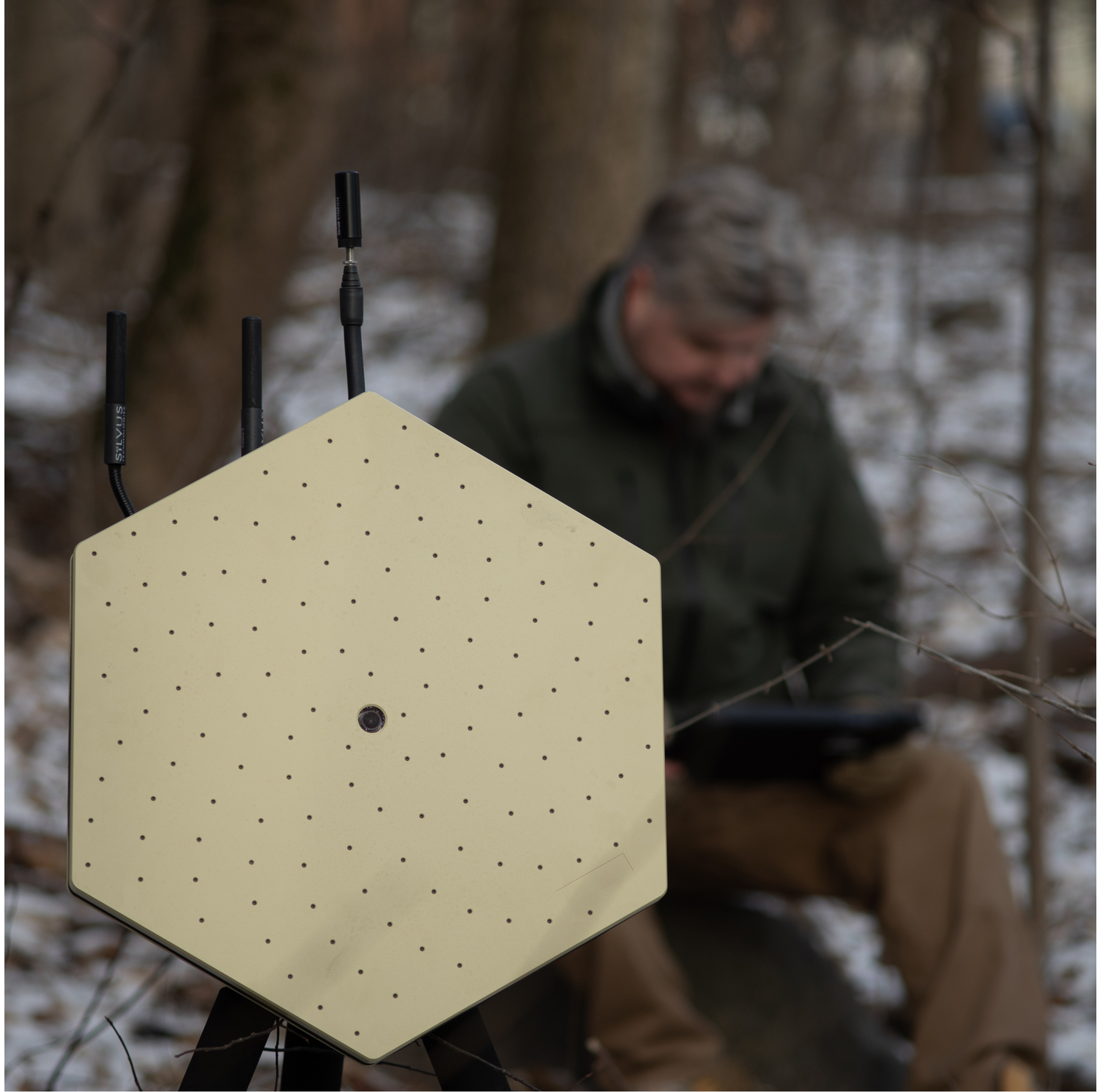
Real-time Safety

Guaranteed by the Compiler!

Anders Schau Knatten


STELUS

C++ Quiz

You've answered 0 of 178 questions correctly. (Clear)

Question #197 Difficulty: ●○○

According to the C++23 standard, what is the output of this program?

```
#include <iostream>

int j = 1;

int main() {
  int& i = j, j;
  j = 2;
  std::cout << i << j;
}
```

Answer: The program is guaranteed to output: Answer

Problems? View a hint or try another question.

I give up, show me the answer (make 3 more attempts first).

Mode : Training

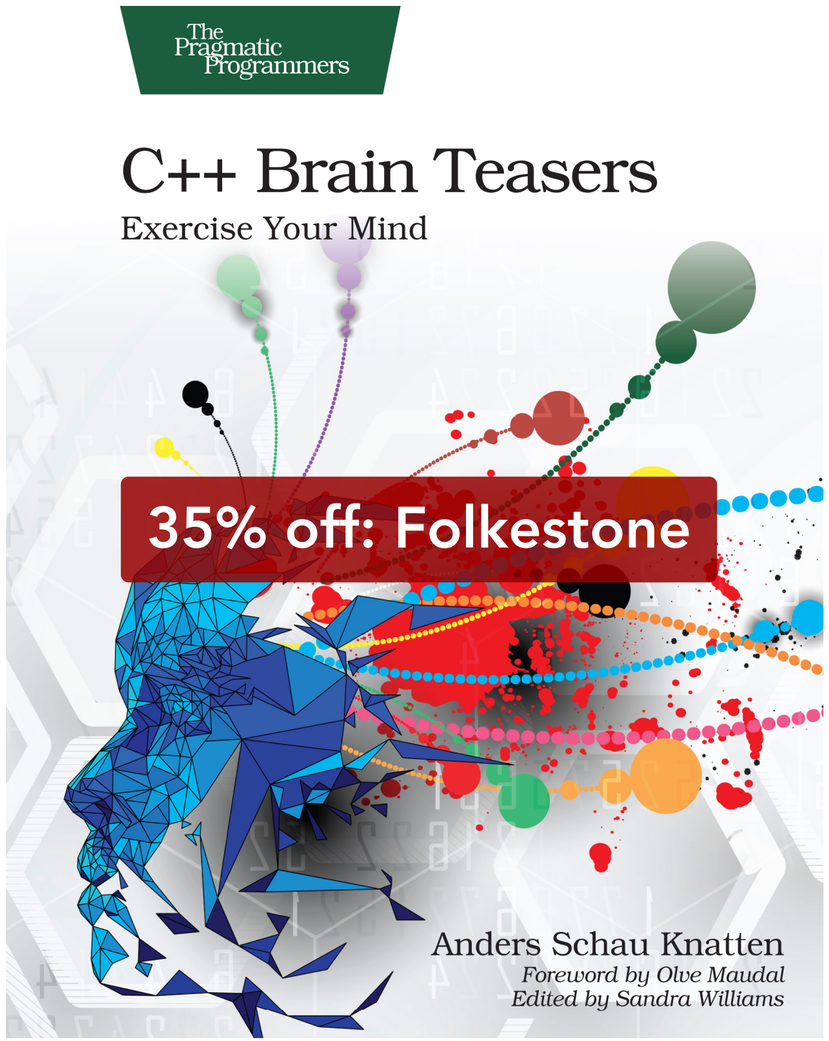
You are currently in training mode, answering random questions. Why not Start a new quiz? Then you can boast about your score, and invite your friends.

Contribute

Create your own!

Android app

Get Sergey Vasilchenko's CppQuiz Android app.


The Pragmatic Programmers
C++ Brain Teasers
Exercise Your Mind
35% off: Folkestone
Anders Schau Knatten
Foreword by Olve Maudal
Edited by Sandra Williams

REAL-TIME SAFETY - GUARANTEED BY THE COMPILER!

DEADLINES

HARD / SOFT REAL-TIME

REAL-TIME == PERFORMANCE?

REAL-TIME != PERFORMANCE

PREDICTABILITY

PREDICTABILITY

- Time ≤ deadline

PREDICTABILITY

- Time ≤ deadline
- Every time!

JUST BENCHMARK IT?

JUST BENCHMARK IT?

- Must not vary!

HOW CAN IT VARY?

MIGHT WANT TO AVOID

- Locks
- Allocations / deallocations
- Exceptions
- Syscalls
- I/O
- Swapping
- Descheduling

NOT EASY!

Dave Rowland @ C++ on Sea 2024

Catching Real-time Safety Violations in C++

https://www.youtube.com/watch?v=n_jeX1s1rkg

CLANG 20

- Function Effect Analysis (FEA)
- RealTimeSanitizer

CLANG 20

- Function Effect Analysis (FEA)
- RealTimeSanitizer

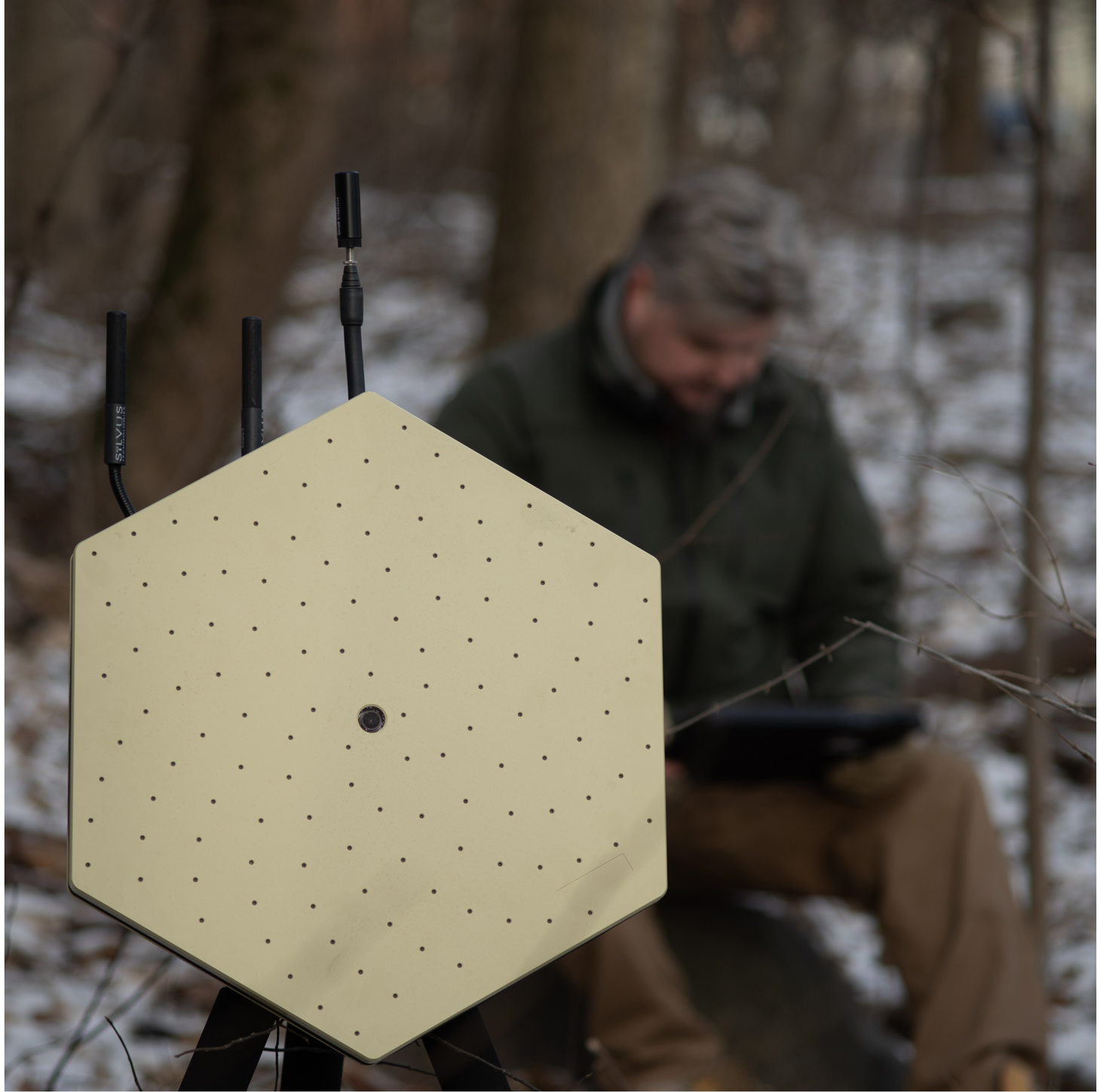
AppleClang 17 (and 21) has FEA but not RTSan

MIGHT WANT TO AVOID

- Locks
- Allocations / deallocations
- Exceptions
- Syscalls
- I/O
- Swapping
- Descheduling

FEA / RTSAN CAN HELP WITH

- Locks
- Allocations / deallocations
- Exceptions
- Syscalls
- I/O
- ~~Swapping~~
- ~~Descheduling~~


STELUS

PERFORMANCE CONSTRAINTS

- Attributes on functions for certain guarantees
- Kinda like `const` or `noexcept`
- Can promise that a function

PERFORMANCE CONSTRAINTS

- Attributes on functions for certain guarantees
- Kinda like `const` or `noexcept`
- Can promise that a function
  - Doesn't throw exceptions

PERFORMANCE CONSTRAINTS

- Attributes on functions for certain guarantees
- Kinda like `const` or `noexcept`
- Can promise that a function
  - Doesn't throw exceptions
  - Doesn't allocate / deallocate

PERFORMANCE CONSTRAINTS

- Attributes on functions for certain guarantees
- Kinda like `const` or `noexcept`
- Can promise that a function
  - Doesn't throw exceptions
  - Doesn't allocate / deallocate
  - Doesn't block on something

PERFORMANCE CONSTRAINTS

- Function Effect Analysis (FEA)
  - Compile time check (`-Wfunction-effects`)

PERFORMANCE CONSTRAINTS

- Function Effect Analysis (FEA)
  - Compile time check (`-Wfunction-effects`)
- RealTimeSanitizer
  - Run-time sanitizer (`-fsanitize=realtime`)

PERFORMANCE CONSTRAINT ATTRIBUTES

PERFORMANCE CONSTRAINT ATTRIBUTES

```
[[clang::nonallocating]]
```

- Will not allocate or deallocate on the heap
- Will not throw or catch exceptions

PERFORMANCE CONSTRAINT ATTRIBUTES

`[[clang::nonallocating]]`

- Will not allocate or deallocate on the heap
- Will not throw or catch exceptions

`[[clang::nonblocking]]`

- All of the above
- Will also not take a lock

FUNCTION EFFECT ANALYSIS (FEA)

```
int get_value() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
int get_value() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
-Wfunction-effects
```

```
int get_value() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
-Wfunction-effects
```

```
warning: function with 'nonallocating' attribute must not
allocate or deallocate memory [-Wfunction-effects]
int* i = new int{42};
         ^
```

```
int get_value() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
-Wfunction-effects
```

```
warning: function with 'nonallocating' attribute must not
allocate or deallocate memory [-Wfunction-effects]
int* i = new int{42};
         ^
```

```
-Wfunction-effects -Werror
```

```
int get_value() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
-Wfunction-effects
```

```
warning: function with 'nonallocating' attribute must not
allocate or deallocate memory [-Wfunction-effects]
int* i = new int{42};
         ^
```

```
-Wfunction-effects -Werror
```

```
error: function with 'nonallocating' attribute must not
allocate or deallocate memory [-Werror,-Wfunction-effects]
int* i = new int{42};
         ^
```

```
int f() [[clang::nonallocating]]
{
    throw 42;
}
```

```
int f() [[clang::nonallocating]]
{
    throw 42;
}
```

```
warning: function with 'nonallocating' attribute must not
throw or catch exceptions
```

```
int g();

int f() [[clang::nonallocating]]
{
    return g();
}
```

```
int g();

int f() [[clang::nonallocating]]
{
    return g();
}
```

```
warning: function with 'nonallocating' attribute must not call
non-'nonallocating' function 'g'
```

```
int g()
{
    return 42;
}

int f() [[clang::nonallocating]]
{
    return g();
}
```

```cpp
int g()
{
    return 42;
}

int f() [[clang::nonallocating]]
{
    return g();
}
```

✅

```
template <typename T>
T simple(T t)
{
    return t;
}

int f(int i) [[clang::nonallocating]]
{
    return simple(i);
}
```

```cpp
template <typename T>
T simple(T t)
{
    return t;
}

std::vector<int> f(std::vector<int> i) [[clang::nonallocating]]
{
    return simple(i);
}
```

```cpp
template <typename T>
T simple(T t)
{
    return t;
}

std::vector<int> f(std::vector<int> i) [[clang::nonallocating]]
{
    return simple(i);
}
```

```
warning: function with 'nonallocating' attribute must not call
non-'nonallocating' function 'simple<std::vector<int>>'

note: function cannot be inferred 'nonallocating' because it calls
non-'nonallocating' destructor 'std::vector<int>::~vector'
```

```
int f() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
int f() [[clang::nonallocating]]
{
    int* i = new int{42};
    return *i;
}
```

```
warning: function with 'nonallocating' attribute must not
allocate or deallocate memory
```

```
int f() [[clang::nonallocating]]
{
    auto i = std::make_unique<int>(42);
    return *i;
}
```

```cpp
int f() [[clang::nonallocating]]
{
    auto i = std::make_unique<int>(42);
    return *i;
}
```

```
warning: function with 'nonallocating' attribute must not call
non-'nonallocating'function 'std::make_unique<int, int, 0>'
```

```
int (*getter)() [[clang::nonallocating]];

int get();

int f()
{
  getter = &get;
  return getter();
}
```

```
int (*getter)() [[clang::nonallocating]];

int get();

int f()
{
  getter = &get;
  return getter();
}
```

```
int (*getter)() [[clang::nonallocating]];

int get();

int f()
{
  getter = &get;
  return getter();
}
```

```
warning: attribute 'nonallocating' should not be added via type conversion
getter = &get;
```

```
struct Base
{
    virtual int get() [[clang::nonallocating]] = 0;
};

struct Derived : public Base
{
    int get() override
    {
        auto* i = new int{42};
        return *i;
    }
};
```

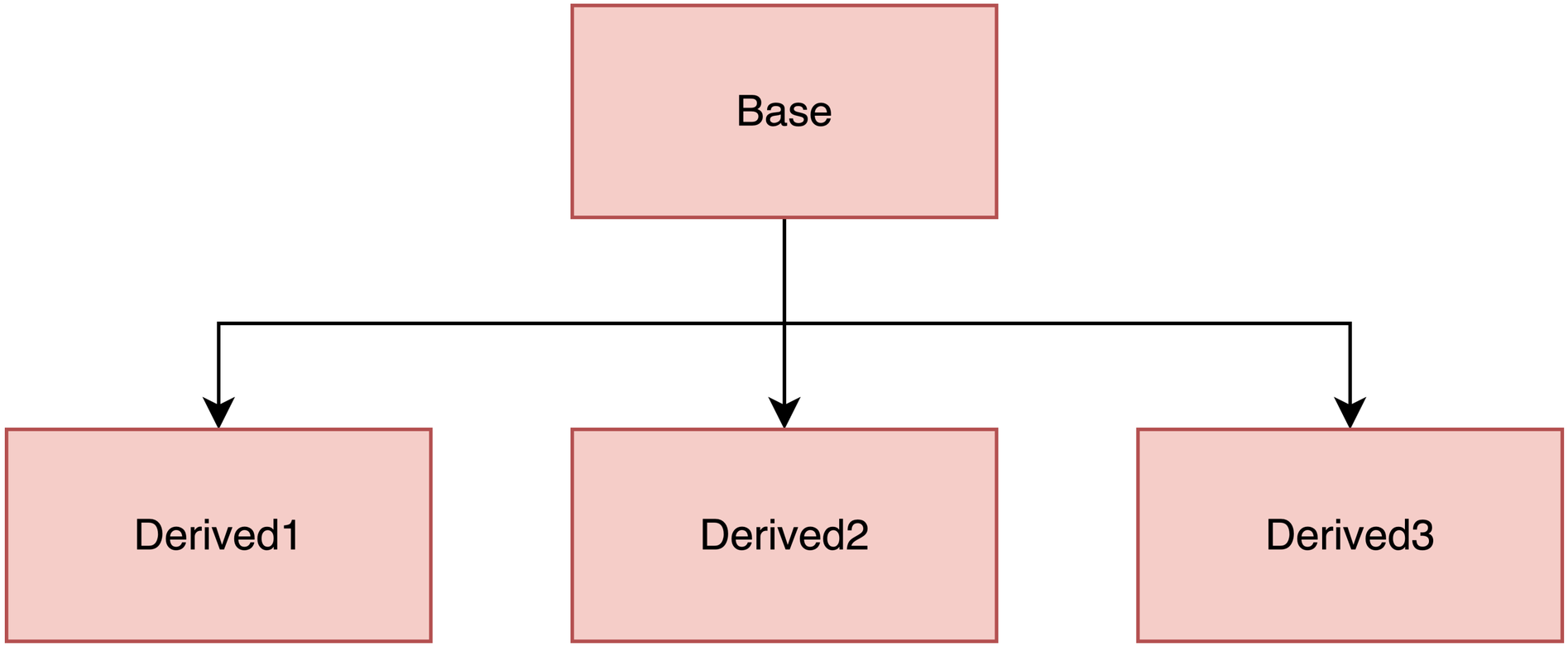
```
struct Base
{
    virtual int get() [[clang::nonallocating]] = 0;
};

struct Derived : public Base
{
    int get() override
    {
        auto* i = new int{42};
        return *i;
    }
};
```

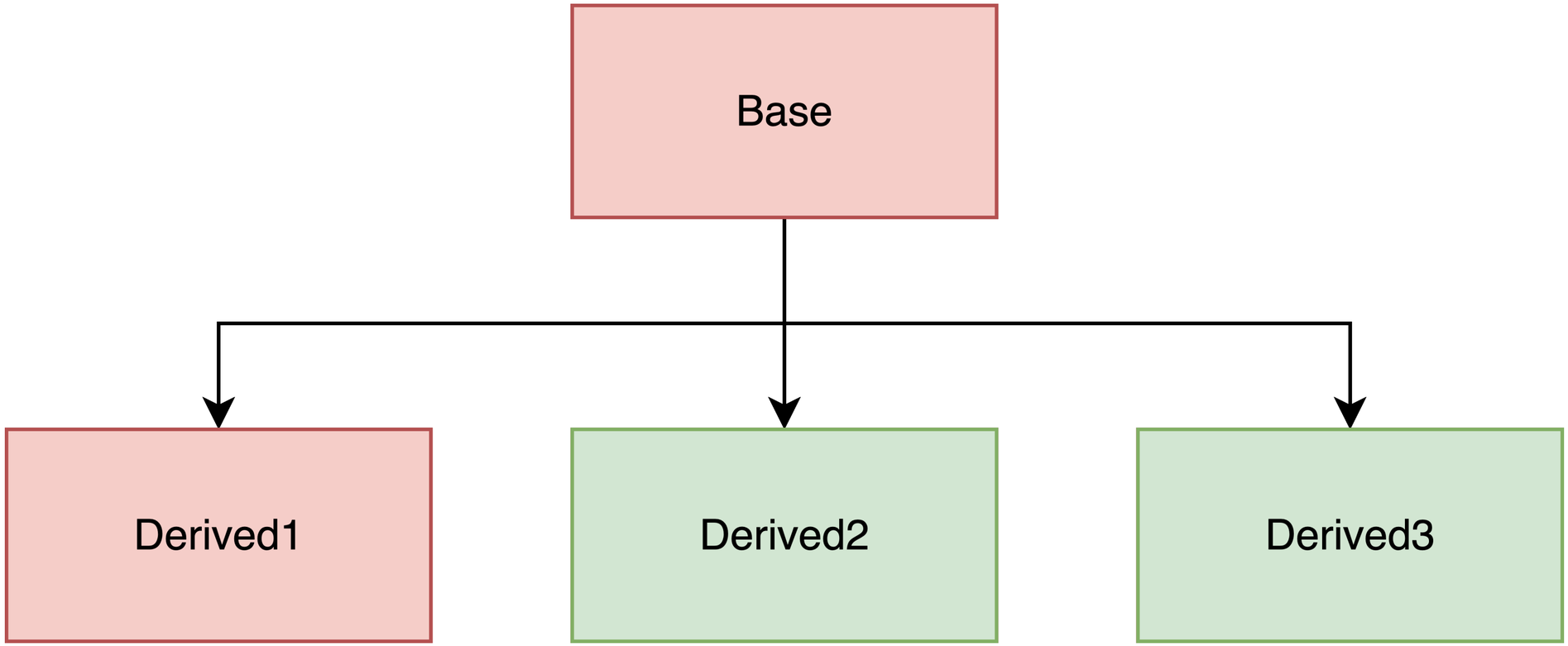
```
1  struct Base
2  {
3      virtual int get() [[clang::nonallocating]] = 0;
4  };
5
6  struct Derived : public Base
7  {
8      int get() override
9      {
10         auto* i = new int{42};
11         return *i;
12     }
13 };
```

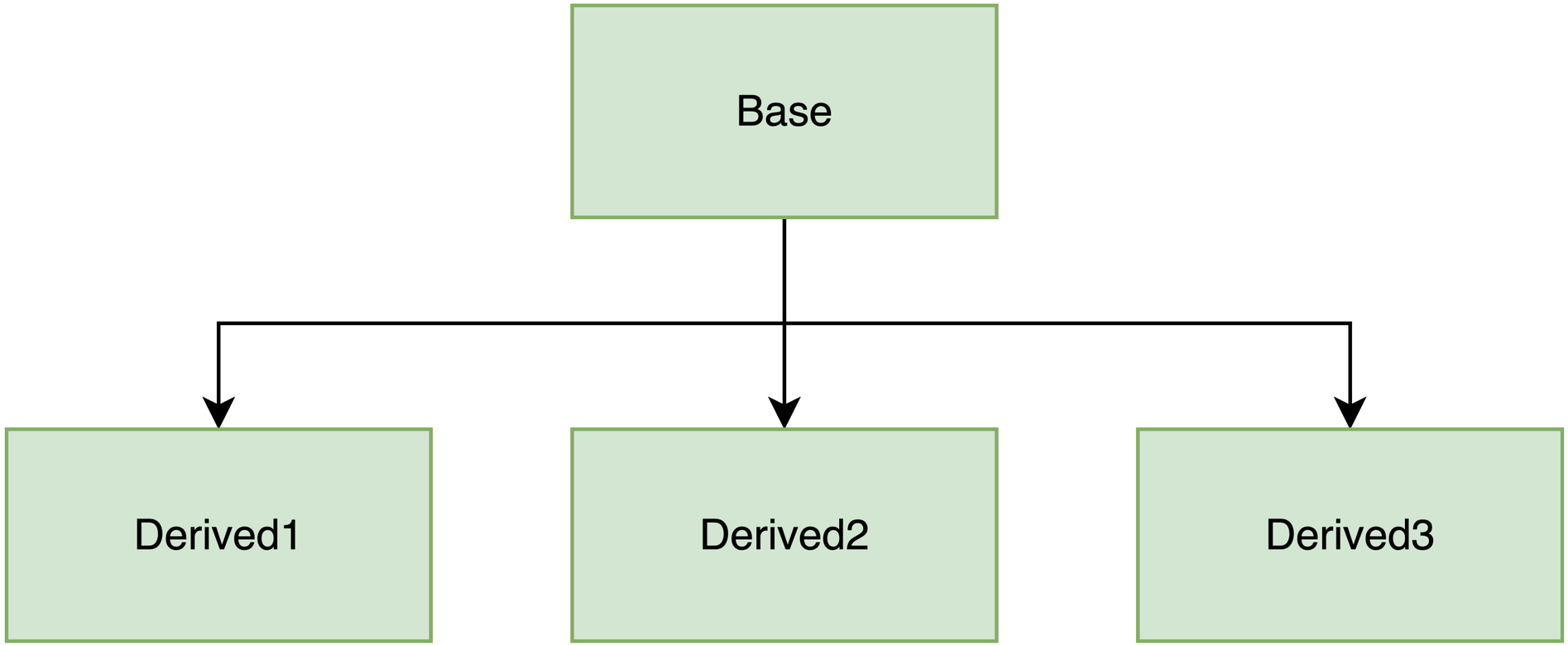
```
warning: function with 'nonallocating' attribute must not
allocate or deallocate memory
auto* i = new int{42};
          ^
```

```
struct Base
{
    virtual int get() = 0;
};

struct Derived : public Base
{
    int get() [[clang::nonallocating]] override
    {
        return 42;
    }
};
```

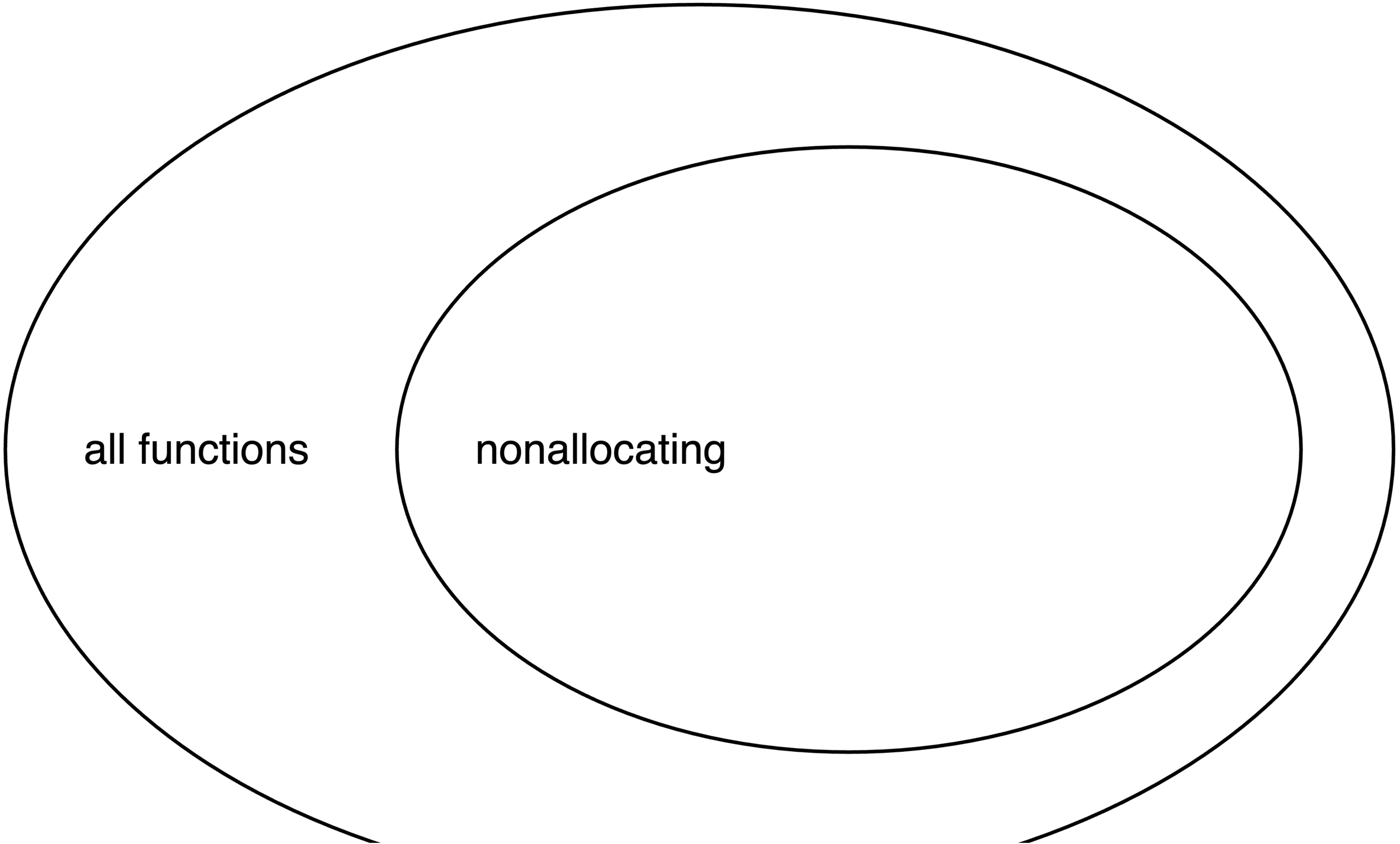
```cpp
struct Base
{
    virtual int get() = 0;
};

struct Derived : public Base
{
    int get() [[clang::nonallocating]] override
    {
        return 42;
    }
};
```


Base
Derived1
Derived2
Derived3

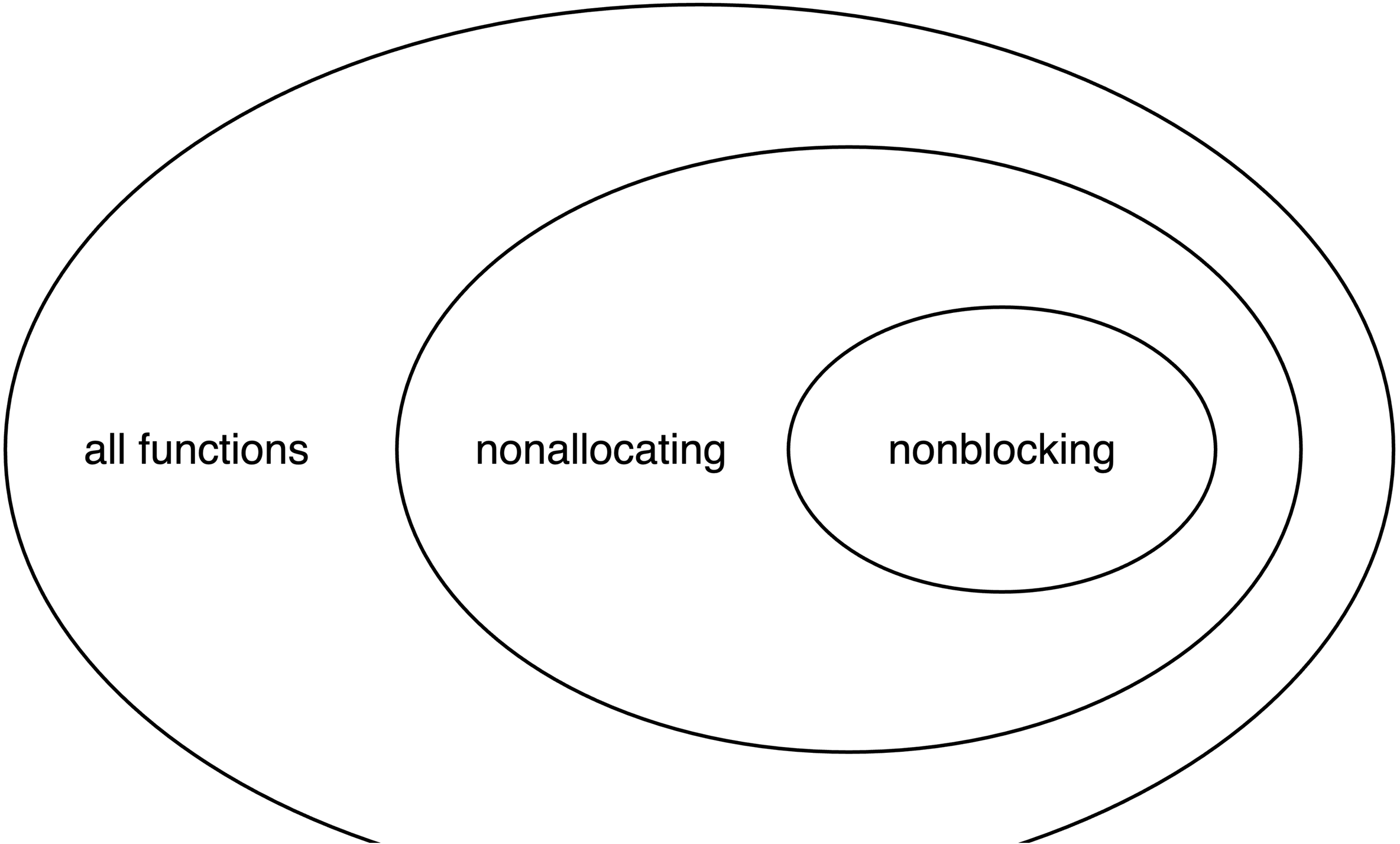

Base
Derived1
Derived2
Derived3


Base
Derived1
Derived2
Derived3

SEMANTICS


all functions
nonallocating

SEMANTICS


all functions
nonallocating
nonblocking

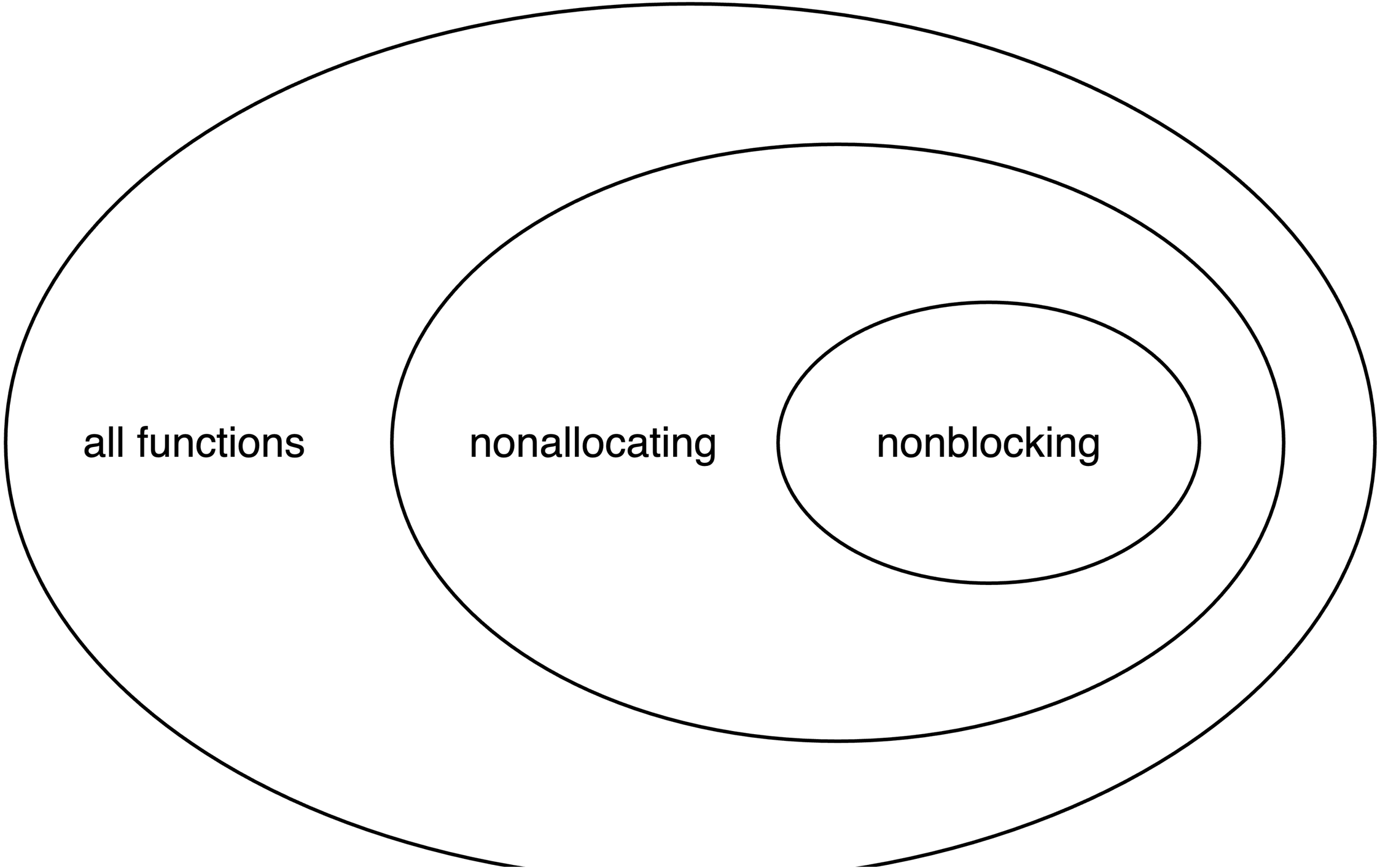
```
int f() [[clang::nonblocking]];

int g() [[clang::nonallocating]]
{
  return f();
}
```

```
int f() [[clang::nonallocating]];

int g() [[clang::nonblocking]]
{
  return f();
}
```

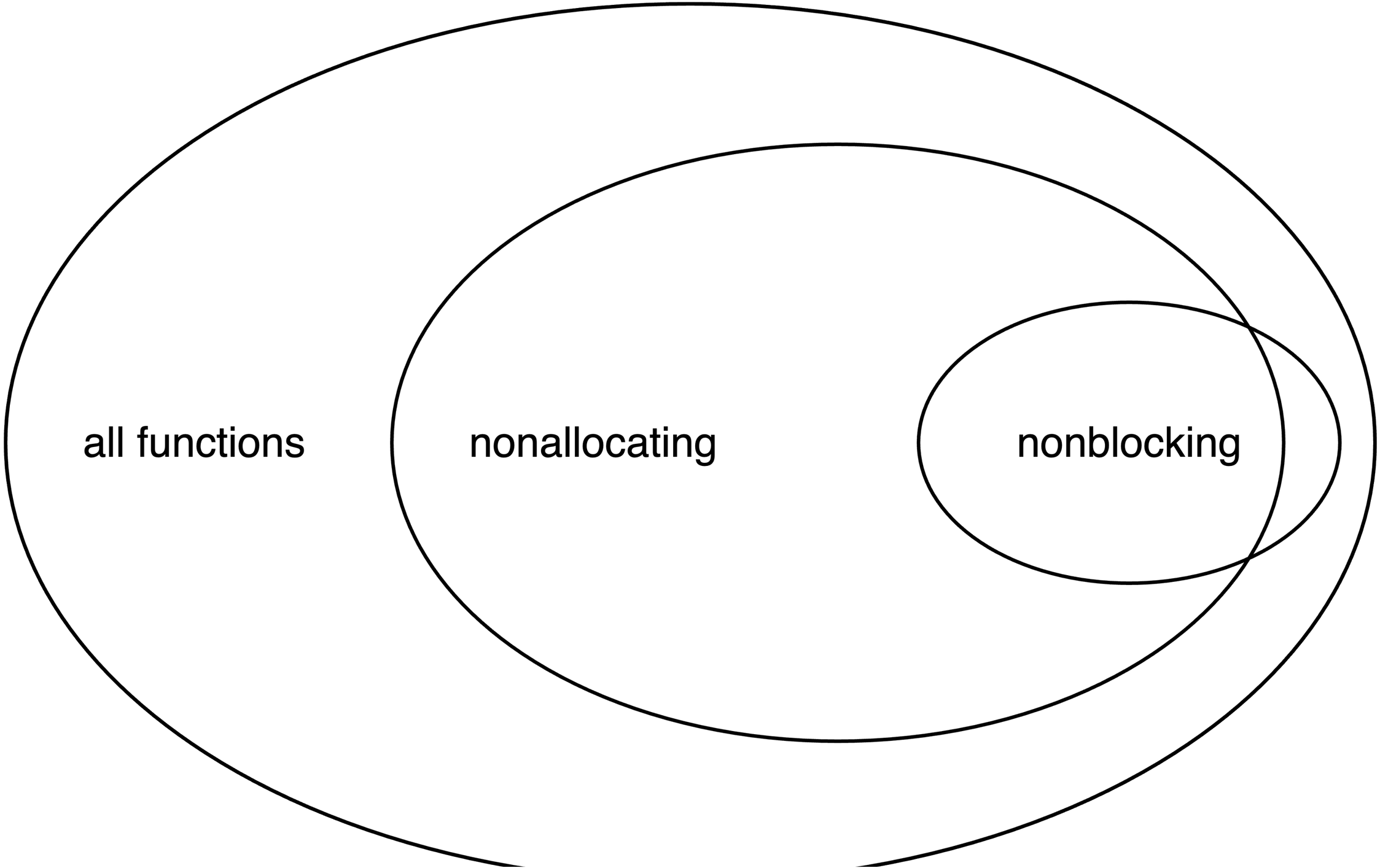
```
int f() [[clang::nonallocating]];

int g() [[clang::nonblocking]]
{
  return f();
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' function 'f'
```

SEMANTICS


all functions
nonallocating
nonblocking

SEMANTICS


all functions
nonallocating
nonblocking

Just use `nonblocking`

Just use `nonblocking`

- Simpler

Just use `nonblocking`

- Simpler
- Locks are worse(?) than allocations

Just use `nonblocking`

- Simpler
- Locks are worse(?) than allocations
- Better compatibility with RealtimeSanitizer

What about `noexcept`?

```
std::string f() noexcept
{
    try
    {
        throw std::runtime_error("ohai");
    } catch (const std::exception& e)
    {
        return e.what();
    }
}
```

```
std::string f() noexcept
{
    throw std::runtime_error("ohai");
}
```

```
void throwy()
{
    throw std::runtime_error("ohai");
}

std::string f() noexcept
{
    throwy();
    return "foo";
}
```

```
void throwy()
{
    throw std::runtime_error("ohai");
}

std::string f() noexcept [[clang::nonblocking]]
{
    throwy();
    return "foo";
}
```

```
void throwy()
{
    throw std::runtime_error("ohai");
}

std::string f() noexcept [[clang::nonblocking]]
{
    throwy();
    return "foo";
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' function 'throwy'

note: function cannot be inferred 'nonblocking'
because it throws or catches exceptions
```

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS:

COMPILERS

COMPILERS

- Clang 20
- AppleClang 17 (FEA only, no RTSan)

OUR COMPILERS

- Clang 20
- AppleClang 17
- *GCC 13*

```
if((CMAKE_CXX_COMPILER_ID STREQUAL "AppleClang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "17.0")
   OR (CMAKE_CXX_COMPILER_ID STREQUAL "Clang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "20.0")
)
    add_compile_options(-Wfunction-effects)
    add_compile_definitions(ENABLE_FEA)
endif()
```

```
if((CMAKE_CXX_COMPILER_ID STREQUAL "AppleClang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "17.0")
   OR (CMAKE_CXX_COMPILER_ID STREQUAL "Clang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "20.0")
)
    add_compile_options(-Wfunction-effects)
    add_compile_definitions(ENABLE_FEA)
endif()
```

```
if((CMAKE_CXX_COMPILER_ID STREQUAL "AppleClang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "17.0")
   OR (CMAKE_CXX_COMPILER_ID STREQUAL "Clang"
      AND CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL "20.0")
)
    add_compile_options(-Wfunction-effects)
    add_compile_definitions(ENABLE_FEA)
endif()
```

```
// FunctionEffectAnalysis.h:

#ifdef ENABLE_FEA
    #define SQ_NONBLOCKING [[clang::nonblocking]]
#else
    #define SQ_NONBLOCKING
#endif
```

```
// FunctionEffectAnalysis.h:

#ifdef ENABLE_FEA
    #define SQ_NONBLOCKING [[clang::nonblocking]]
#else
    #define SQ_NONBLOCKING
#endif
```

```
void f() SQ_NONBLOCKING;
```

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS:

THIRD-PARTY CODE

```
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    libproc(buffer.data(), buffer.size());
}
```

```
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    libproc(buffer.data(), buffer.size());
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' function 'libproc'
```

```cpp
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    libproc(buffer.data(), buffer.size());
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' function 'libproc'
```

```cpp
void process(std::span<char> buffer) [[clang::nonblocking]]
{
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wfunction-effects"
    libproc(buffer.data(), buffer.size());
#pragma clang diagnostic pop
}
```

```
#ifdef ENABLE_FEA
    #define SQ_NONBLOCKING [[clang::nonblocking]]
    #define SQ_FEA_IGNORE(...)                                          \
        _Pragma("clang diagnostic push")                                \
        _Pragma("clang diagnostic ignored \"-Wfunction-effects\"")      \
        __VA_ARGS__ _Pragma("clang diagnostic pop")

#else
    #define SQ_NONBLOCKING
    #define SQ_FEA_IGNORE(...) __VA_ARGS__
#endif
```

```
#ifdef ENABLE_FEA
    #define SQ_NONBLOCKING [[clang::nonblocking]]
    #define SQ_FEA_IGNORE(...)                                          \
        _Pragma("clang diagnostic push")                                \
        _Pragma("clang diagnostic ignored \"-Wfunction-effects\"")      \
        __VA_ARGS__ _Pragma("clang diagnostic pop")

#else
    #define SQ_NONBLOCKING
    #define SQ_FEA_IGNORE(...) __VA_ARGS__
#endif
```

```
#ifdef ENABLE_FEA
    #define SQ_NONBLOCKING [[clang::nonblocking]]
    #define SQ_FEA_IGNORE(...)                                          \
        _Pragma("clang diagnostic push")                                \
        _Pragma("clang diagnostic ignored \"-Wfunction-effects\"")      \
        __VA_ARGS__ _Pragma("clang diagnostic pop")

#else
    #define SQ_NONBLOCKING
    #define SQ_FEA_IGNORE(...) __VA_ARGS__
#endif
```

```
void process(std::span<char> buffer) SQ_NONBLOCKING
{
    SQ_FEA_IGNORE(
        libproc(buffer.data(), buffer.size());
    )
}
```

```
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    SQ_FEA_IGNORE(
        libproc(buffer.data(), buffer.size());
    )
}
```

```
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    SQ_FEA_IGNORE(
        libproc(buffer.data(), buffer.size());
    )
}
```

```
-fsanitize=realtime
```

```
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    SQ_FEA_IGNORE(
        libproc(buffer.data(), buffer.size());
    )
}
```

```
-fsanitize=realtime
```

```
==72564==ERROR: RealtimeSanitizer: unsafe-library-call
Intercepted call to real-time unsafe function `malloc` in real-time context!
    #0 0x0001012932dc in malloc+0x20 (libclang_rt.rtsan_osx_dynamic.dylib:arm64+0x32dc)
    #1 0x00019cbfb9d4 in _malloc_type_malloc_outlined+0x60 (libsystem_malloc.dylib:arm64+0x1d9d4)
    #2 0x00019cd993dc in operator new(unsigned long)+0x30 (libc++abi.dylib:arm64+0x1a3dc)
    #3 0x000100f51258 in libproc(char*, unsigned long)+0x18 (third-party:arm64+0x100001258)
    #4 0x000100f512c0 in process(std::__1::span<char, 18446744073709551615ul>)+0x4c (third-party:arm64+0x1000012c0)
    #5 0x000100f513f8 in main+0x34 (third-party:arm64+0x1000013f8)
    #6 0x00019ca3eb94 in start+0x17b8 (dyld:arm64+0xfffffffffff3ab94)

SUMMARY: RealtimeSanitizer: unsafe-library-call (libsystem_malloc.dylib:arm64+0x1d9d4) in _malloc_type_malloc_outlined+0x60
```

```c
// libproc.h:
void libproc(char *buffer, size_t size);
```

```c
// libproc.h:
void libproc(char *buffer, size_t size);
```

```c
// libproc_redeclare.h:
#include <libproc.h>
void libproc(char *buffer, size_t size) [[clang::nonblocking]];
```

```
// libproc.h:
void libproc(char *buffer, size_t size);
```

```
// libproc_redeclare.h:
#include <libproc.h>
void libproc(char *buffer, size_t size) [[clang::nonblocking]];
```

```
# include <libproc_redeclare.h>
void process(std::span<char> buffer) [[clang::nonblocking]]
{
    libproc(buffer.data(), buffer.size());
}
```

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS:

LAMBDAS, CALLBACKS AND TYPE ERASURE

```
void f() [[clang::nonblocking]]
{
    auto lambda = [](int i) { return i * 2; };
}
```

```
void f() [[clang::nonblocking]]
{
    auto lambda = [](int i) { return i * 2; };
}
```

```
void f() [[clang::nonblocking]]
{
    auto lambda = [](int i) [[clang::nonblocking]] { return i * 2; };
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { new int{}; return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { new int{}; return i * 2; });
}
```

```
warning: function with 'nonblocking' attribute must not call
    non-'nonblocking' function 'std::transform(...)'
note: function cannot be inferred 'nonblocking' because it calls non-'nonblocking' lambda (...)
(...)
note: lambda cannot be inferred 'nonblocking' because it allocates or deallocates memory
                       [](int i) { new int{}; return i * 2; });
                                   ^
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { new int{}; return i * 2; });
}
```

```
warning: function with 'nonblocking' attribute must not call
    non-'nonblocking' function 'std::transform(...)'
note: function cannot be inferred 'nonblocking' because it calls non-'nonblocking' lambda (...)
(...)
note: lambda cannot be inferred 'nonblocking' because it allocates or deallocates memory
                       [](int i) { new int{}; return i * 2; });
                                   ^
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { return i * 2; });
}
```

```cpp
void process(std::vector<int> &ints) [[clang::nonblocking]]
{
    std::transform(ints.begin(), ints.end(), ints.begin(),
                   [](int i) { new int{}; return i * 2; });
}
```

```
warning: function with 'nonblocking' attribute must not call
    non-'nonblocking' function 'std::transform(...)'
note: function cannot be inferred 'nonblocking' because it calls non-'nonblocking' lambda (...)
(...)
note: lambda cannot be inferred 'nonblocking' because it allocates or deallocates memory
                       [](int i) { new int{}; return i * 2; });
                                   ^
```

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS:

LAMBDAS, *CALLBACKS* AND TYPE ERASURE

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
void chill_code()
{
    EvenNumberProcessor processor{[](int i) { std::cout << i << " "; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
void chill_code()
{
    EvenNumberProcessor processor{[](int i) { std::cout << i << " "; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
}
```

```cpp
template <typename Callback>
struct EvenNumberProcessor
{
    Callback callback;

    void process(int i)
    {
        if (i % 2 == 0) { callback(i); }
    }
};
```

```cpp
int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
    return sum;
}
```

```cpp
void chill_code()
{
    EvenNumberProcessor processor{[](int i) { std::cout << i << " "; }};
    for (int i = 0; i < 5; ++i)
    {
        processor.process(i);
    }
}
```

CHALLENGES, LESSONS LEARNED, TIPS & TRICKS:

LAMBDAS, CALLBACKS AND *TYPE ERASURE*

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i);
};

void EvenNumberProcessor::process(int i)
{
    if (i % 2 == 0) { callback(i); }
}

int real_time_code() [[clang::nonblocking]]
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    for (int i = 0; i < 5; ++i) { processor.process(i); }
    return sum;
}

int main()
{
    std::cout << real_time_code();
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' constructor 'std::function(...)'
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    std::function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```
warning: function with 'nonblocking' attribute must not call
non-'nonblocking' function 'std::function<void (int)>::operator()'
   callback(i);
   ^
```

```cpp
template <typename Return, typename... Args>
class nonblocking_function<Return(Args...)> {
public:
    Return operator()(Args... args) const noexcept [[clang::nonblocking]]
    {
        return std::invoke(*mCallable, std::forward<Args>(args)...);
    }
private:
    std::unique_ptr<Concept> mCallable;
}
```

```cpp
template <typename Return, typename... Args>
class nonblocking_function<Return(Args...)> {
public:
    Return operator()(Args... args) const noexcept [[clang::nonblocking]]
    {
        return std::invoke(*mCallable, std::forward<Args>(args)...);
    }
private:
    std::unique_ptr<Concept> mCallable;
}
```

```cpp
template <typename Return, typename... Args>
class nonblocking_function<Return(Args...)> {
public:
    Return operator()(Args... args) const noexcept [[clang::nonblocking]]
    {
        return std::invoke(*mCallable, std::forward<Args>(args)...);
    }
private:
    std::unique_ptr<Concept> mCallable;
}
```

```cpp
template <typename Return, typename... Args>
class nonblocking_function<Return(Args...)> {
public:
    Return operator()(Args... args) const noexcept [[clang::nonblocking]]
    {
        return std::invoke(*mCallable, std::forward<Args>(args)...);
    }
private:
    std::unique_ptr<Concept> mCallable;
}
```

```cpp
struct EvenNumberProcessor
{
    nonblocking_function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```
struct EvenNumberProcessor
{
    nonblocking_function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    nonblocking_function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
struct EvenNumberProcessor
{
    nonblocking_function<void(int)> callback;
    void process(int i) [[clang::nonblocking]];
};

void EvenNumberProcessor::process(int i) [[clang::nonblocking]]
{
    if (i % 2 == 0) { callback(i); }
}

void real_time_code(EvenNumberProcessor &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

int main()
{
    int sum = 0;
    EvenNumberProcessor processor{[&sum](int i) { sum += i; }};
    real_time_code(processor);
    std::cout << sum;
}
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
void real_time_code(EvenNumberProcessor<true> &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

void chill_code(EvenNumberProcessor<false> &processor)
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
void real_time_code(EvenNumberProcessor<true> &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

void chill_code(EvenNumberProcessor<false> &processor)
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
void real_time_code(EvenNumberProcessor<true> &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

void chill_code(EvenNumberProcessor<false> &processor)
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}
```

```cpp
template <bool NonBlocking>
struct EvenNumberProcessor
{
    using CallbackT =
        std::conditional_t<NonBlocking, nonblocking_function<void(int)>,
                           std::function<void(int)>>;
    CallbackT callback;
    void process(int i) [[clang::nonblocking(NonBlocking)]];
};
```

```cpp
void real_time_code(EvenNumberProcessor<true> &processor) [[clang::nonblocking]]
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}

void chill_code(EvenNumberProcessor<false> &processor)
{
    for (int i = 0; i < 5; ++i) { processor.process(i); }
}
```

FINAL TIPS AND TRICKS

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`
- Start "at the bottom"

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`
- Start "at the bottom"
- Start locally while waiting for CI

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`
- Start "at the bottom"
- Start locally while waiting for CI
- Try to only ignore actually safe code

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`
- Start "at the bottom"
- Start locally while waiting for CI
- Try to only ignore actually safe code
- Worst case, can also ignore unsafe code (probably add `__rtsan::ScopedDisabler`)

FINAL TIPS AND TRICKS

- Easiest for regular procedural/functional/OO code
- Can manually annotate `[[clang::blocking]]`
- Start "at the bottom"
- Start locally while waiting for CI
- Try to only ignore actually safe code
- Worst case, can also ignore unsafe code (probably add `__rtsan::ScopedDisabler`)
- Don't use it everywhere!

Real-time Safety

Guaranteed by the Compiler!

Anders Schau Knatten