SAFE STATIC INITIALISATION AND CLEANUP IN LIBRARIES

ASHLEY ROLL

Code and Slides: github.com/AshleyRoll/conferences/cppnorth2024

TOPICS

- Storage Duration and Linkage
- Non-local and local variable initialisation
- How Static Storage Duration is managed by the compiler
- Common Static initialisation patterns
- Libraries and Static Storage initialisation
- Walk-through example library code

ASSUMPTIONS

- C++17 and above
- Not covering `thread_local` variables / `thread` storage duration
- Not covering runtime load and unload of shared libraries
- Trying to strike a balance between *understanding* and *exact*
  - some implementation details glossed over

WHAT IS STATIC INITIALIZATION?

- Storage Duration for values that live outside a scope
- Often Global State
  - Logger
  - Thread pools
  - Resource managers
- Iostream has `std::cin`, `std::cout`, etc
- 3rd Party libraries like Curl and OpenSSL can have startup and shutdown requirements

STORAGE DURATION

- *automatic* - **stack**
  block entered → block exited
- *static* - **fixed memory**
  program begins → program ends
- *dynamic* - **heap**
  `new` → `delete`

```cpp
static int localCounter{0}; // static duration
int globalCounter{10};      // static duration

namespace { // anonymous namespace
    int localValue{1};      // static duration
}

namespace ns {
    int localValue{2};      // static duration
}

void function() {
    int i{10};              // automatic duration
    static int j{0};        // static duration
    int *a =                // "a" automatic duration
            new int[10];    // "object" dynamic duration
    delete [] a;
}

struct jar {
    static int pickle_count;
};
// define once in program
int jar::pickle_count{10};  // static duration

struct bottle {
    // many definitions, linker chooses one
    inline static double litres{0.5};   // static duration
};
```

LINKAGE

- *No Linkage*
  only the current scope can access
- *Internal Linkage*
  only the current translation unit can access
- *External Linkage*
  all translation units can access single instance

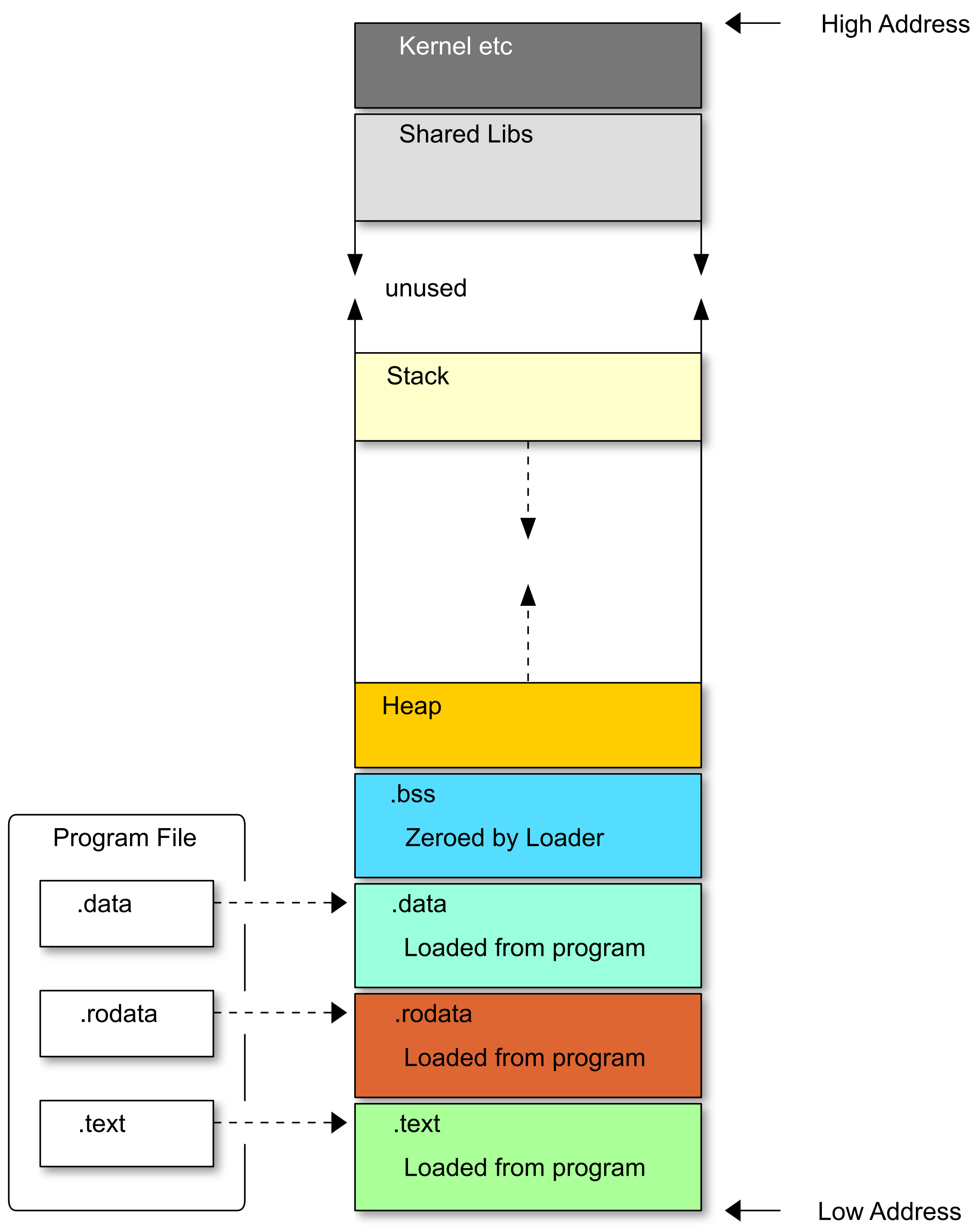
A variable with *Linkage* has a fixed location in the program allocated by the linker. This is done using segments (`.bss`, `.data`, `.rodata`)

```cpp
static int localCounter{0}; // internal linkage
int globalCounter{10};      // external linkage

namespace { // anonymous namespace
    int value{1};           // internal linkage
}

namespace ns {
    int globalValue{2};     // external linkage
    static int value{2};    // internal linkage
}


void function() {
    int i{10};              // no linkage
    static int j{0};        // no linkage
    int *a =                // "a" no linkage
             new int[10];   // "object" no linkage
    delete [] a;
}

struct jar {
    static int pickle_count;
};
// define once in program
int jar::pickle_count{10};  // external linkage

struct bottle {
    // many definitions, linker chooses one
    inline static double litres{0.5};   // external linkage
};
```

DETOUR: PROGRAM LOADING AND RUNNING

- Loader & Kernel setup memory space
- Loader places program segments in memory
- Loader links any shared libraries in
- Loader establishes **Heap** and **Stack**
- Runs program (jumps to `_start` in runtime lib)
  - Initialisation routines run
  - `main()` called
  - Finaliser routines run
  - exit

⚠ Slideware - some simplifications



NON-LOCAL VARIABLES

- Scoped at namespace level, including global namespace
- `class` or `struct` level static variables, including templates
- Have *static* storage duration
- Have *internal* or *external* linkage
- Mapped into fixed location in `.data` (mutable) / `.rodata` (const) or `.bss` sections
- ⚠ *In Principle*[1] ⚠ initialised before program starts

1. Your Mileage May Vary, what most of the rest of this talk is about

LOCAL VARIABLES OF STATIC STORAGE DURATION

- `static` variables inside block scopes
- Have *static* storage duration
- Have *no* linkage
- Mapped into fixed location in `.data` / `.rodata` or `.bss` sections
- Initialised first time control flow passes their declaration[1]
  - Compiler must ensure this is thread-safe
  - Has some minor cost associated each time control passes through
  - Compiler can move initialisation to before program start in some cases

---

1. We will also explore this further

STATIC STORAGE INITIALIZATION IN THE C++ STANDARD

- Static Initialisation
  - Constant initialisation
  - Zero initialisation
- Dynamic Initialisation
- Static local variables

CONSTANT INITIALISATION

- Compile time known constant (not zero)
- Placed into `.data` (or `.rodata`) segments

basic.start.static

ZERO INITIALISATION

- **IF** Compile time known to be zero (`0`, `nullptr`)
- Placed into `.bss` segment

basic.start.static

DYNAMIC INITIALISATION

- **Everything else**
- Placed into `.bss` segment (zeroed); **AND** *Dynamic initialisation* code emitted to compute value
- Ordering is complex:
  - **Unordered** - class template static members
  - **Partially Ordered** - inline static variables IFF consistently ordered in all TUs
  - **Ordered** - all other static storage duration variables in order of definition in TU
- Implementation defined if happens before `main()` or deferred until used (like static local variables)

basic.start.dynamic

⚠ This is a simplification. Hot Take: avoid dynamic initialisation.

STATIC LOCAL VARIABLES

- Standard states initialisation strongly happens before first access; but implementation defined if happens before `main()`
- What really happens:
  - *dynamic initialisation* will happen first time control passes through declaration
  - Requires compiler generated locks for thread safety
  - **Unless** constant/zero initialised, which happens before `main()`

basic.start.dynamic

STATIC STORAGE COMPILER IMPLEMENTATION

- How does a compiler/linker make this work?
- Will look at *gcc* and *Clang*, but essentially same in all compilers
- Goal is to understand why initialisation is specified this way

Sorry, some assembly code follows…

CONSTANT INITIALISATION

example.cpp

```cpp
// both static duration
int globalCounter{10};           // external linkage
static int localCounter{11};     // internal linkage
```

compiler-explorer.com/z/4bjoTnGqY

Values are already set, after program is loaded before any code runs because it is written directly into the program file

x86-64 Asm - gcc

```asm
    .globl  globalCounter    ; export symbol
    .data                    ; use .data segment
globalCounter:
    .long   10               ; globalCounter=10

localCounter:
    .long   11               ; localCounter=11
```

ZERO INITIALISATION

example.cpp

```
// both static duration
int defaultInitInt;
int zeroInitInt{0};

void *defaultInitPtr;
int *nullInitPtr{nullptr};
```

compiler-explorer.com/z/bn7T4Po99

Values are zeroed automatically by the loader before any code runs. The `.bss` segment is not actually stored in the program file, just the required size.

x86-64 Asm - gcc

```
    .globl  nullInitPtr     ; export symbol
    .bss                    ; use .bss segment
nullInitPtr:
    .zero   8               ; nullInitPtr = 0/nullptr

    .globl  defaultInitPtr  ; export symbol
defaultInitPtr:
    .zero   8               ; defaultInitPtr = 0/nullptr

    .globl  zeroInitInt; export symbol
zeroInitInt:
    .zero   4               ; zeroInitInt = 0

    .globl  defaultInitInt  ; export symbol
defaultInitInt:             ; defaultInitInt = 0
    .zero   4
```

re-ordered by alignment?

DYNAMIC INITIALISATION

example.cpp

```cpp
extern int externalInteger;
int globalInteger1{externalInteger};
int globalInteger2{externalInteger};
```

compiler-explorer.com/z/ja8G3b3jd

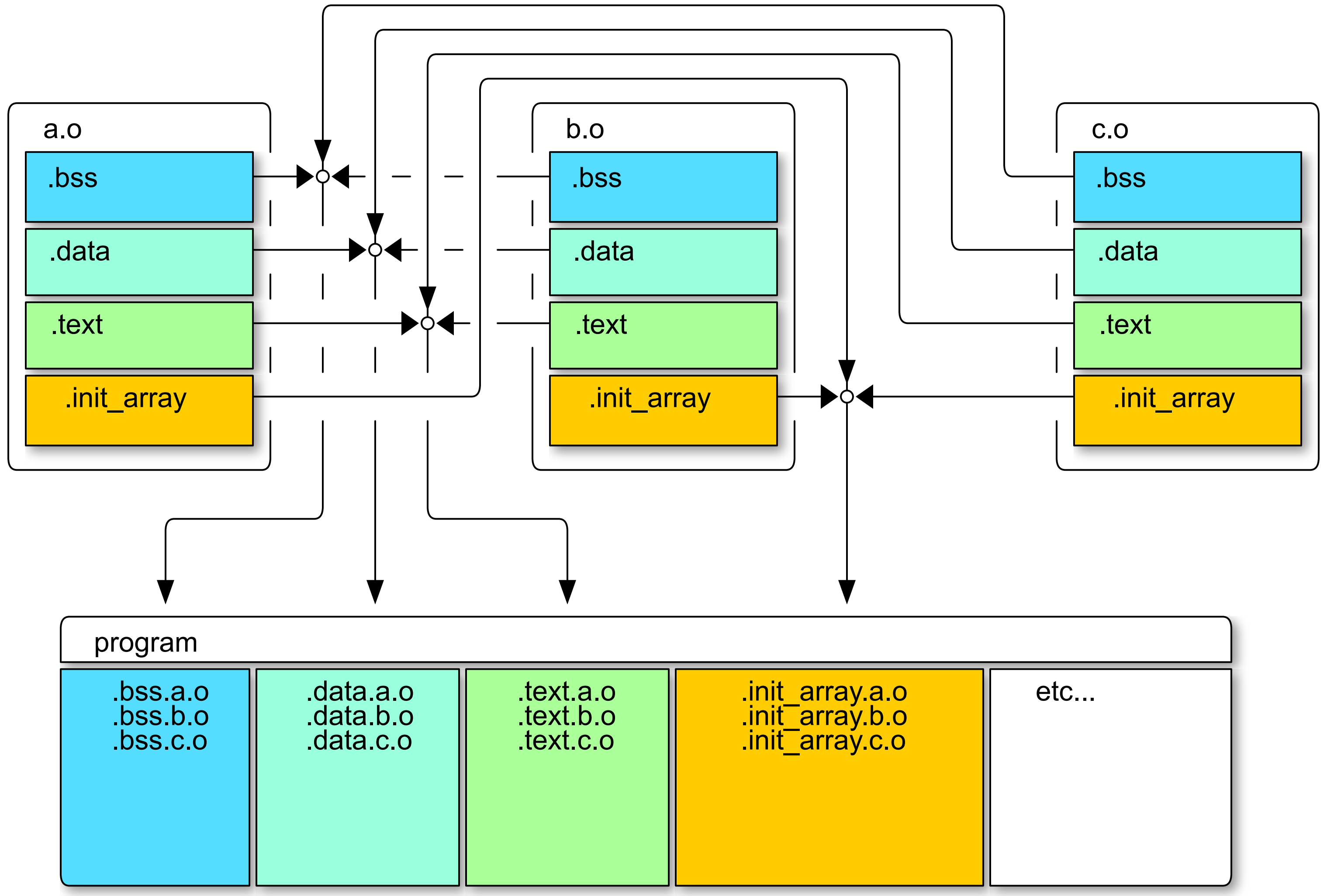
- Compiler can't determine the value
- Zero initialises the variables
- Writes function to compute value at run time. May register destructor with `std::atexit()`
- Adds that function to `.init_array` list to be called by startup code

x86-64 Asm - Clang

```asm
    .text                       ; use code segment
_GLOBAL__sub_I_example.cpp:
    mov     rax, qword ptr [rip + externalInteger@GOTPCREL]
    mov     eax, dword ptr [rax]
    ; Initialise variables, in order from TU
    mov     dword ptr [rip + globalInteger1], eax
    mov     dword ptr [rip + globalInteger2], eax
    ret

    .bss                        ; use .bss segment
    .globl  globalInteger1      ; export symbol
globalInteger1:
    .long   0

    .globl  globalInteger2      ; export symbol
globalInteger2:
    .long   0

    ; place the address of _GLOBAL__sub_I_example into
    ; a section called .init_array
    ; Startup code calls all these before main()
    .section .init_array,"aw",@init_array
    .quad   _GLOBAL__sub_I_example.cpp
```

DETOUR: LINKING

- Linker gathers all the sections and puts them together in program



- Order of object file sections depends on link order
- Link order depends on **???** (build system, configuration, etc..)

DETOUR: START UP `.init_array`

- Implementation supplied start up code:
  - Establishes environment
  - Runs initialisers in `.init_array`
  - calls `main()`
- Remember, initialiser order is fragile/unpredictable!
  - Dynamic initialisation can happen in unpredictable order!
- `std::atexit()` has list of calls to run after `main()`
  - Reverse order of registration

Artist’s impression

```
// pointer to function returning void and taking
// no arguments.
using void_fn = void(*)(void);

// provided by linker from segment bounds
extern void_fn *init_array_start;
extern void_fn *init_array_end;

extern void setup_environment();
extern int main();

std::vector<void_fn> exit_list{};
void atexit(void_fn f) { exit_list.push_back(f); }

int start()
{
    setup_environment();

    std::span init{init_array_start, init_array_end};
    for(auto *fn : init) {
        fn();
    }

    auto exit = main();

    for(auto *fn : exit_list | std::ranges::views::reverse) {
        fn();
    }

    return exit;
}
```

STATIC LOCAL VARIABLES - CONSTANT INITIALISATION

example.cpp

```
int& get_constant()
{
    static int s_const{42}; // or 0
    return s_const;
}
```

compiler-explorer.com/z/q94rh66Pd

- Same as constant initialisation of any other variable
- Also works for zero initialisation

x86-64 Asm - Clang

```
    .text                        ; use .text (code) segment
    .globl  get_constant()       ; export function
get_constant():
    lea     rax, [rip + get_constant()::s_const]
    ret

    .data                        ; use .data segment
get_constant()::s_const:
    .long   42                   ; get_constant()::s_const = 42
```

```
    ; get_constant() as above

    .bss                        ; use .bss segment
get_constant()::s_const:
    .long   0                   ; get_constant()::s_const = 0
```

STATIC LOCAL VARIABLES - DYNAMIC INITIALISATION

example.cpp

```cpp
extern int externalValue;

int& get_dynamic()
{
    static int s_dyn{42*externalValue};
    return s_dyn;
}
```

compiler-explorer.com/z/5zMbac9cE

- Effectively initialises first time control passes the definition
- Compiler creates a guard variable
- Double check locking
- Costs single check even after initialised
- `__cxa_guard_acquire` and `__cxa_guard_release` generally involve global mutex operation

x86-64 Asm - Clang

```asm
    .text
    .globl  get_dynamic()    ; export function
get_dynamic():
    movzx   eax, byte ptr [rip + guard for get_dynamic()::s_dyn]
    test    al, al           ; check if already initialised
    je      .LBB0_1
    lea     rax, [rip + get_dynamic()::s_dyn]
    ret                      ; return the value

.LBB0_1:
    push    rax
    lea     rdi, [rip + guard for get_dynamic()::s_dyn]
    call    __cxa_guard_acquire@PLT
    test    eax, eax         ; double check if initialised
    je      .LBB0_3          ; skip if it was
    mov     rax, qword ptr [rip + externalValue@GOTPCREL]
    imul    eax, dword ptr [rax], 42
    mov     dword ptr [rip + get_dynamic()::s_dyn], eax
    lea     rdi, [rip + guard for get_dynamic()::s_dyn]
    call    __cxa_guard_release@PLT
.LBB0_3:
    add     rsp, 8
    lea     rax, [rip + get_dynamic()::s_dyn]
    ret                      ; return value

    ; local .data segment allocations
    .local  get_dynamic()::s_dyn
    .comm   get_dynamic()::s_dyn,4,4
    .local  guard for get_dynamic()::s_dyn
    .comm   guard for get_dynamic()::s_dyn,8,8
```

CONSTRUCTORS AND DESTRUCTORS FOR STATIC STORAGE DURATION

- Dynamic Initialisation for Non Trivial types (with constructors and destructors)
- With possible exception for `constexpr` constructors
- The destructor will be registered with `std::atexit()` when constructed

CONSTRUCTOR AND DESTRUCTOR EXAMPLE

example.cpp

```cpp
struct S
{
    S();
    ~S();
};

S s;
```

- Familiar Dynamic Initialisation pattern calls `ctor`
  - Registers `dtor` with `std::atexit()` (`__cxa_atexit`)
- Similar with local variables of static duration
  - Registers `dtor` after construction

compiler-explorer.com/z/1af88d3s6

x86-64 Asm - Clang

```asm
    .text                           ; use code segment
_GLOBAL__sub_I_example.cpp:
    push    rbx
    ; call constructor
    lea     rbx, [rip + s]
    mov     rdi, rbx
    call    S::S()@PLT
    ; register destructor
    mov     rdi, qword ptr [rip + S::~S()@GOTPCREL]
    lea     rdx, [rip + __dso_handle]
    mov     rsi, rbx
    pop     rbx
    jmp     __cxa_atexit@PLT ; tail call

    .bss                            ; use .bss segment
    .globl  s                       ; export symbol
s:
    .zero   1                       ; must have unique address

    ; place the address of _GLOBAL__sub_I_example into
    ; a section called .init_array
    ; Startup code calls all these before main()
    .section        .init_array,"aw",@init_array
    .quad   _GLOBAL__sub_I_example.cpp
```

SO, WHAT DOES THIS MEAN?

- Constant or Zero initialisation is predictable and reliable
- Dynamic initialisation is much harder:
  - Different types of initialisation, even in the same file, may not be predictably ordered
  - All bets are off across TUs
    - The linker will choose
    - The order will likely be based on argument order in the link command
    - The order can change arbitrarily
    - It *might* work now, but can easily change

This is where the *Static Initialisation Order Fiasco* comes from!

COMMON INITIALIZATION PATTERNS

- Meyers' Singleton (Construct on first use)
- Nifty Counter
- Static buffer and Reference

MEYERS' SINGLETON

- Very common Singleton pattern
- Can be free function or class member
- Initialisation on first use

```cpp
class Singleton
{
public:
    static Singleton& instance()
    {
        static Singleton instance{};
        return instance;
    }

private:
    Singleton();
    ~Singleton();
};

class log { /* ... */ };

log& get_log()
{
    static log theLog{};
    return theLog;
}
```

DEPENDENT SINGLETONS?

```
struct Log
{
    static Log& get() {
        static Log instance{}; return instance;
    }

    void write(std::string_view msg) { fmt::print("LOG: {}\n", msg); }

    Log() { fmt::print("Log::Log();\n"); }
    ~Log() { fmt::print("Log::~Log();\n"); }
};

struct Singleton
{
    static Singleton& get() {
        static Singleton instance{}; return instance;
    }

    void test() { Log::get().write("Singleton::test();"); }

    Singleton() {
        fmt::print("Singleton::Singleton()\n");
        Log::get().write("in Singleton ctor");
    };
    ~Singleton() {
        fmt::print("Singleton::~Singleton()\n");
        Log::get().write("in Singleton dtor");
    };
};

int main() { Singleton::get().test(); }
```

Looks Good; ship it!

```
Singleton::Singleton()
Log::Log();
LOG: in Singleton ctor
LOG: Singleton::test();
Singleton::~Singleton()
LOG: in Singleton dtor
Log::~Log();
```

compiler-explorer.com/z/cv53nq7cz

DEPENDENT SINGLETONS...

```cpp
struct Log
{
    static Log& get() {
        static Log instance{}; return instance;
    }

    void write(std::string_view msg) { fmt::print("LOG: {}\n", msg); }

    Log() { fmt::print("Log::Log();\n"); }
    ~Log() { fmt::print("Log::~Log();\n"); }
};

struct Singleton
{
    static Singleton& get() {
        static Singleton instance{}; return instance;
    }

    void test() { Log::get().write("Singleton::test();"); }

    Singleton() {
        fmt::print("Singleton::Singleton()\n");
        /* *** Log::get().write("in Singleton ctor"); *** */
    };
    ~Singleton() {
        fmt::print("Singleton::~Singleton()\n");
        Log::get().write("in Singleton dtor");
    };
};

int main() { Singleton::get().test(); }
```

```
Singleton::Singleton()
Log::Log();
LOG: Singleton::test();
Log::~Log();
Singleton::~Singleton()
LOG: in Singleton dtor
```

- `Log` use after destruction!
- `std::atexit()` ordering problem
- Worked because `Log` was constructed before `Singleton` finished construction
- Now, `Log` is constructed after `Singleton`

⚠ Avoid Dependencies -

Especially in `ctor/dtor`

NIFTY COUNTER

- Allows us to take direct control over lifetime
- Injects an *Internal Linkage* object of *Static Storage* duration into each TU that includes the header
- Injection happens before any use of dependencies - Ordered init in TU
- Nifty counter object `ctor`/`dtor` maintain reference count
- first `ctor` call constructs dependencies
- last `dtor` call destroys dependencies
- `<iostream>` uses this for `std::cout`, `std::cin` etc.

NIFTY COUNTER EXAMPLE

nifty.hpp

```cpp
#include <vector>

static struct nifty_counter {
    nifty_counter();
    ~nifty_counter();
} const NiftyCounterInstance;

// the global resource
extern std::vector<int> *someVector;
```

main.cpp

```cpp
#include "nifty.hpp"
#include <vector>

int main() {
    return someVector->size();
}
```

compiler-explorer.com/z/v4K5rqjhM

nifty.cpp

```cpp
#include "nifty.hpp"
#include <vector>

static int instance_count{0};            // zero init
std::vector<int> *someVector{nullptr};  // zero init

nifty_counter::nifty_counter() {
    if(instance_count++ == 0) {
        someVector = new std::vector{1, 2, 3, 4};
    }
}

nifty_counter::~nifty_counter() {
    if(--instance_count == 0) {
        delete someVector;
        someVector = nullptr;
    }
}
```

STATIC BUFFER & REFERENCE

- Allows a global reference to object
- Reserve a buffer to create object within
- Reference is constant initialised to point to buffer
- use *Nifty Counter* to control initialisation

EXAMPLE

log.hpp

```cpp
struct Log {
    Log();
    ~Log();
    int test() { return 42;};
};
extern Log& log;

static struct LogInitialiser {
    LogInitialiser();
    ~LogInitialiser();
} const logInitialiser;
```

main.cpp

```cpp
#include "log.hpp"

int main() {
    return log.test();
}
```

compiler-explorer.com/z/75Yn5jTqr

log.cpp

```cpp
#include <new>          // placement new
#include <type_traits>  // aligned_storage

// internal linkage
static int init_counter;
static std::aligned_storage_t<sizeof(Log), alignof(Log)> logBuf;

// external linkage
Log& log = reinterpret_cast<Log&>(logBuf);

Log::Log() {}
Log::~Log() {}

LogInitialiser::LogInitialiser() {
    if (init_counter++ == 0) {
        new (&log) Log();
    }
}

LogInitialiser::~LogInitialiser() {
    if (--init_counter == 0) {
        log.~Log();
    }
}
```

LINKING AND LOADING LIBRARIES

- What does the C++ standard say about linking and loading libraries?
  - Nothing - just the *Abstract Machine*
  - Everything else is *Implementation Defined*

STATICALLY LINKED LIBRARIES

- Just a container of segments and symbols `.a` / `.lib`
- Linker adds them to the executable
- This includes the `.data` and `.bss` segments for constant/zero initialisation
- Also the `.init_array` segment for dynamic initialisation
- Therefore same ordering issues

SHARED LIBRARIES (DYNAMICALLY LINKED LIBRARIES)

- Linker flags that the `.so` / `.dll` should be loaded at run time
- Program loader does this and wires up symbols
- Compiler must ensure that initialisation occurs **as if** it was running on the *Abstract Machine*
- Program Loader doesn't know anything about C++ rules
- Can be loaded/unloaded at run time as well

PLATFORM SPECIFIC HOOKS

> ⚠ If you are using this, you're well outside of C++ standard — beyond the scope of this session!

- Linux
  - `__attribute__((constructor))` / `__attribute__((destructor))`
  - Any number of functions placed in `.ctors` & `.dtors` segments — just like `.init_array`
  - Ordering of calls is undefined
  - `_init()` and `_fini()` can be used to run specific code
- Windows
  - Must run all initialisers on load, finalisers on unload
  - `DllMain()` is called on process / thread attach / detach to allow you to run specific code

LIBRARY STATIC INITIALISATION

- How can we ensure proper a safe initialisation in a library?
  - What can we rely on?
  - Cross platform?

WHAT CAN WE RELY ON?

- Zero Initialisation
- Constant Initialisation
- ~~Dynamic Initialisation~~
- Ordered initialisation within each TU
- Nifty Counter
- Meyers' Singleton - without interdependencies
- Class Member initialisation order

SOME IDEAS

- Several approaches are possible
- A simple library might get away with a simple approach
- Combinations for more complex scenarios

IDEA 1: CLIENT CALLS FUNCTIONS

Provide `start()` and `stop()` methods that client calls to initialise state and clean up

PROS:

- Simple & Traditional - many existing C libraries
- Allocate and initialise (library) global state in `start()`
- Destroy and de-allocate (library) global state in `stop()`
- Can inject configuration into start up

CONS:

- Easy for client to forget to call one or both
- Generally run-time errors
- Is it valid to `start()` and `stop()` multiple times?
- What about threads?

IDEA 2: NIFTY COUNTER TO INITIALISE

Use a Nifty Counter to automatically call `start()` and `stop()` like functions

PROS:

- Automatic by including library header
- STL uses this for `<iostream>`
- Initialisation happens before `main()`

CONS:

- Can only initialise before `main()` and destroy after - program lifetime
- No opportunity for configuration to be injected

IDEA 3: ONE SINGLETON TO RULE THEM ALL

Wrap all (library) global state into a container object and Access it through Meyers' Singleton

PROS:

- Nothing for client to do
- All initialisation happens in one place - avoid singleton interdependencies
- Container class can order initialisation and pass dependencies between members

CONS:

- Lacks precise control on timing
- No opportunity for configuration to be injected
- Can couple many parts of the library to one TU (can be mitigated)

IDEA 4: RAII LIBRARY HANDLE

- Library exposes top level handle; client controls lifetime explicitly
- All Api calls go through the handle object
- Manages (library) global state lifetime

PROS:

- Client can choose to have one or many non-overlapping lifetimes
- Intuitive if all (top level) calls are handle member functions
- Can inject configuration into start up
- Can also create non-static dependencies, threads, queues etc.

CONS:

- Client can store returned objects outside of lifetime scope
  - Use-after-free: prefer Value Semantics
  - Not specific to this option
- Can couple many parts of the library to one TU (use compilation firewalls — PIMPL)

LIBRARY EXAMPLE IMPLEMENTATION

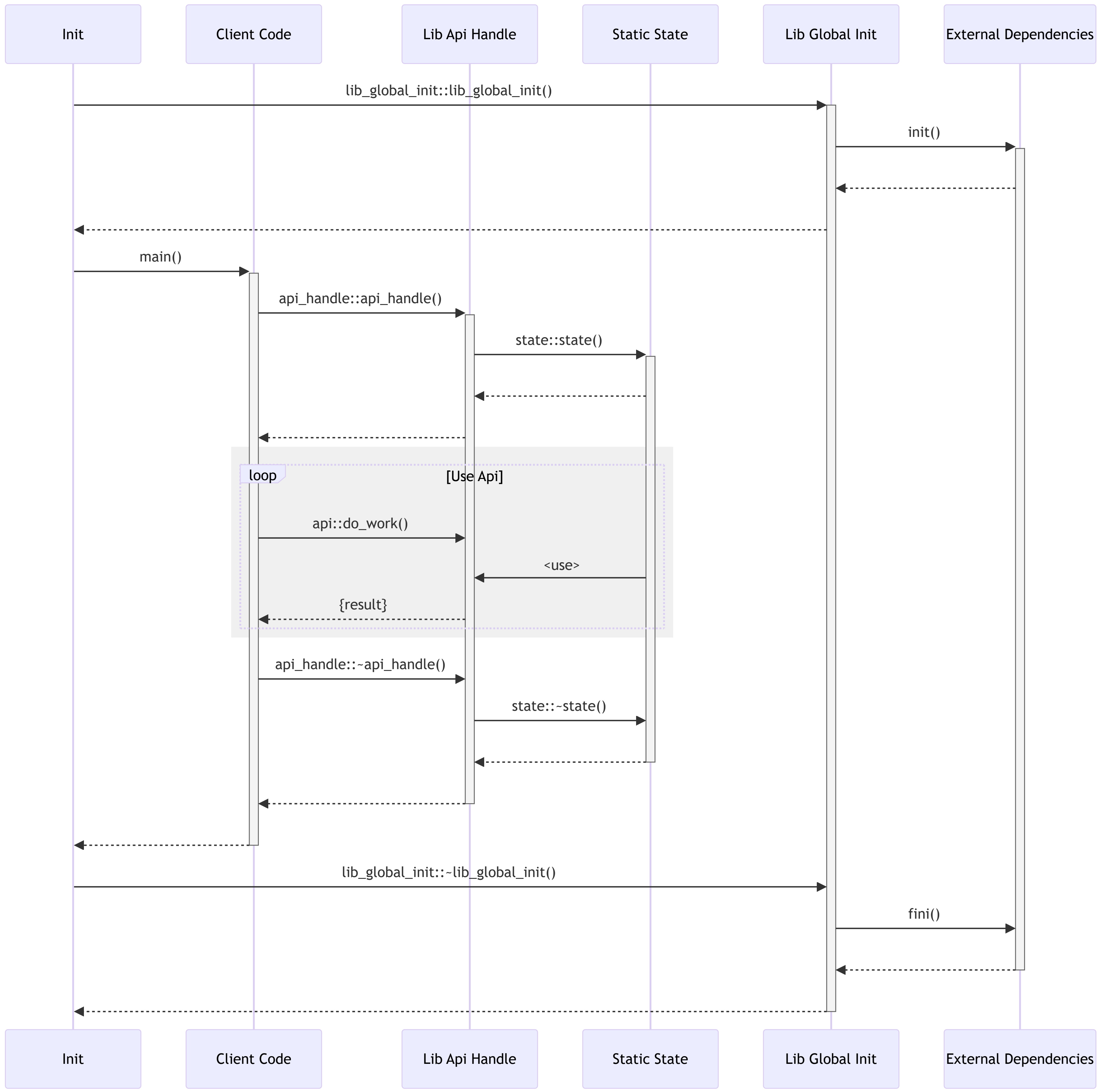
Walk-through of a worked example, imaginatively named `libExample`.

- Uses RAII Handle Object to manage lifetimes with single instance validation
- Uses Nifty Counter for one-time initialisation
- Uses PIMPL idiom for compilation fire-walling implementation
- Separates lifetime management from (internal) global access

Code is simplified for slides, but full working example:

github.com/AshleyRoll/conferences/cppnorth2024/code

LIBRARY EXAMPLE LIFETIME



LIBEXAMPLE FUNCTIONALITY

- Lets have a worker thread that pulls items from a queue
- The client adds work to the queue
- There is a logger that is accessible to the worker and other components
- Not shown in presentation, but available in code example
  - `queue` is a simple thread safe queue which provides a producer and consumer object
  - `worker` is a wrapper over `std::jthread` which takes a queue consumer

LIBEXAMPLE API - RAII OBJECT

example_api.hpp

```cpp
// forward declare PIMPL abstraction for example_api
class example_api_impl;

// The public interface and RAII handle to the
// example library.
class example_api
{
public:
  example_api();
  ~example_api();

  // disable copy and only allow move for the lifetime
  // management of the API resources.
  example_api(example_api &) = delete;
  auto operator=(example_api const &)
    -> example_api & = delete;
  example_api(example_api &&) = default;
  auto operator=(example_api &&)
    -> example_api & = default;

  // Expose public API here, and delegate to the
  // PIMPL class internally
  void do_work(std::string const &message);

private:
  std::unique_ptr<example_api_impl> m_impl;  // PIMPL
};
```

example_api.cpp

```cpp
example_api::example_api()
  : m_impl{ std::make_unique<example_api_impl>() }
{
  fmt::println("example_api::ctor()");
}

example_api::~example_api()
{
  fmt::println("example_api::~dtor()");
}

void example_api::do_work(std::string const &message)
{
  m_impl->do_work(message);
}
```

LIBEXAMPLE API IMPLEMENTATION

example_api_impl.hpp

```cpp
class example_api_impl
{
public:
  example_api_impl();
  ~example_api_impl();

  // TODO: Implement public API here
  void do_work(std::string const &message);

  // disable move and copy
  example_api_impl(example_api_impl &) = delete;
  auto operator=(example_api_impl const &)
    -> example_api_impl & = delete;
  example_api_impl(example_api_impl &&) = delete;
  auto operator=(example_api_impl &&)
    -> example_api_impl & = delete;

private:
  library_instance_lock m_instanceLock{};
  scoped_singleton<logger> m_log;
  queue m_workQueue{};
  worker m_worker;
};
```

example_api_impl.cpp

```cpp
example_api_impl::example_api_impl()
  : m_log{ "static_log" }
  , m_worker{ m_workQueue.get_consumer() }
{
  fmt::println("example_api_impl::ctor()");
}

example_api_impl::~example_api_impl()
{
  fmt::println("example_api_impl::~dtor()");
}

void example_api_impl::do_work(
  std::string const &message)
{
  // log shortcut, we will see how shortly
  log::info(
    fmt::format(R"(example_api_impl::do_work("{}"))",
    message));
  m_workQueue.get_producer().send(message);
}
```

SINGLE ACTIVE HANDLE

- Validate only a single active handle before any other initialisation
- If this fails, the `example_api_impl` ctor will fail and no other members (`m_log`, `m_workQueue`, `m_worker`) will be created.
- `constinit` (C++20) ensures our static storage is constant/zero initialised
  - compiler will error if it has to resort to dynamic initialisation

library_instance_lock.hpp

```
struct library_instance_lock
{
  constinit inline static std::atomic_bool s_instanceAlive{};

  library_instance_lock()
  {
    bool expected{ false };
    if (not s_instanceAlive.compare_exchange_strong(expected, true)) {
      throw library_instance_already_exists();
    }
  }

  ~library_instance_lock()
  {
    s_instanceAlive.store(false);
  }

  // ... disable move/copy ...
};
```

SEPARATE LIFETIME FROM ACCESS

- `scoped_singleton<T>`
- Separates the control of lifetime from access
  - Access is effectively a global pointer
  - Lifetime is managed by owning object - `example_api_impl`

scoped_singleton.hpp

```
template<typename T>
class scoped_singleton
{
public:
  template<typename... Args>
  explicit scoped_singleton(Args &&...args)
    : m_instance(std::forward<Args>(args)...)
  {
    if (s_instance != nullptr) {
      throw std::logic_error(
        "scoped_singleton<T> is already active");
    }
    s_instance = &m_instance;
  }

  ~scoped_singleton() { s_instance = nullptr; }

  // ... disable move/copy - static pointer would be broken

  static auto instance() -> T &
  {
    if (s_instance == nullptr) {
      throw std::logic_error("no live instance");
    }
    return *s_instance;
  }

private:
  T m_instance;
  constinit inline static T *s_instance{nullptr};
};
```

ACCESSING SCOPED_SINGLETON<LOGGER>

- The `scoped_singleton` provides a reference to the active `instance()`
- A bit much to type everywhere, so wrap it in some helper functions

```cpp
class logger
{
public:
  explicit logger(std::string name);
  ~logger();

  void error(std::string const &message);
  void warn(std::string const &message);
  void info(std::string const &message);

private:
  std::string m_name;
};
```

```cpp
namespace log {
  inline static void error(std::string const &message)
  {
    scoped_singleton<logger>::instance().error(message);
  }

  inline static void warn(std::string const &message)
  {
    scoped_singleton<logger>::instance().warn(message);
  }

  inline static void info(std::string const &message)
  {
    scoped_singleton<logger>::instance().info(message);
  }
}  // namespace log
```

ONE TIME INITIALIZATION

- What if `libExample` depended on 3rd party libraries?
  - `libcurl` or `openssl` require one time initialisation and cleanup
- This is independent on the lifetime of our `example_api` RAII object
- Use a *Nifty Counter* to initialise automatically

example_api.hpp

```cpp
static struct library_global_initializer
{
  library_global_initializer() noexcept;
  ~library_global_initializer() noexcept;

  // ... disable move/copy
  // - avoid corrupting counter ...
} const LibraryGlobalInitializer;
```

library_global_initializer.cpp

```cpp
namespace {
  void global_initialize() noexcept
  {
    // TODO: Call all dependencies start up routines here
    // Example:
    //   libcurl - curl_global_init()
    fmt::println("Initializing lib::example dependencies");
  }

  void global_finalize() noexcept
  {
    // TODO: Call all dependencies clean up routines here
    // Example:
    //   libcurl - curl_global_cleanup()
    fmt::println("Finalizing lib::example dependencies");
  }

  int initializationCounter;
}  // namespace

library_global_initializer::library_global_initializer() noexcept
{
  // if this is the first call, we run the initialization
  if (initializationCounter++ == 0) { global_initialize(); }
}

library_global_initializer::~library_global_initializer() noexcept
{
  // if this is the last call, we run the finalization
  if (--initializationCounter == 0) { global_finalize(); }
}
```

CLIENT CODE

client.cpp

```cpp
int main() {
example_api api{}; // api lifetime begins

api.do_work("task 1");
api.do_work("task 2");

} // api lifetime ends
```

```cpp
int main() {
example_api api{}; // api lifetime begins

api.do_work("task 1");
api.do_work("task 2");

// ...

example_api another{};  // Exception
another.do_work("nope");

} // api lifetime ends
```

ADDITIONAL NICETIES

The example code project also:

- Verifies that the client code links against the same version of the code the header is from
- CMake project with presets for
  - `clang` and `gcc`
  - *static* vs *shared* library
  - Ensures symbols are hidden by default in the library

THANK YOU, QUESTIONS?

GitHub https://github.com/AshleyRoll

Discord *#include*<C++> @AshleyRoll

Slack *Cpplang* @AshleyRoll