicit of proposal comes from making it the user's problem, not providing elegant solution
- Not useful

CAPTURE FUCNTIONS

OR "WHAT IS BEING PROPOSED INSTEAD?"

BASED ON GCC WITH SLIGHT MODIFICATION

- Uses familiar function declaration/definition syntax
- Adds `_Capture(...)` clause to explicitly grab memory

`struct`/`union` THAT IS CALLABLE ("INVOCABLE")

- Implementation-defined structure (or union) type that holds all data
  - `sizeof`, `alignof` capable
  - Regular object (that can be `memcpy`'d or otherwise manipulated like any other complete object)
  - Can be passed in and returned in normal way users expect

```
void take_something(void* p);

int main(int argc, [[maybe_unused]] char* argv[]) {
    int f () _Capture(argc) {
        return argc + 1;
    }
    return f();
}
```

`struct`/`union` THAT IS CALLABLE ("INVOCABLE")

- Address-of operation produces a `void*` (it's an object), not a function pointer

```
void take_something(void* p);

int main(int argc, [[maybe_unused]] const char* argv[]) {
    int f () _Capture(argc) {
        return argc + 1;
    }
    take_something(&f);
    take_something(f); // constraint violation -- cannot convert {object} to `void*`
    return f();
}
```

`struct`/`union` THAT IS CALLABLE ("INVOCABLE")

- Empty capture lists result in a capture-less closure
  - Implicit conversion to typed function pointer
  - Useful idiom: object + hand-crafted, in-line trampoline to work with old code

```
int perform_work(int (work_fn)(void*), void* userdata);

int main(int argc, [[maybe_unused]] const char* argv[]) {
    int f () _Capture(argc) {
        return argc + 1;
    }
    int f_jumpoff (void* userdata) _Capture() {
        typeof(f)* pf = userdata; // pointer to closure type
        return (*pf)();
    }
    return perform_work(f_jumpoff, &f);
}
```

WEAKNESS: DOES NOT WORK WITH `qsort`!

- No `void*` userdata with function pointer? No way to pass as pure function pointer
  - Same problem for the Wide Function Pointers, Apple Blocks, etc. etc.
- Needs a "trampoline", implicit trampolines (conversion to plain function pointer)
  - Talked about in the Appendix §6.3
  - GCC uses their own stack or heap based trampolines -- comes at a price
  - Apple Blocks has an explicit function to make a trampoline (and unmake it)

LAMBDAS ARE LARGELY THE SAME

- Lambdas are largely the same as Capture Functions, but they work as expressions
- Usable for `qsort` purposes (with no captures)

```
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int list[] = { 2, 11, 32, 49, 57, 20, 110, 203 };
    qsort(list, _Countof(list), sizeof(*list),
        // lambda syntax
        [](const void* untyped_left, const void* untyped_right) {
            const int* left = untyped_left;
            const int* right = untyped_right;
            return *left - *right;
        }
    );

    return list[0]; // return 2;
}
```

NEW MEMBER ACCESS FOR CAPTURES

```
#include <stdio.h>

int main () {
    int x = 30;
    double y = 5.0;
    char z = 'a';
```

```
    int cap_fn0 () _Capture(=, &renamed_x = x, &z) {
        printf("inside cap_fn0 | renamed_x: %d, y: %f, z: %c\n",
            renamed_x, y, z);
    }

    int cap_fn1 () _Capture(&, renamed_y = y, z) {
        printf("inside cap_fn1 | x: %d, renamed_y: %f, z: %c\n",
            x, renamed_y, z);
    }
    // ...
```

NEW MEMBER ACCESS FOR CAPTURES

```
    // ...
    x = 60;
    y = 10.0;
    z = 'z';

    cap_fn0();
    cap_fn1();
```

```
    printf("\n");

    printf("inside main fn | cap_fn0.renamed_x: %d, cap_fn0.y: %f, cap_fn0.z: %c\n",
        cap_fn0.renamed_x, cap_fn0.y, cap_fn0.z);
    printf("inside main fn | cap_fn1.x: %d, cap_fn1.renamed_y: %f, cap_fn1.z: %c\n",
        cap_fn1.x, cap_fn1.renamed_y, cap_fn1.z);

    return 0;
}
```

NEW MEMBER ACCESS FOR CAPTURES

Prints:

```
inside cap_fn0 | renamed_x: 60, y: 5.0, z: a
inside cap_fn1 | x: 60, renamed_y: 10.0, z: z

inside main fn | cap_fn0.renamed_x: 60, cap_fn0.y: 5.0, cap_fn0.z: a
```

```
inside main fn | cap_fn1.x: 60, cap_fn1.renamed_y: 10.0, cap_fn1.z: z
```

PROPOSING BOTH

- Lambdas: C++ Compatibility, Expression-Based, "Shorthand"
- Capture Functions: GNU Nested Function Familiarity, Declaration-Based, "Longform"

FEATURESET

Feature	C Lambdas	Capture Functions
Capture By-Name	✅ (`[&]`, `[&ident]`)	✅ (`_Capture(&)`, `_Capture(&ident)`)
Capture By-Value	✅ (`[=]`, `[ident]`)	✅ (`_Capture(=)`, `_Capture(ident)`)
Selective Capture	✅	✅
Safe to Return Closure	✅	✅
Relocatable to Heap (Lifetime Management)	✅ (`malloc`/`memcpy`/`free`)	✅ (`malloc`/`memcpy`/`free`)
Usable Directly as Expression	✅	❌
Forward-Declarable	❌	✅
Immediately Invokable	✅	✅
Convertible to Function Pointer	⚠️ (only capture-less)	⚠️ (only capture-less)
Convertible to "Wide" Function Type	✅	✅
Access to Non-Erased Object/Type	✅ (unique type/size)	✅ (unique type/size)
Access to Captures through Object/Type	✅	✅
Recursion Possible	❌ (`__self_func` required)	✅

IMPORTANT FOR TRANSITIONING

- Approximate GNU Nested Function Behavior by using all-capture &
- Approximate Apple Blocks Behavior by using all-value-capture =

EXAMPLE FROM N3654, MEANT AS LAMBDA QUIZ

"What is the output of this?" - Meant to be a Gotcha Quiz.

```
#include <stdio.h>

int j = 3;

int main()
{
  int i = 3;
  auto foo = [=](){ printf("%d\n", i); };
  auto bar = [=](){ printf("%d\n", j); };
  i = j = 4;
  foo();
  bar();
}
```

EXAMPLE FROM N3654, MEANT AS LAMBDA QUIZ

The answer is "3" and then "4" (https://godbolt.org/z/KW4j1zG93)

```
#include <stdio.h>

int j = 3;

int main()
{
  int i = 3;
  auto foo = [=](){ printf("%d\n", i); };
  auto bar = [=](){ printf("%d\n", j); };
  i = j = 4;
  foo();
  bar();
}
```

IS THIS WRONG?

N3564 implies it is wrong! Except, with Apple Blocks...

```
#include <stdio.h>

int j = 3;

int main()
{
    int i = 3;
    auto foo = ^(){ printf("%d\n", i); };
    auto bar = ^(){ printf("%d\n", j); };
    i = j = 4;
    foo();
    bar();
}
```

IS THIS WRONG?

The answer is "3" and then "4" (https://godbolt.org/z/a9c79cjYb)

```
#include <stdio.h>

int j = 3;

int main()
{
    int i = 3;
    auto foo = ^(){ printf("%d\n", i); };
    auto bar = ^(){ printf("%d\n", j); };
    i = j = 4;
    foo();
    bar();
}
```

GNU NESTED FUNCTIONS PRODUCES DIFFERENT ANSWER

The answer here is "4" and then "4" (https://godbolt.org/z/voWG3Gjo3)

```c
#include <stdio.h>

int j = 3;

int main()
{
    int i = 3;
    void foo() { printf("%d\n", i); };
    void bar() { printf("%d\n", j); };
    i = j = 4;
    foo();
    bar();
}
```

BUT... LAMBDAS CAN GIVE THE EXACT SAME ANSWER!

The answer here is "4" and then "4" (https://godbolt.org/z/EW8PETdxz)

```
#include <stdio.h>

int j = 3;

int main()
{
    int i = 3;
    auto foo = [&](){ printf("%d\n", i); };
    auto bar = [&](){ printf("%d\n", j); };
    i = j = 4;
    foo();
    bar();
}
```

AND THAT'S OKAY!!

A unifying solution allows for both paradigms to be used;

- Use the `_Capture(&)` to get GNU Nested Function behavior, works for that ecosystem
- Use `_Capture(=)` to get Apple Blocks behavior, works for that ecosystem
  - With specific `&ident` to emulate `__block` vars!

THEREFORE, WE PROPOSE EXPLICIT CAPTURES TO ALLOW FOR THIS!

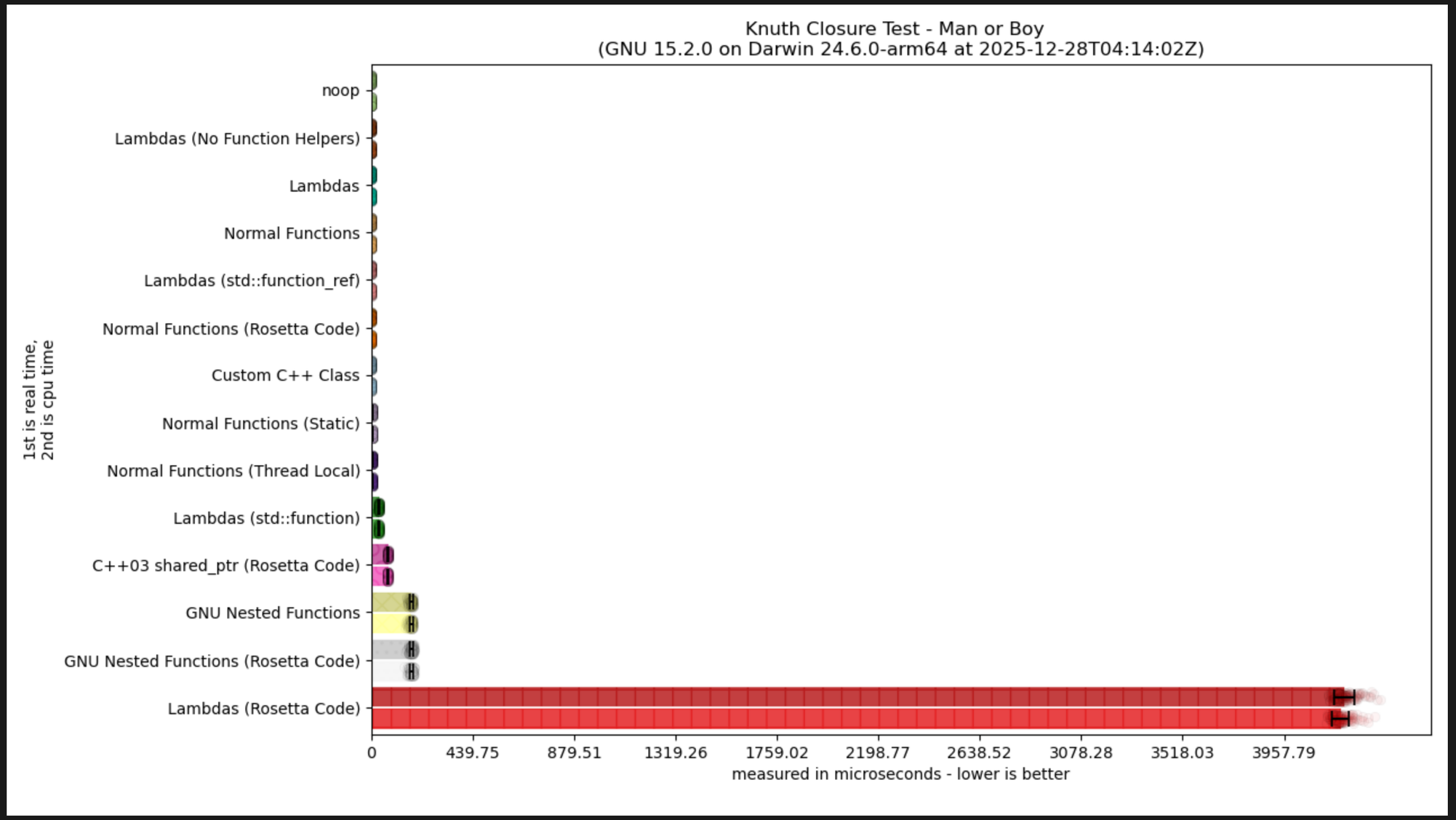
NOBODY has to sacrifice what they want!

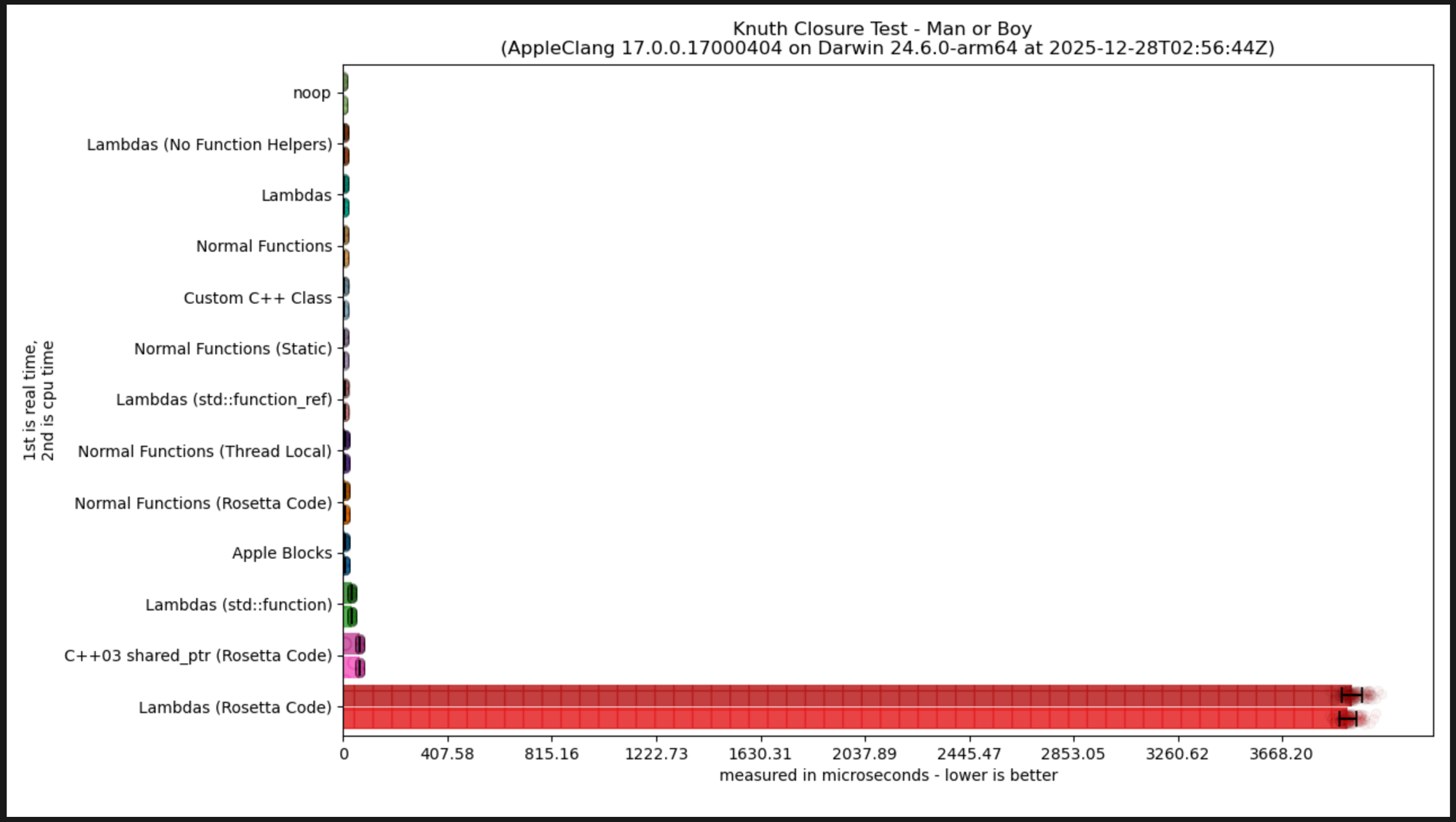
FIRST APPROXIMATION - PERFORMANCE

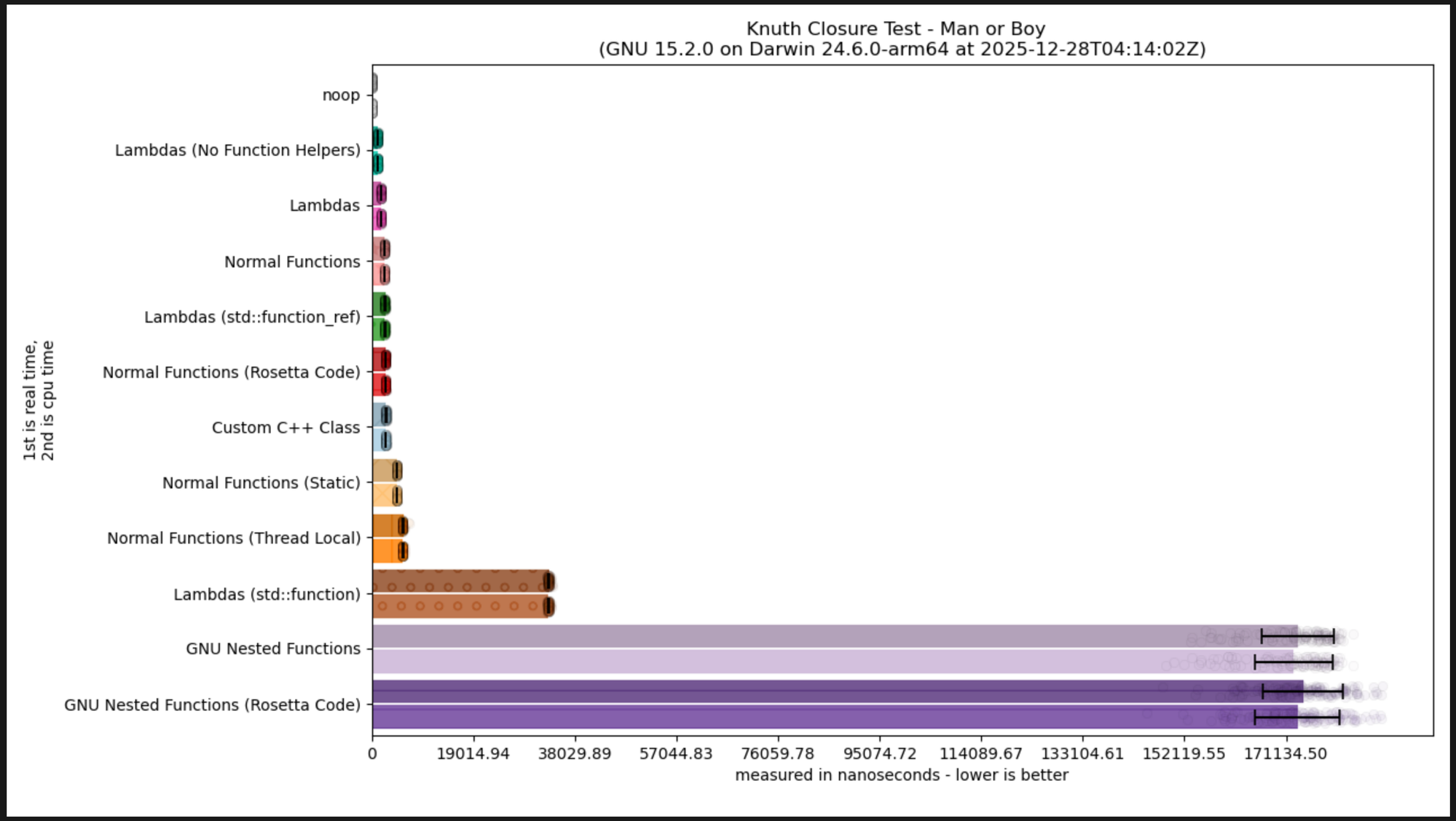
NOT ALWAYS THE BEST !

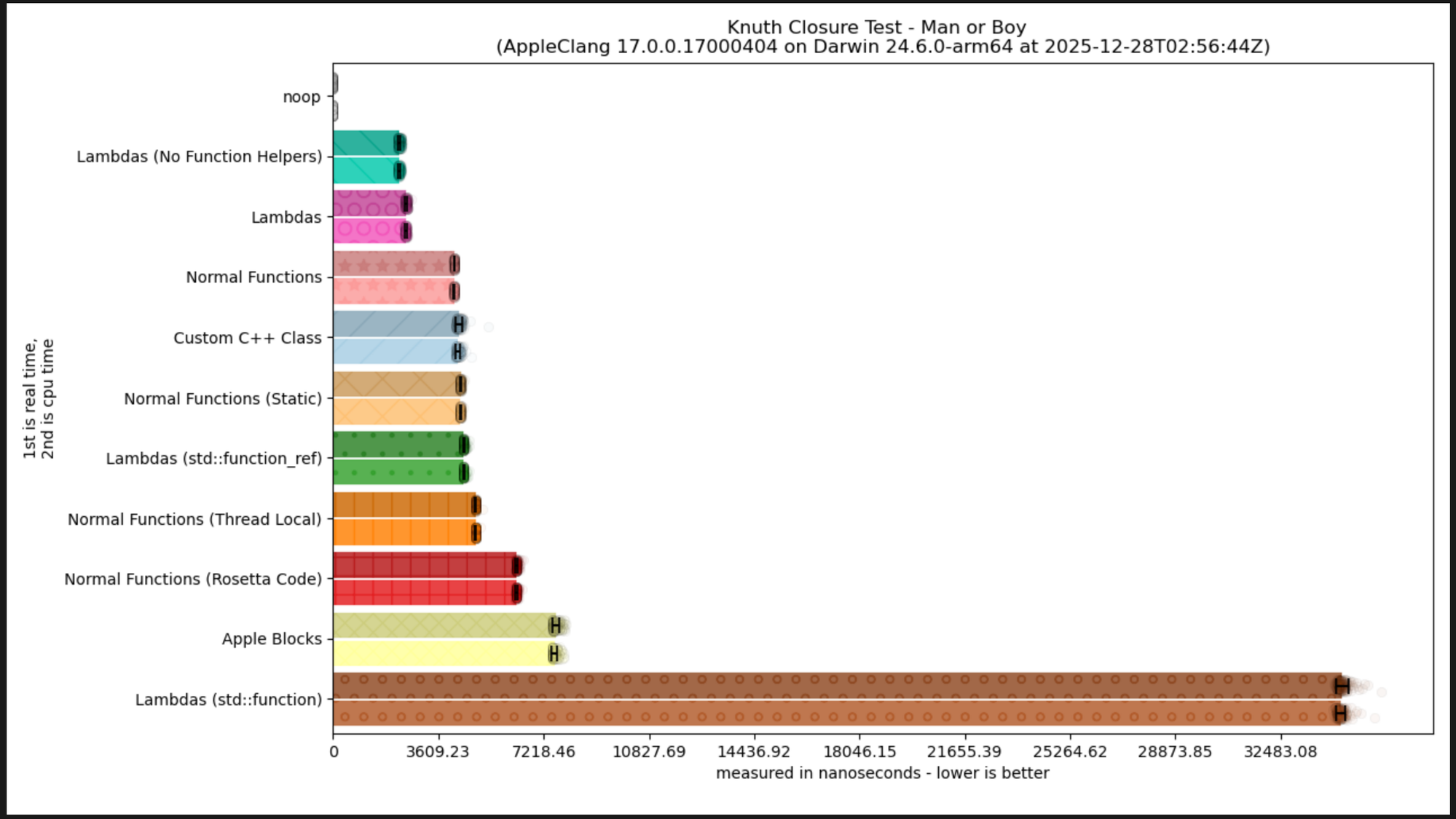
Captures current implementation strength / weaknesses

but not full potential !


Knuth Closure Test - Man or Boy
(GNU 15.2.0 on Darwin 24.6.0-arm64 at 2025-12-28T04:14:02Z)
noop
Lambdas (No Function Helpers)
Lambdas
Normal Functions
Lambdas (std::function_ref)
Normal Functions (Rosetta Code)
Custom C++ Class
Normal Functions (Static)
Normal Functions (Thread Local)
Lambdas (std::function)
C++03 shared_ptr (Rosetta Code)
GNU Nested Functions
GNU Nested Functions (Rosetta Code)
Lambdas (Rosetta Code)
1st is real time,
2nd is cpu time
0
439.75
879.51
1319.26
1759.02
2198.77
2638.52
3078.28
3518.03
3957.79
measured in microseconds - lower is better


Knuth Closure Test - Man or Boy
(AppleClang 17.0.0.17000404 on Darwin 24.6.0-arm64 at 2025-12-28T02:56:44Z)
noop
Lambdas (No Function Helpers)
Lambdas
Normal Functions
Custom C++ Class
Normal Functions (Static)
Lambdas (std::function_ref)
Normal Functions (Thread Local)
Normal Functions (Rosetta Code)
Apple Blocks
Lambdas (std::function)
C++03 shared_ptr (Rosetta Code)
Lambdas (Rosetta Code)
1st is real time,
2nd is cpu time
0
407.58
815.16
1222.73
1630.31
2037.89
2445.47
2853.05
3260.62
3668.20
measured in microseconds - lower is better


Knuth Closure Test - Man or Boy
(GNU 15.2.0 on Darwin 24.6.0-arm64 at 2025-12-28T04:14:02Z)
noop
Lambdas (No Function Helpers)
Lambdas
Normal Functions
Lambdas (std::function_ref)
Normal Functions (Rosetta Code)
Custom C++ Class
Normal Functions (Static)
Normal Functions (Thread Local)
Lambdas (std::function)
GNU Nested Functions
GNU Nested Functions (Rosetta Code)
1st is real time,
2nd is cpu time
0
19014.94
38029.89
57044.83
76059.78
95074.72
114089.67
133104.61
152119.55
171134.50
measured in nanoseconds - lower is better


Knuth Closure Test - Man or Boy
(AppleClang 17.0.0.17000404 on Darwin 24.6.0-arm64 at 2025-12-28T02:56:44Z)
noop
Lambdas (No Function Helpers)
Lambdas
Normal Functions
Custom C++ Class
Normal Functions (Static)
Lambdas (std::function_ref)
Normal Functions (Thread Local)
Normal Functions (Rosetta Code)
Apple Blocks
Lambdas (std::function)
1st is real time,
2nd is cpu time
0
3609.23
7218.46
10827.69
14436.92
18046.15
21655.39
25264.62
28873.85
32483.08
measured in nanoseconds - lower is better

FULL ANALYSIS AVAILABLE AT:

- The Cost of a Closure in C - https://thephd.dev/the-cost-of-a-closure-in-c-c2y
- The Cost of a Closure in C, The Rest - https://thephd.dev/the-cost-of-a-closure-in-c-c2y-followup

TAKEAWAYS I:

- GNU Nested Functions need more work to be viable
  - Need Wide Function Pointer to save some performance, too
- Lambda-style design (but not necessarily Lambdas) produce the best throughput/speed and smallest code - Matches Capture Functions, Gustedt-style and C++-style Lambdas designs
- Allocation-heavy, type-erased designs contract some implicit, unavoidable overhead in the "best" case
  - Apple Blocks (Blocks is a runtime, allocates and deallocates the control block even for the simple case)
- Trampolines are, generally, very bad for performance
  - `-ftrampoline-impl=heap` from GCC (not yet tested, but still clears `icache`)

TAKEAWAYS II:

ALL SOLUTIONS NEED A WIDE FUNCTION POINTER TYPE

- Lambda with `std::function_ref` approximates "Wide Function Pointer" performance
- Will be default of C ecosystem, which is pretty good performance ("second place" on the graph)

ANYTHING ELSE?

- Original proposal has Forward Declarations -- not trivial, probably will remove.
- C++ Lambdas have `mutable` keyword to allow messing with internal value
- In-general, captures do not allow for storage class specifiers. Maybe we allow that?

COMMITTEE QUESTIONS

1. "We like the direction of N3780 and want to see -- at a much later point -- Capture Functions with Wording."
2. "We like the direction of N3780 and want to see -- at a much later point -- Lambdas with Wording."
3. "We encourage further work for C2y/C3a into a Technical Specification." (NOT into the IS.)