From Theory to Practice

A Student Journey Building a C++ Game Engine in Open-Source

CHU YI HERR

About Me

- Major Computer Science
- Pursuing Bachelors of Science at San Francisco State University

Takeaways

- Bringing up ways we solved memory-safety issues using modern C++.
- Discovering how conan simplified package management
- Talking about TheAtlasEngine and designs I approached in for managing lifetimes.
- Walking away with knowledge in how to better leverage modern C++.

What to expect.

S.1. Journey to TheAtlasEngine

S.2. Managing Dependencies using Conan

S.3. Object Model in TheAtlasEngine

S.4. Tracking Game State

S.5 Modern C++ Features in TheAtlasEngine

Takeaways

- Building a robust memory-safe API with modern C++
- Conan can simplify managing dependencies.
- Learning how we approach in ensuring memory safety with object lifetimes.

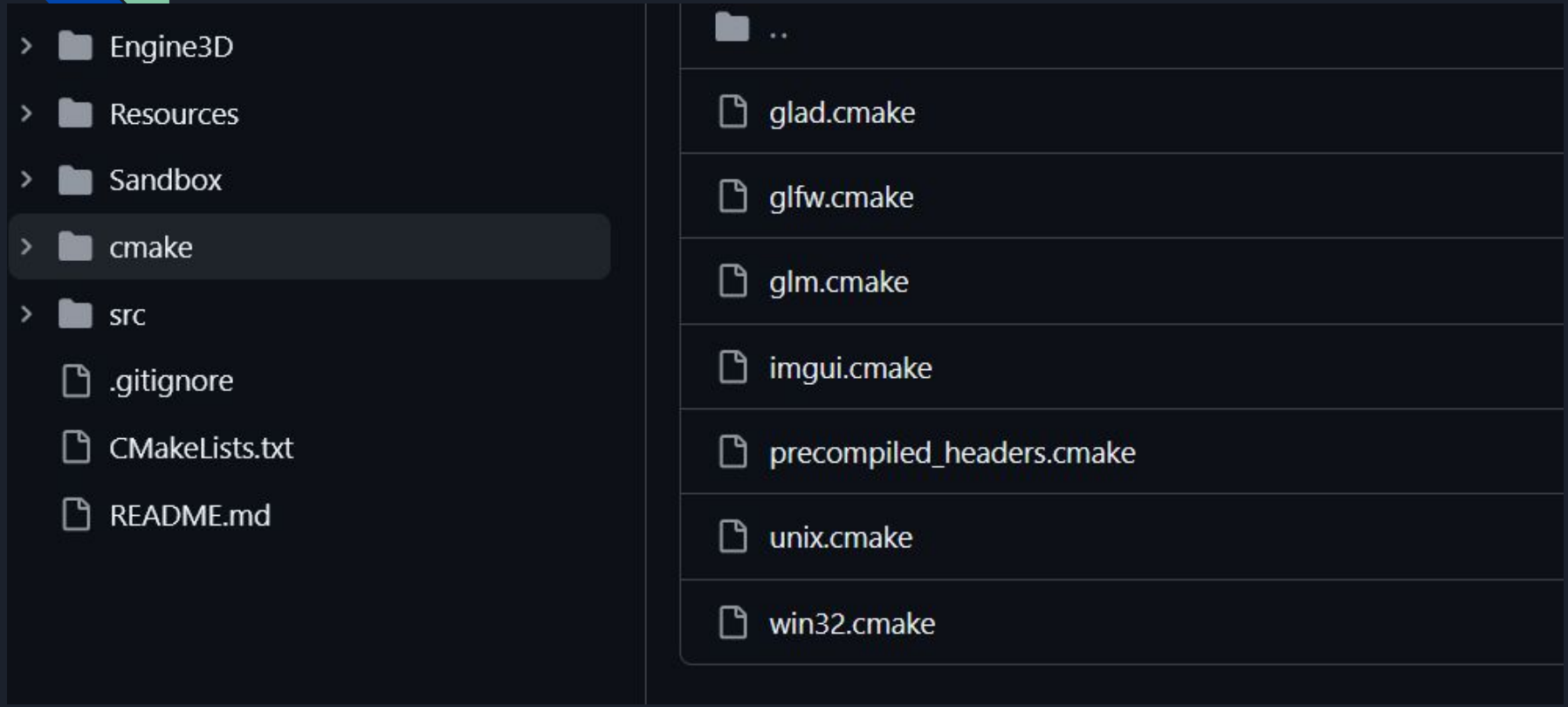
Engine3D before TheAtlasEngine

- Engine3D before TheAtlasEngine
- CMake having difficult learning curve.
- Modern C++ Shift to TheAtlasEngine

Engine3D Before TheAtlasEngine

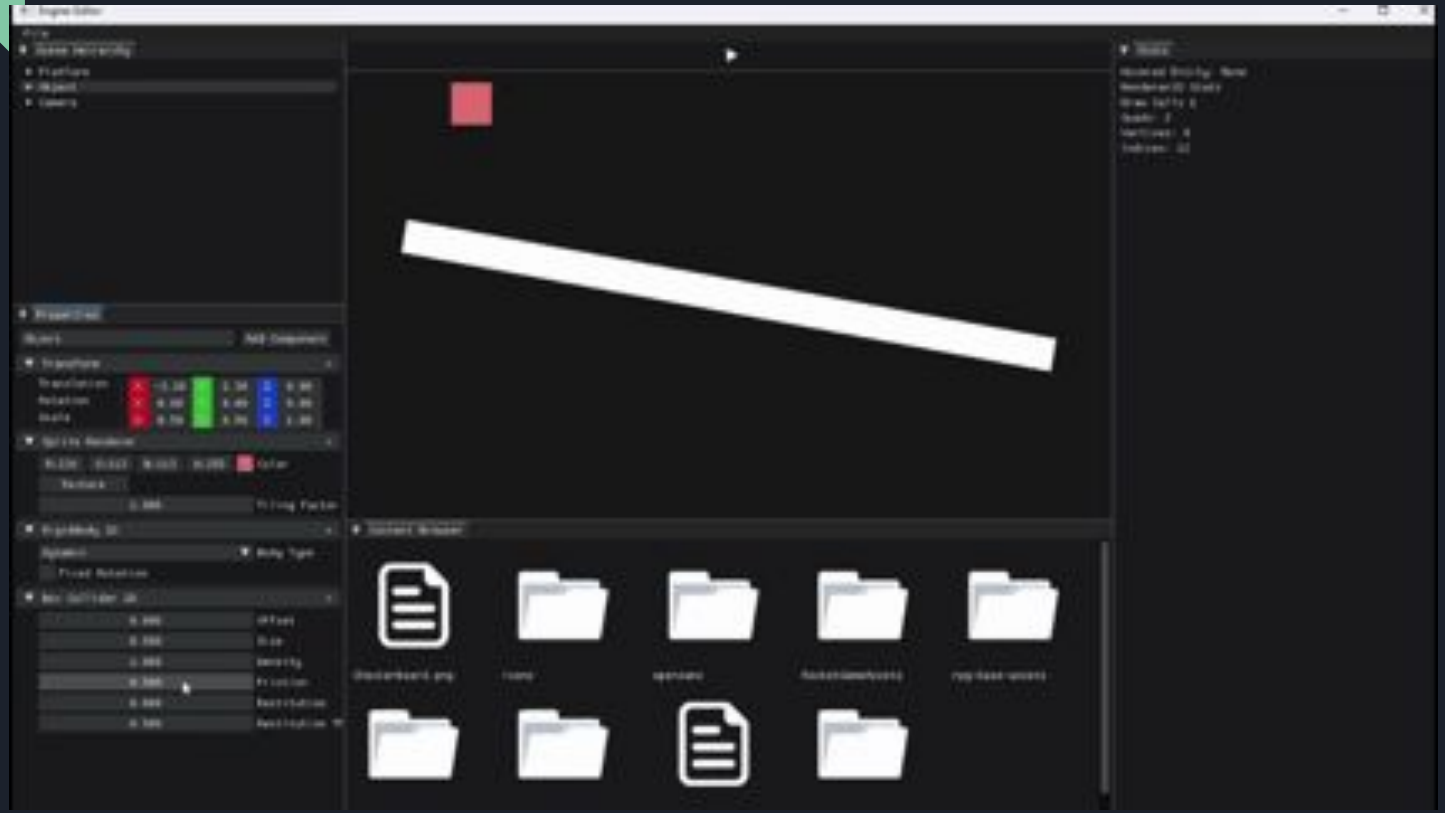
Engine3D Before TheAtlasEngine

CMake has difficult learning curve



```
option( GLFW-CMAKE-STARTER-USE-GLFW-GLAD "Use GLAD from GLFW" ON )

if( GLFW-CMAKE-STARTER-USE-GLFW-GLAD )
    include_directories("${GLFW_SOURCE_DIR}/deps")
    set( GLAD_GL "${GLFW_SOURCE_DIR}/deps/glad/gl.h" )
endif()

if( MSVC )
    SET( CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} /ENTRY:mainCRTStartup" )
endif()

add_subdirectory(external/glm)
add_subdirectory(external/yaml-cpp)

add_subdirectory(external/Box2D-cmake)

include(FetchContent)
add_subdirectory(external/fmt)


add_subdirectory(external/spdlog)

add_subdirectory(external/imgui)

add_subdirectory(external/glad)

target_link_libraries(${PROJECT_NAME}
    ${OPENGL_LIBRARIES}
    glfw
    ${Vulkan_LIBRARIES}
    fmt::fmt
    spdlog::spdlog
    glm::glm
    tobanteGaming::Box2D
    yaml-cpp::yaml-cpp
    imgui
    glad
)
```

The Engine3D Project



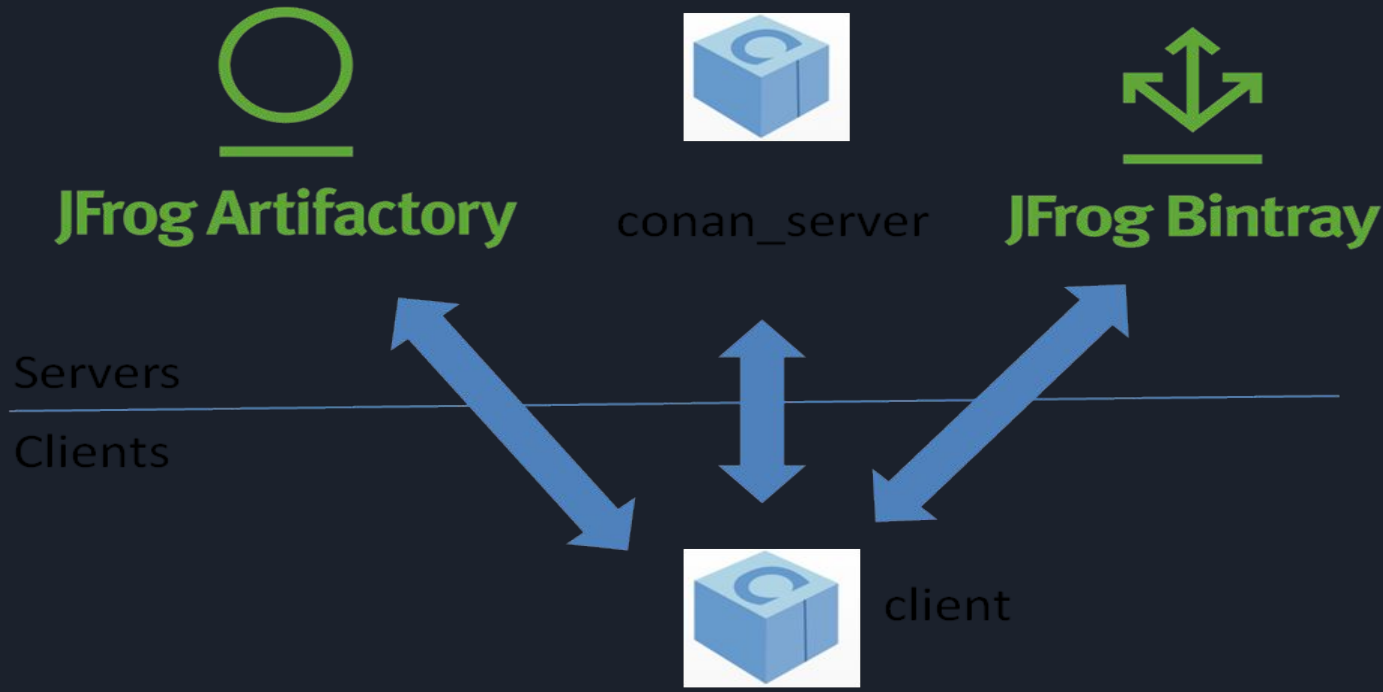
Modern Shift to TheAtlasEngine

Modern C++ shift to TheAtlasEngine

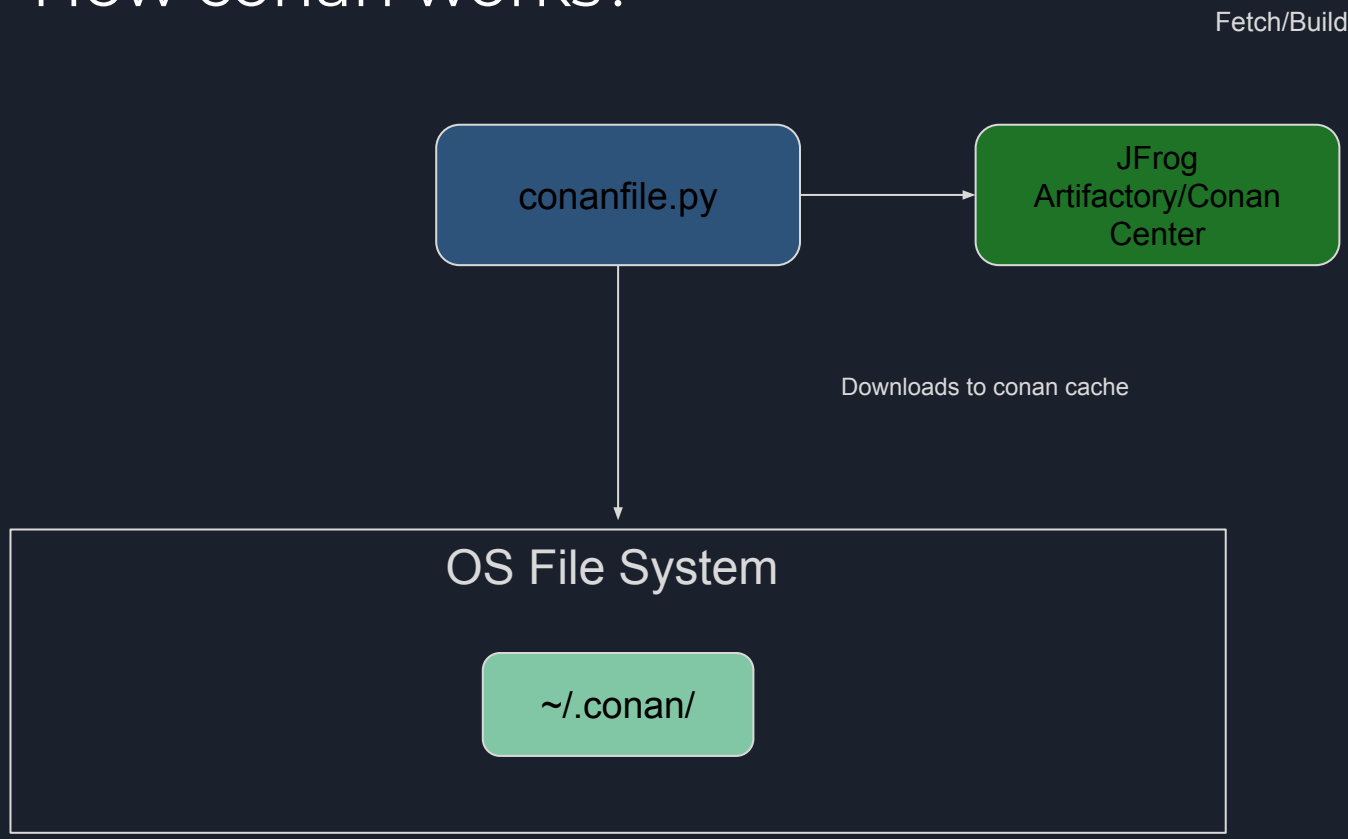
Managing dependencies using Conan

What is conan?

What is conan?



How conan works?



Including dependencies into your project

```
# Putting all of your build-related dependencies here
def build_requirements(self):
    self.tool_requires("make/4.4.1")
    self.tool_requires("cmake/3.27.1")
    self.tool_requires("engine3d-cmake-utils/4.0")

# Putting all of your packages here
def requirements(self):
    self.requires("flecs/4.0.4")
    self.requires("tinyobjloader/2.0.0-rc10")
    self.requires("spdlog/1.15.1")
    self.requires("stb/cci.20230920")
    self.requires("imguidocking/2.0")
    self.requires("joltphysics/5.2.0")
    self.requires("atlas/0.2")
```

Including dependencies into your project

```
add_executable(
    ${PROJECT_NAME}
    application.cpp
    game_world.cpp
    main_scene.cpp
)

find_package(spdlog REQUIRED)
find_package(flecs REQUIRED)
find_package(Jolt REQUIRED)
find_package(atlas REQUIRED)

target_link_libraries(
    ${PROJECT_NAME} PUBLIC
    spdlog::spdlog
    flecs::flecs_static
    Jolt::Jolt
    atlas::atlas
)
```

Cmake utilities for ease of handling dependencies (1)

```
cmake_minimum_required(VERSION 3.27)
project(game-template CXX)

build_application(
    SOURCES
    Application.cpp
    game_world.cpp
    main_scene.cpp

    PACKAGES
    spdlog
    flecs
    Jolt
    atlas

    LINK_PACKAGES
    spdlog::spdlog
    flecs::flecs_static
    Jolt::Jolt
    atlas::atlas
)

target_include_directories(${PROJECT_NAME} PUBLIC ${CMAKE_CURRENT_LIST_DIR})

generate_compile_commands()
```

Cmake utilities for ease of handling dependencies (2)

```
cmake_minimum_required(VERSION 3.27)
project(game-template CXX)

build_application(
    SOURCES
    Application.cpp
    game_world.cpp
    main_scene.cpp

    PACKAGES
    spdlog
    flecs
    Jolt
    atlas

    LINK_PACKAGES
    spdlog::spdlog
    flecs::flecs_static
    Jolt::Jolt
    atlas::atlas
)

target_include_directories(${PROJECT_NAME} PUBLIC ${CMAKE_CURRENT_LIST_DIR})

generate_compile_commands()
```

CMake Utilities in TheAtlasEngine

CMake in TheAtlasEngine project

```
build_core_library(
    DIRECTORIES src

    ENABLE_TESTS ${ENABLE_TESTS_ONLY}

    UNIT_TEST_SOURCES

    tests/main.test.cpp
    tests/basic_add.test.cpp
    tests/entity_component_system.test.cpp
    tests/math.test.cpp
    tests/scene.test.cpp
    tests/jolt_type_conversion.test.cpp
    tests/jolt_engine.test.cpp
    # tests/object.test.cpp

    PACKAGES
    tinyobjloader
    TinyGLTF
    # shaderc # uncomment to use shaderc
    ${SHADERC_PACKAGE}
    watcher
    vulkan-cpp

    LINK_PACKAGES
    tinyobjloader::tinyobjloader
    TinyGLTF::TinyGLTF
    # shaderc::shaderc # uncomment to use shaderc
    ${SHADERC_LINK_PACKAGE}
    watcher::watcher
    vulkan-cpp::vulkan-cpp
)
```

CMake in TheAtlasEngine project

```
build_core_library(
    DIRECTORIES src

    ENABLE_TESTS ${ENABLE_TESTS_ONLY}

    UNIT_TEST_SOURCES

    tests/main.test.cpp
    tests/basic_add.test.cpp
    tests/entity_component_system.test.cpp
    tests/math.test.cpp
    tests/scene.test.cpp
    tests/jolt_type_conversion.test.cpp
    tests/jolt_engine.test.cpp
    # tests/object.test.cpp

    PACKAGES
    tinyobjloader
    TinyGLTF
    # shaderc # uncomment to use shaderc
    ${SHADERC_PACKAGE}
    watcher
    vulkan-cpp

    LINK_PACKAGES
    tinyobjloader::tinyobjloader
    TinyGLTF::TinyGLTF
    # shaderc::shaderc # uncomment to use shaderc
    ${SHADERC_LINK_PACKAGE}
    watcher::watcher
    vulkan-cpp::vulkan-cpp
)
```

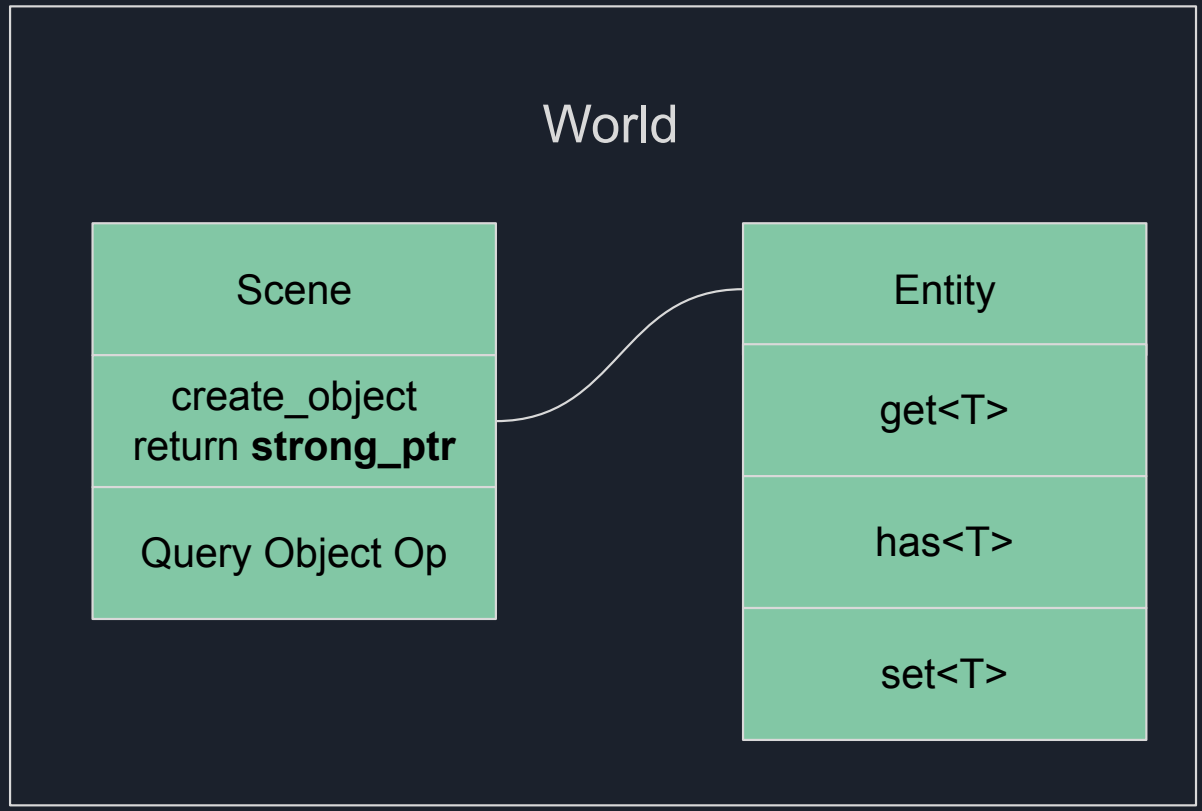
Conanfile in TheAtlasEngine

```
def build_requirements(self):
    self.tool_requires("make/4.4.1")
    self.tool_requires("cmake/3.27.1")
    self.tool_requires("engine3d-cmake-utils/4.0")

def requirements(self):
    self.requires("joltphysics/5.2.0")
    if self.options.enable_shaderc:
        self.requires("shaderc/2024.1")
    self.requires("imguidocking/2.0")
    self.requires("flecs/4.0.4")
    self.requires("glfw/3.4", transitive_headers=True)
    self.requires("opengl/system", transitive_headers=True)
    self.requires("spdlog/1.15.1")
    self.requires("glm/1.0.1", transitive_headers=True)
    self.requires("yaml-cpp/0.8.0", transitive_headers=True)

    # Vulkan-related headers and includes packages
    self.requires("vulkan-headers/1.3.290.0", transitive_headers=True)
    self.requires("tinyobjloader/2.0.0-rc10")
    self.requires("tinygltf/2.9.0")
    self.requires("stb/cci.20230920")

    self.requires("nfd/1.0")
    self.requires("watcher/0.12.0")
    self.requires("boost-ext-ut/2.1.0")
    self.requires("vulkan-cpp/1.0")
```

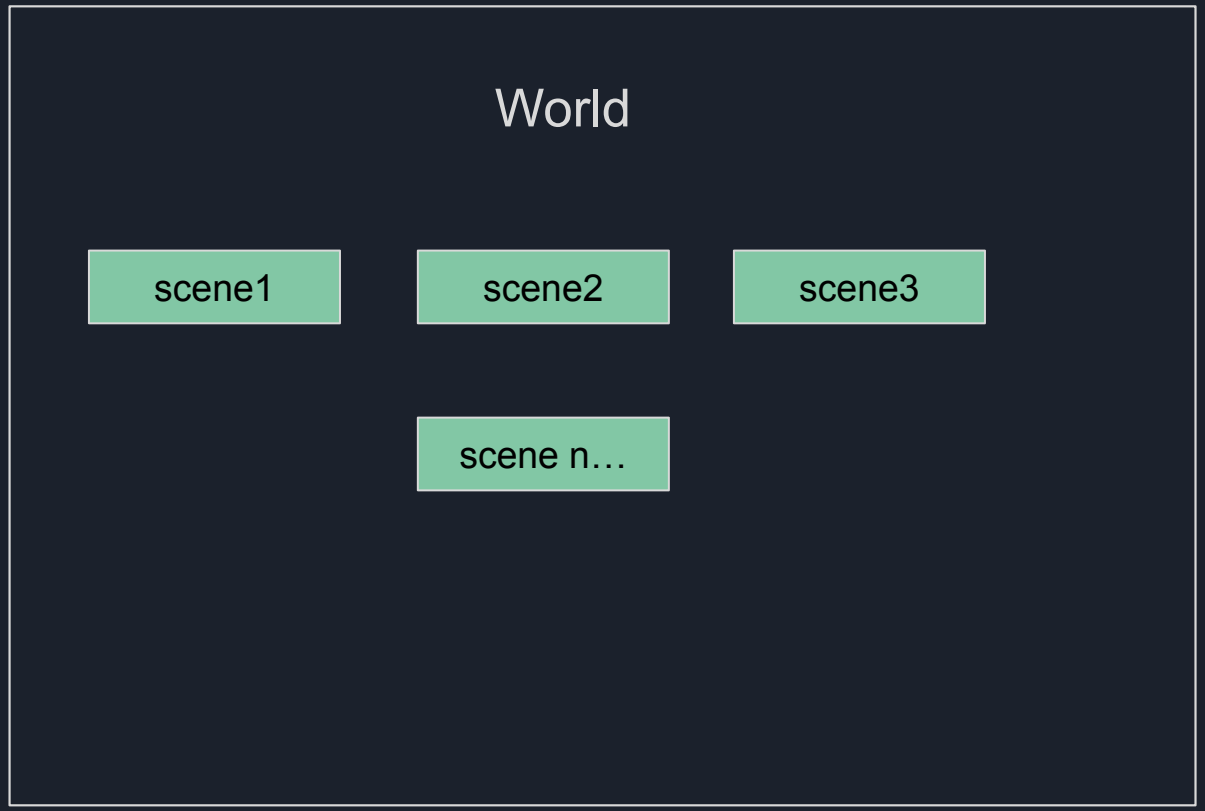
Modern C++ in TheAtlasEngine

Objects in TheAtlasEngine

Object Model in TheAtlasEngine



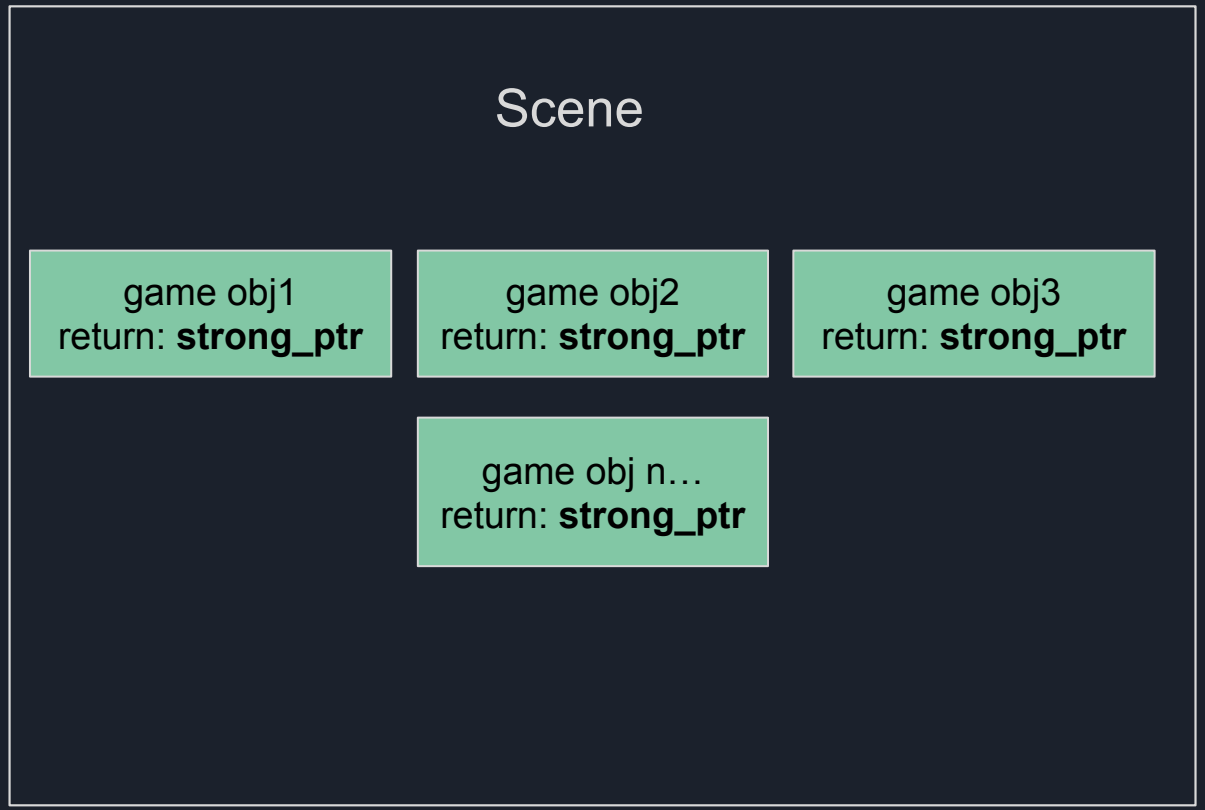
World Scope in TheAtlasEngine



Object Model in TheAtlasEngine

```
class world {
public:
    world() = default;
    world(const std::string& p_name);

    void add_scene(const ref<scene_scope>& p_scene);

    ref<scene_scope> get_scene(const std::string& p_name);

};
```

Scene Scope in TheAtlasEngine

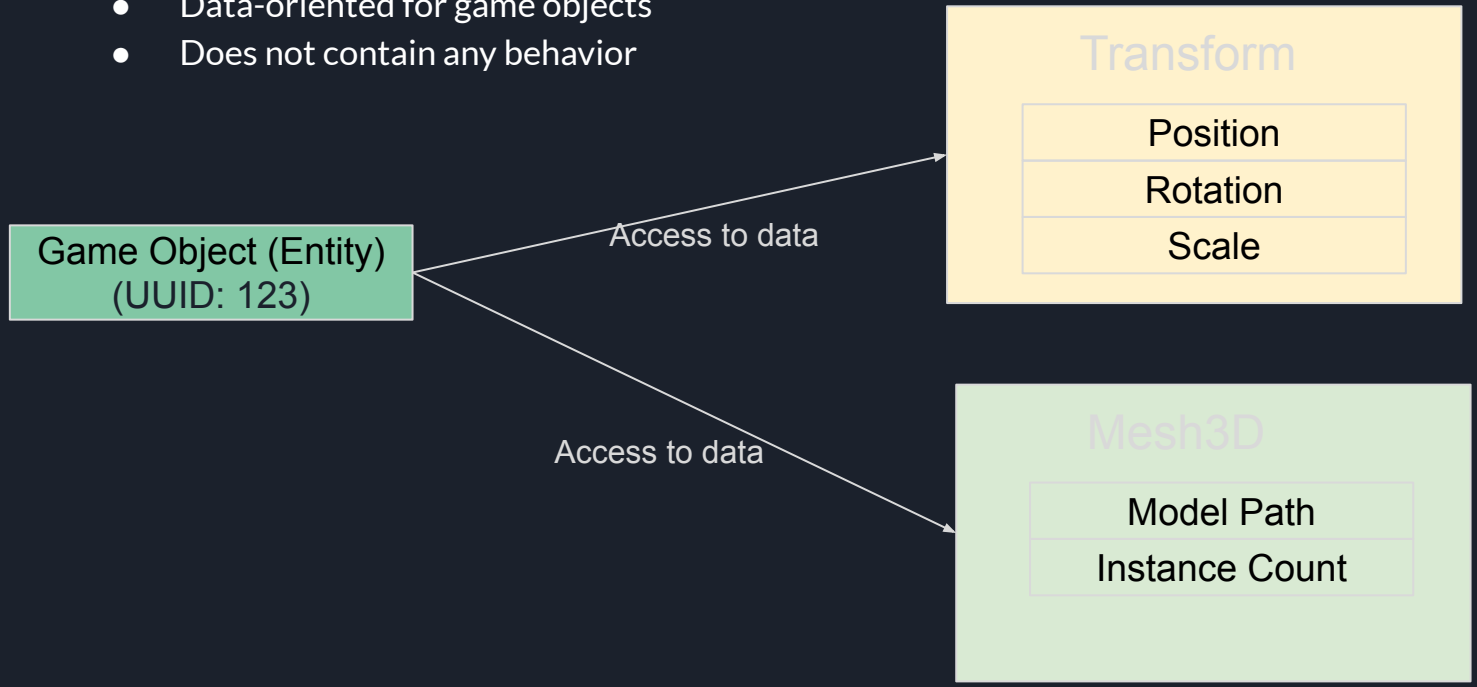


Object Model in TheAtlasEngine

```
class scene {
public:
    scene(const std::string& p_name);

    strong_ref<scene_object> create_object(const std::string& p_name) {
        return create_strong_ref<scene_object>(p_allocator: m_allocator, p_name);
    }

private:
    std::pmr::polymorphic_allocator<> m_allocator;
};
```

Game Objects in TheAtlasEngine

- Data-oriented for game objects
- Does not contain any behavior



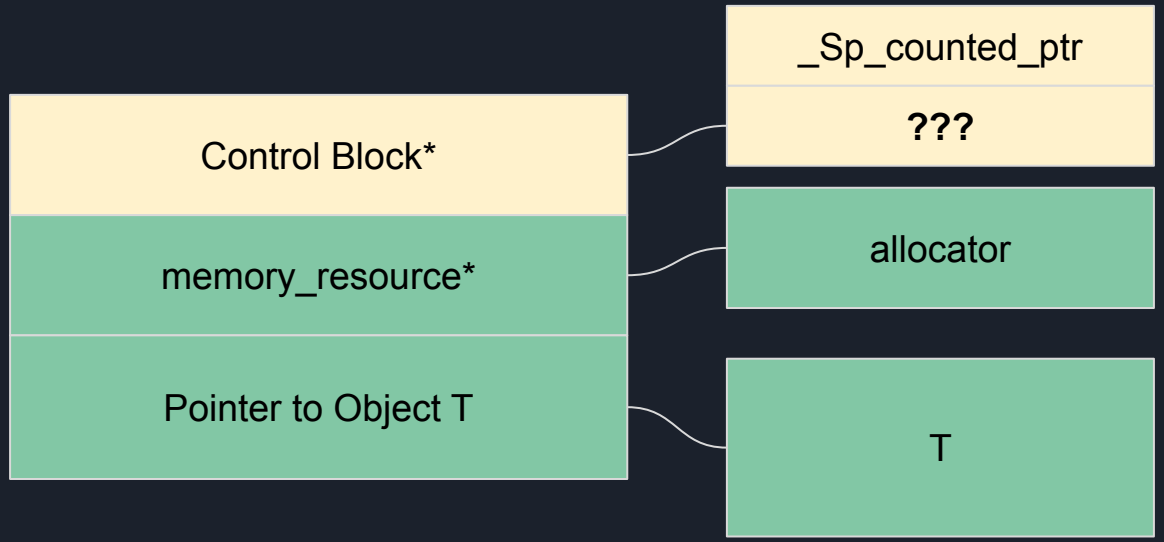
Design Principles for strong_ptr

Why shared_ptr isn’t enough?

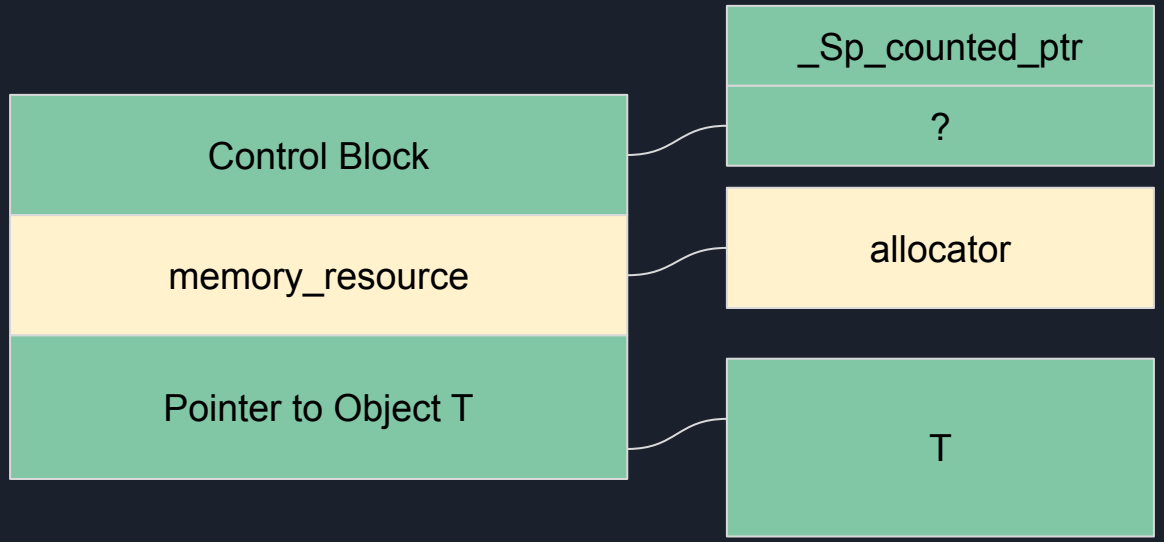
- Objects can be null.
- Inefficient in memory space

Addressing shared_ptr implementation.

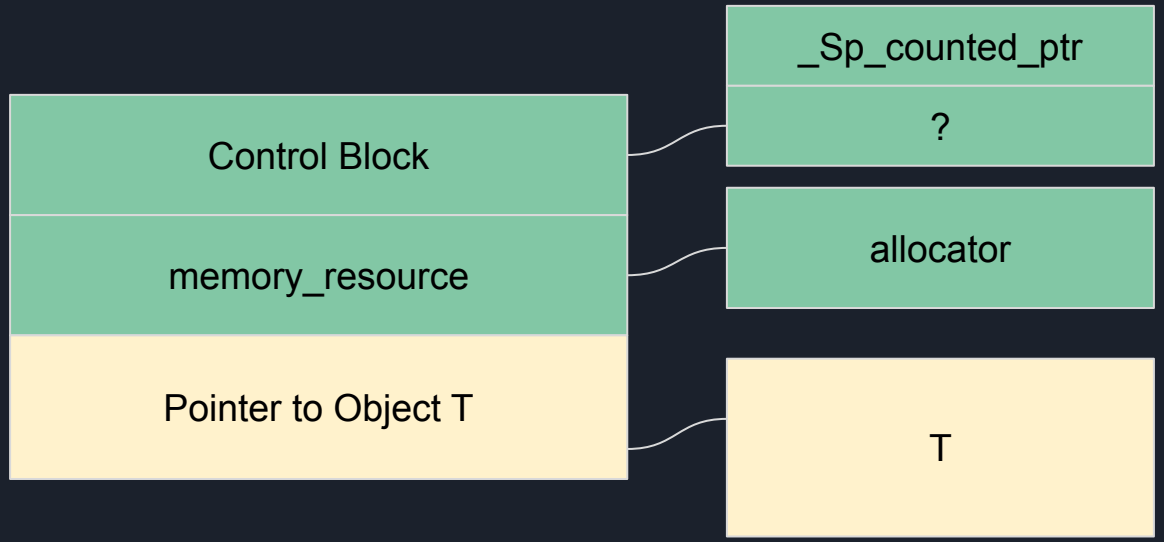
shared_ptr implementation (in libstdc++)?



shared_ptr implementation (in libstdc++)?

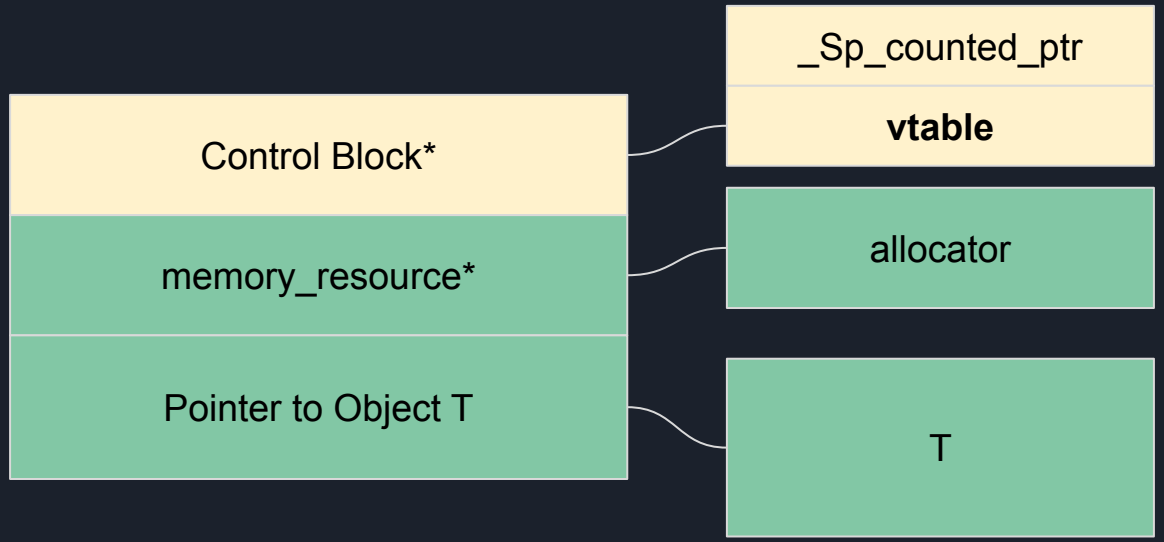


shared_ptr implementation (in libstdc++)?



Where is the inefficiency in shared_ptr?

Generating vtables!



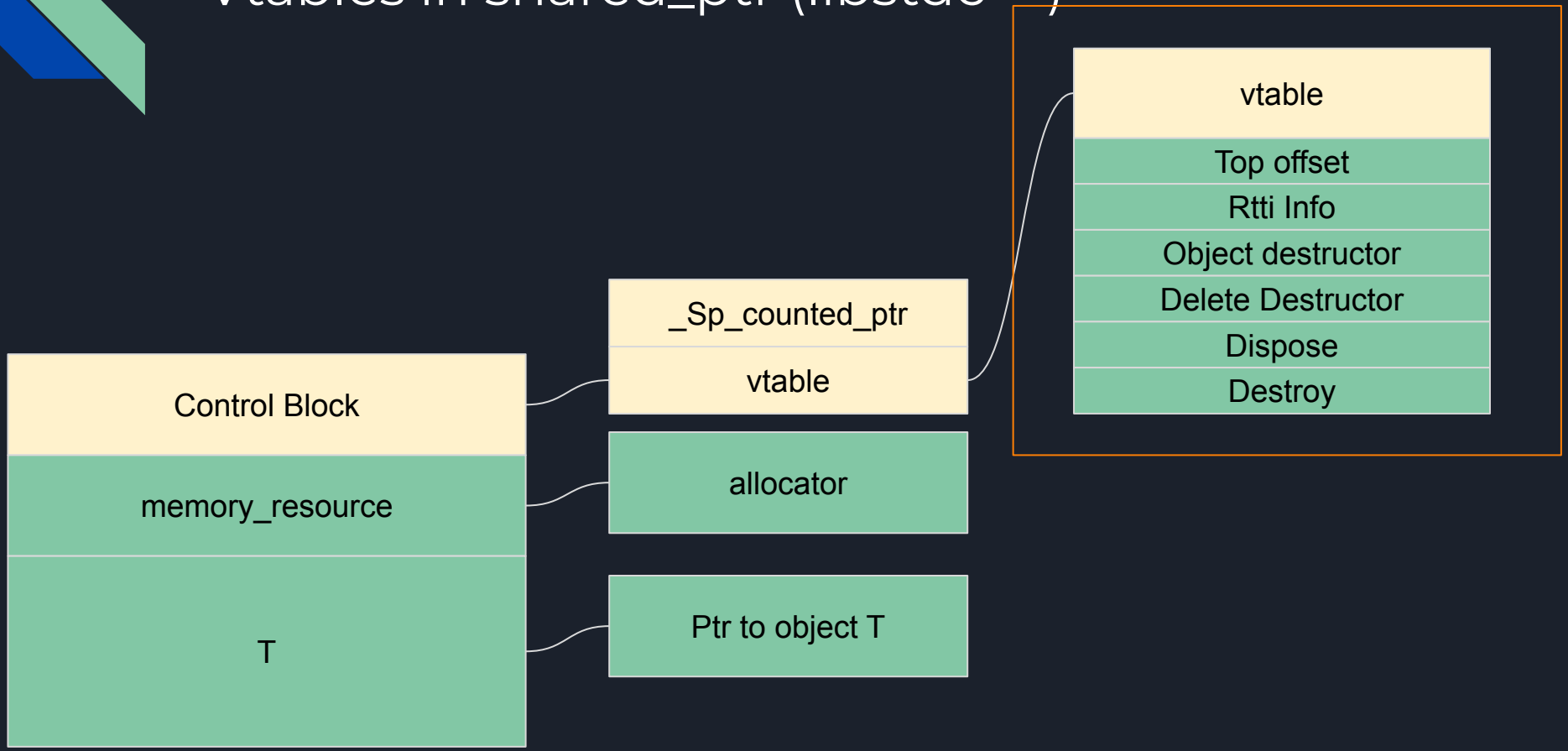
Generating vtables!

Refresher: How does VTables work?

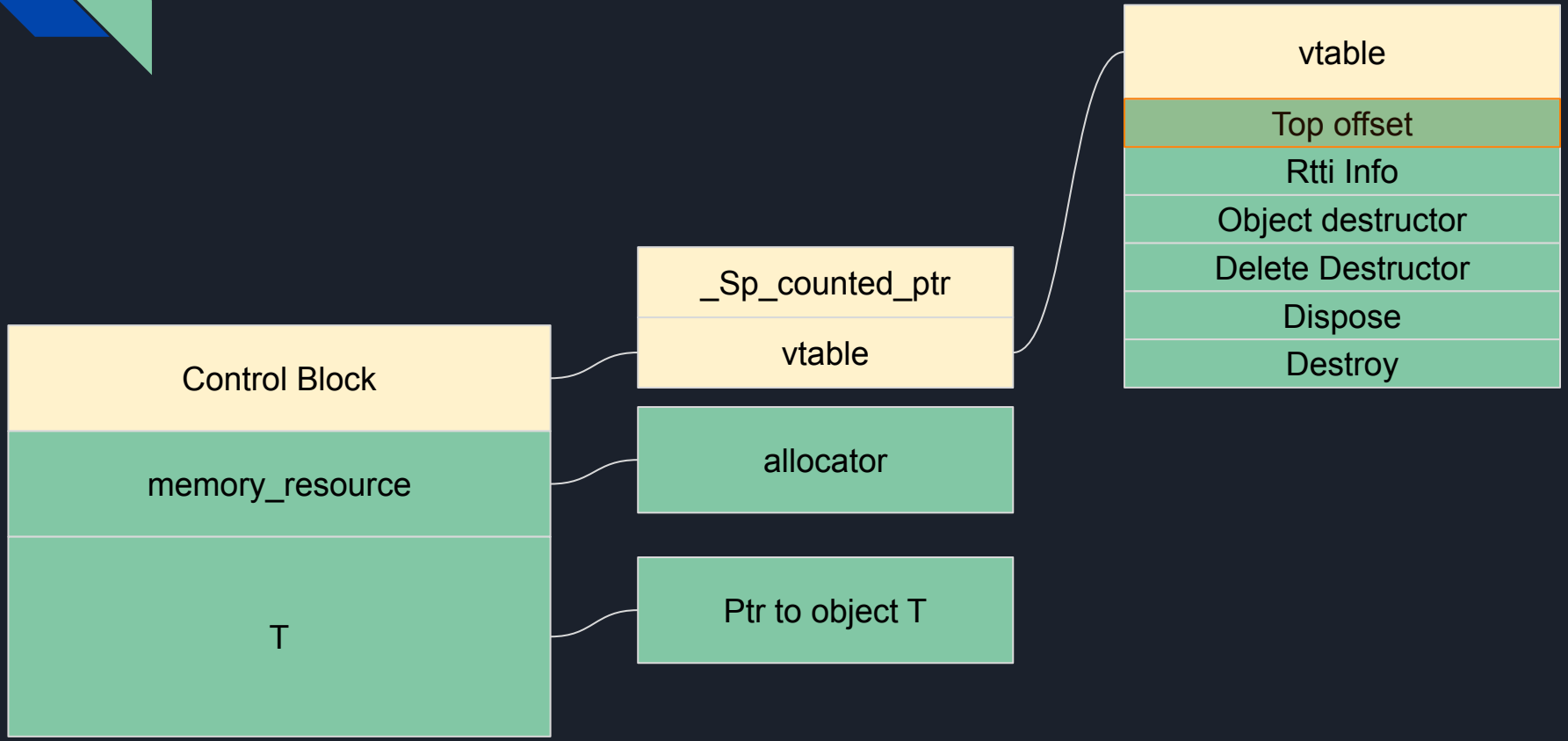
Refresher: How does VTables work?

Looking into the vtable layout

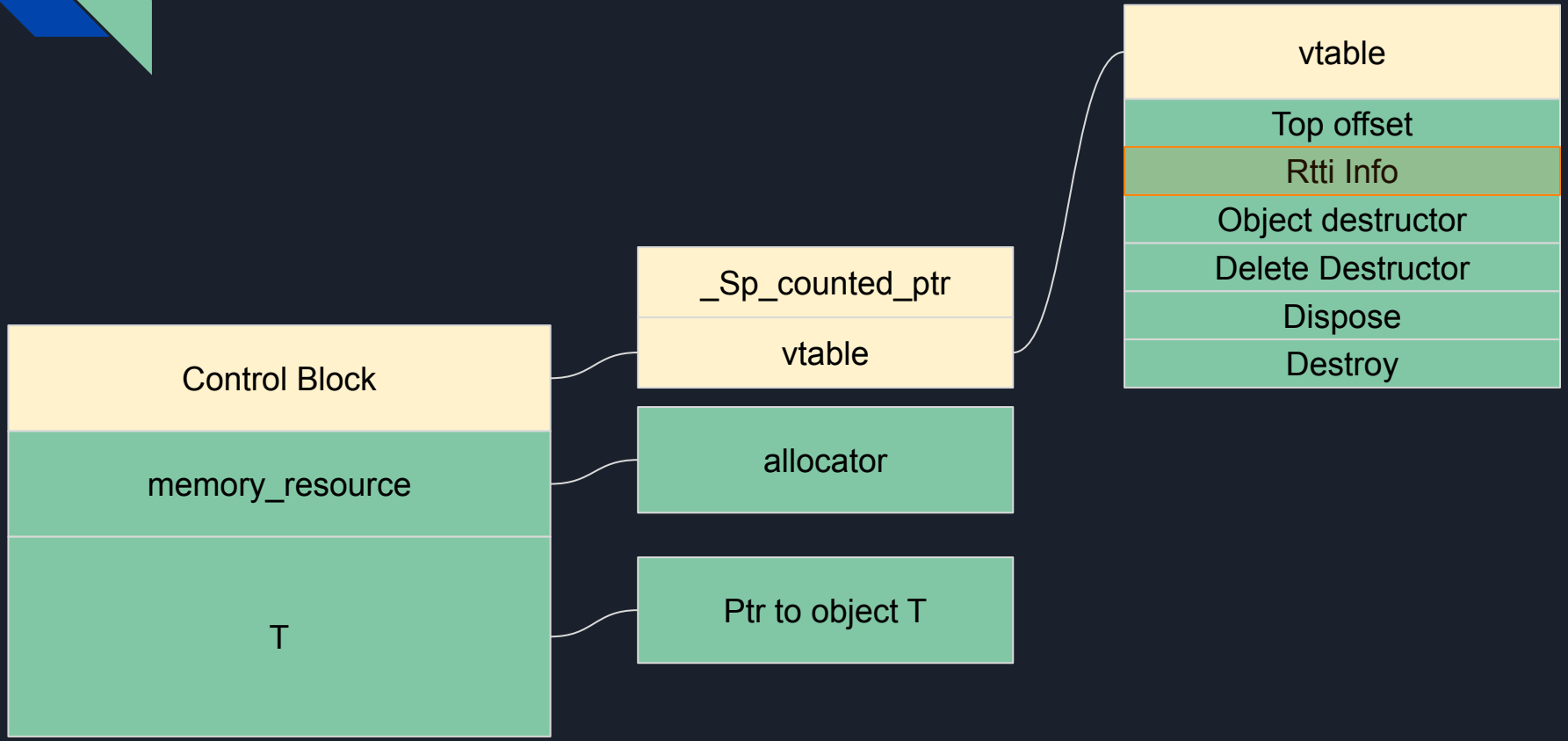
Vtables in shared_ptr (libstdc++)



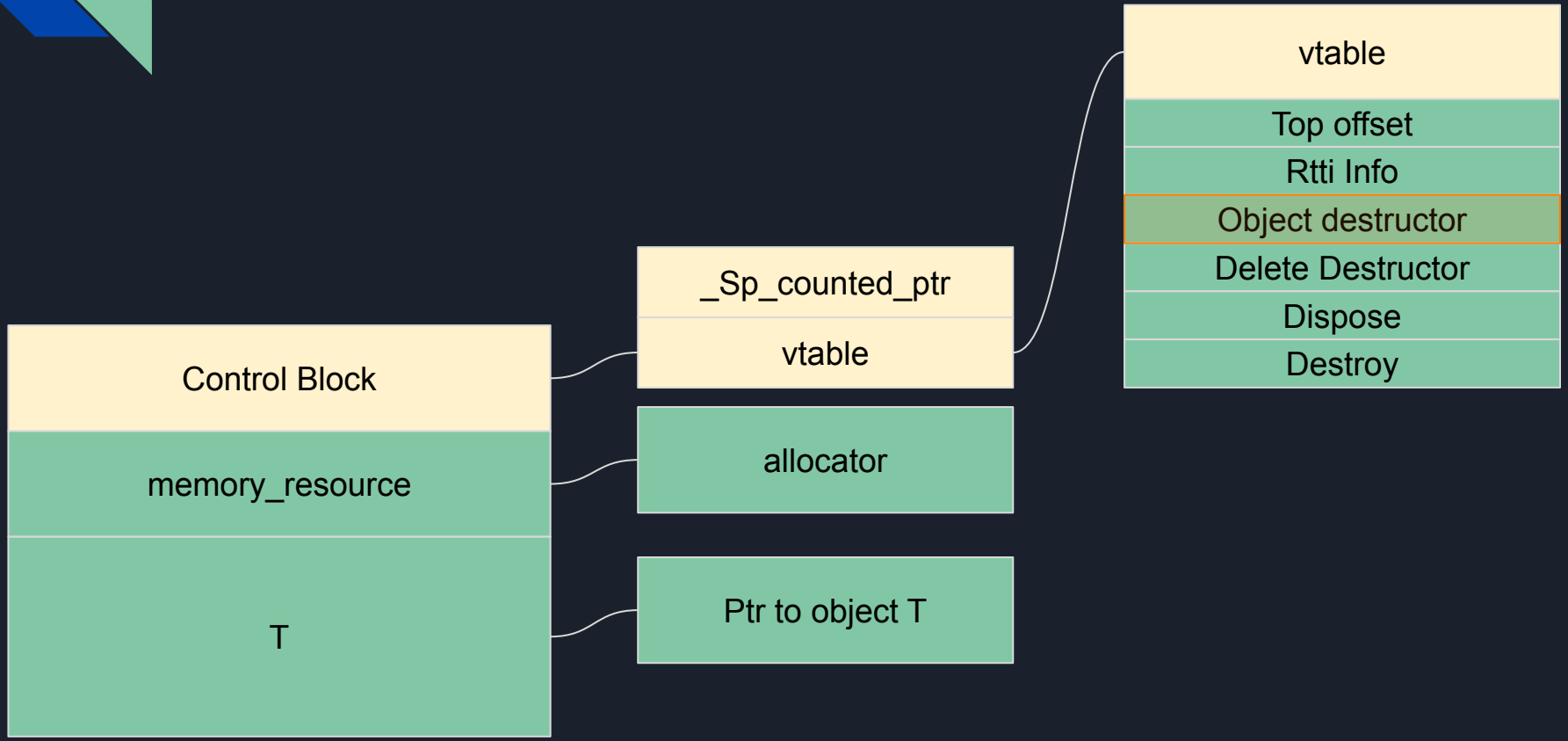
Vtables in shared_ptr (libstdc++)



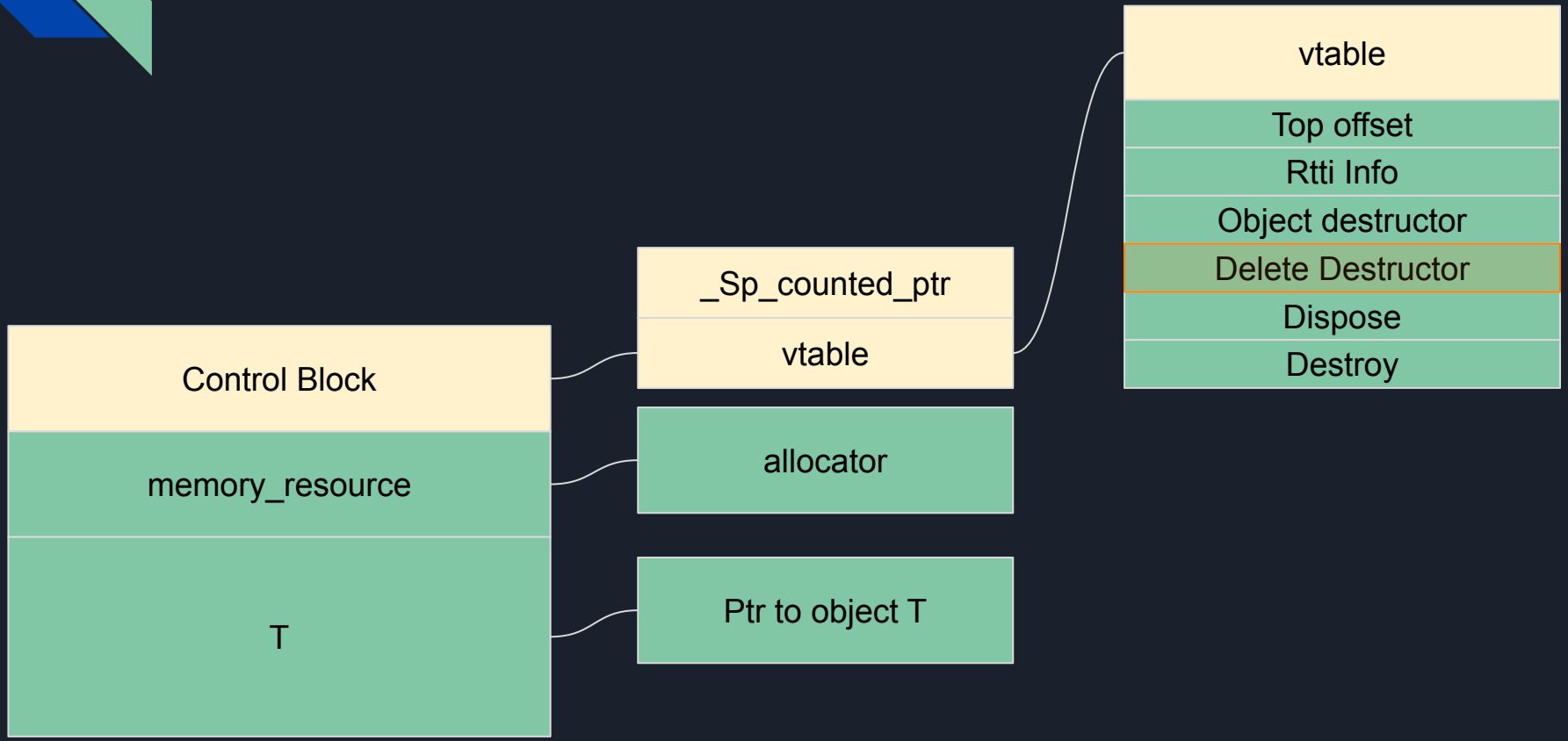
Vtables in shared_ptr (libstdc++)



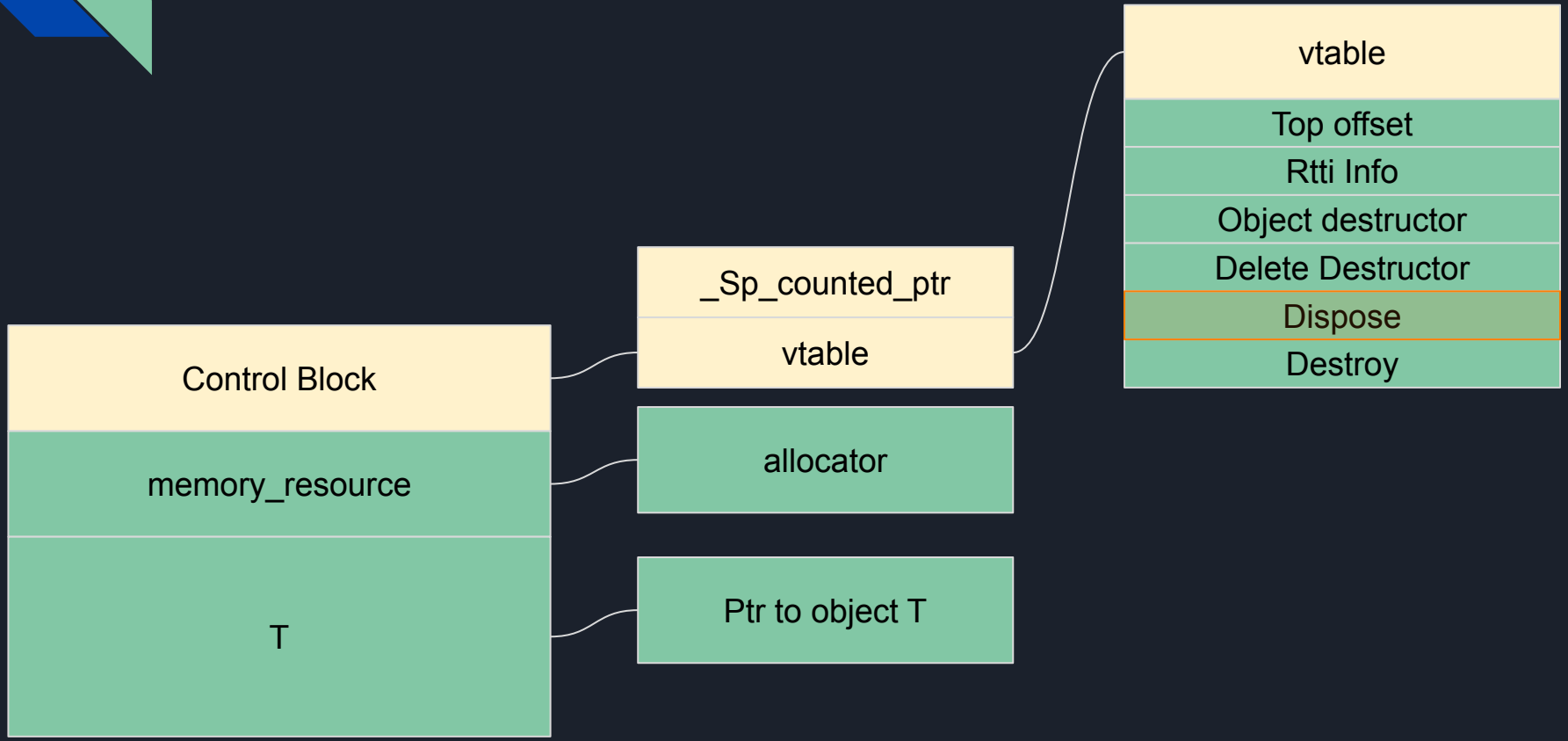
Vtables in shared_ptr (libstdc++)



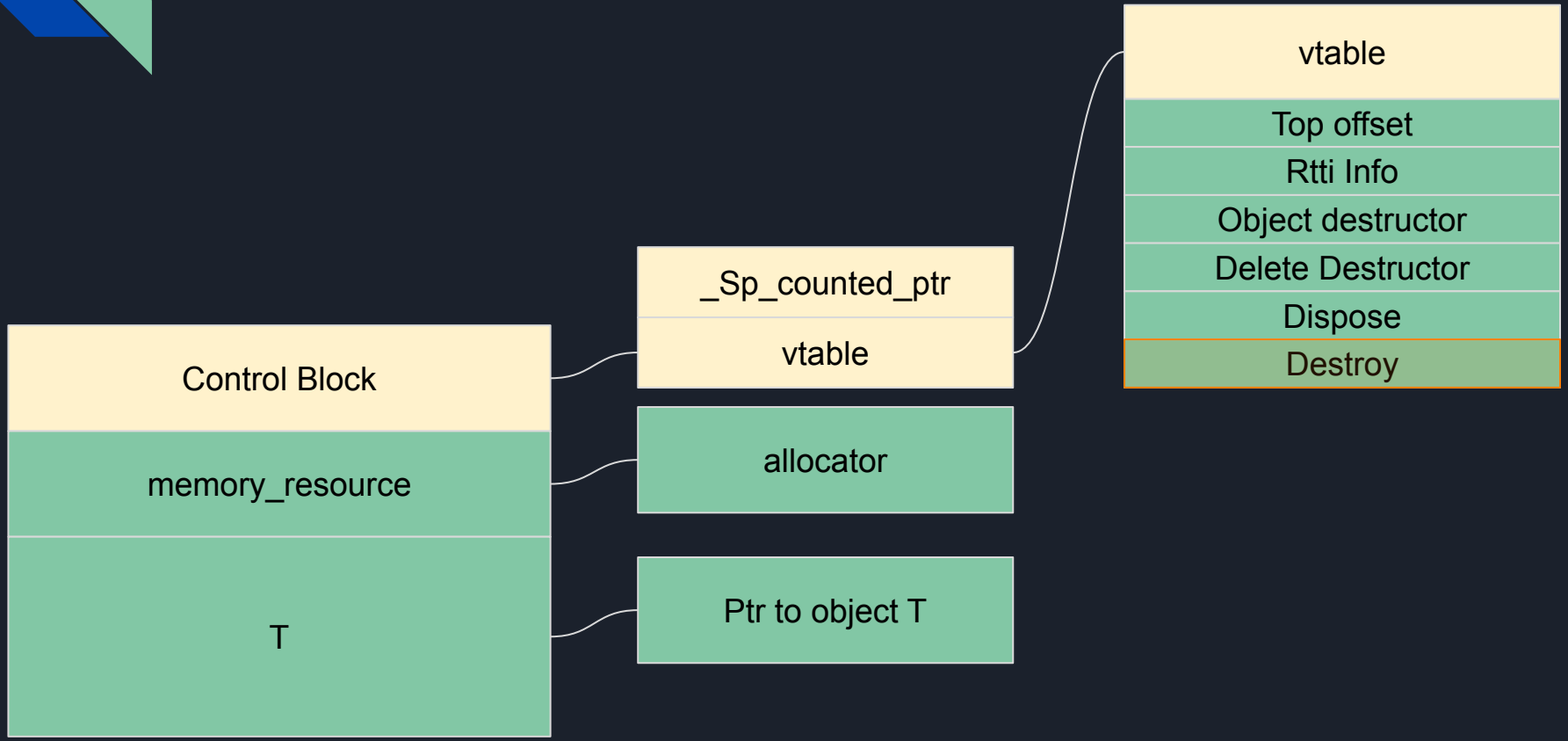
Vtables in shared_ptr (libstdc++)



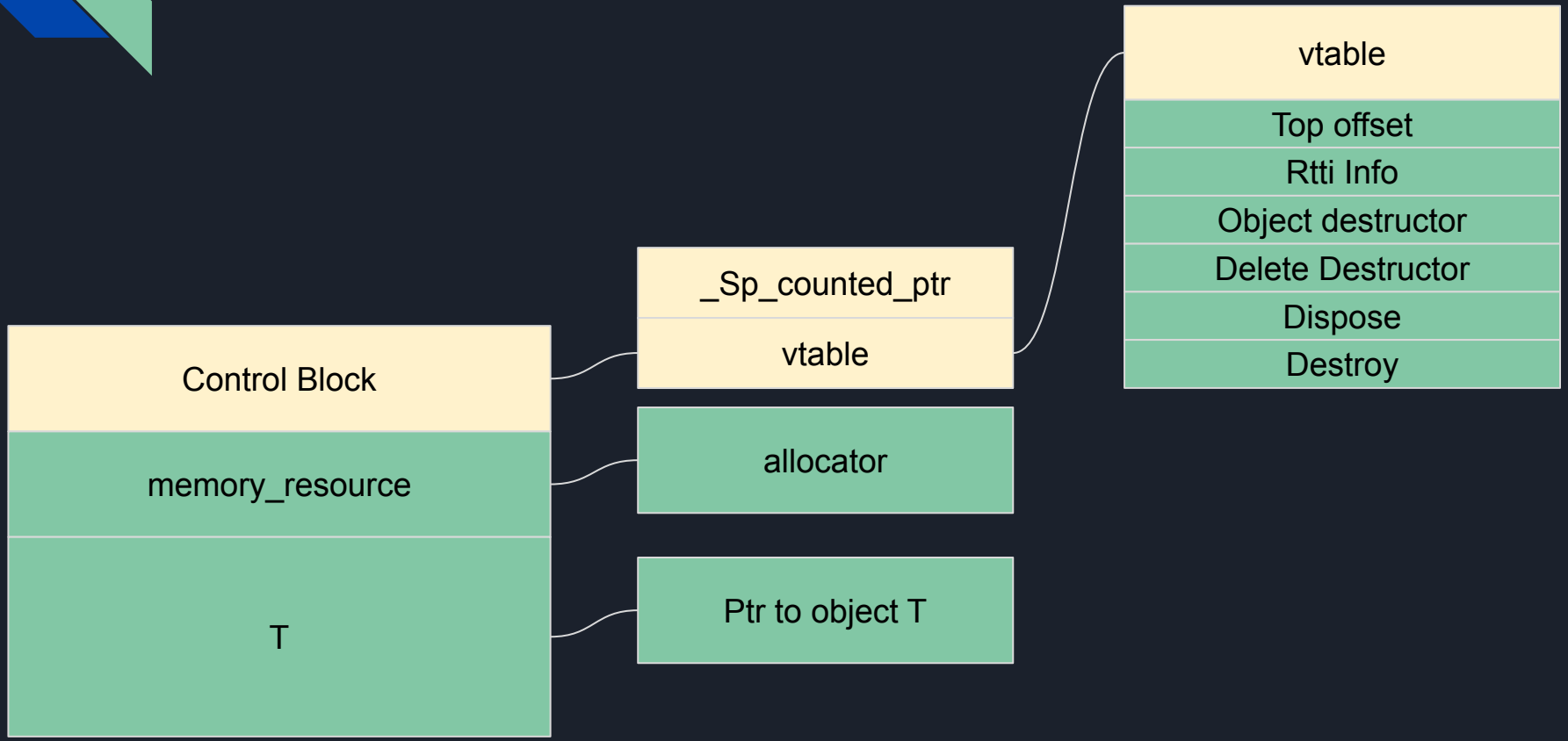
Vtables in shared_ptr (libstdc++)



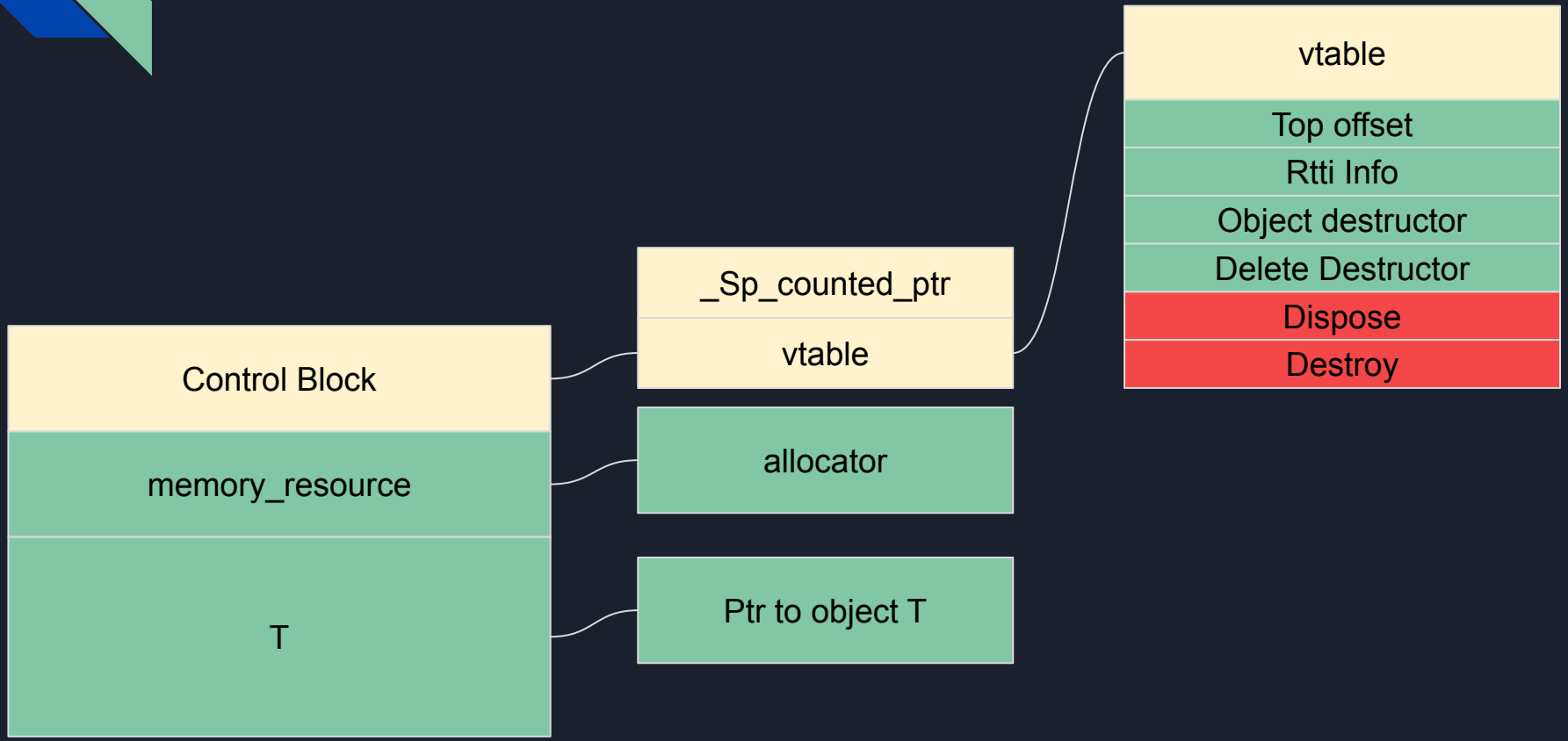
Vtables in shared_ptr (libstdc++)



Vtables in shared_ptr (libstdc++)



Vtables in shared_ptr (libstdc++)



Code Example

vtables in shared_ptr (libstdc++)

```cpp
class shape {
public:
    virtual ~shape() = default;
    virtual void write_data(const std::span<uint8_t>& p_data) = 0;
};


class square : public shape {
public:
    square() = default;

    ~square() override {}

    void write_data(const std::span<uint8_t>& p_data) override {
        // do some stuff
    }
};

int main() {
    std::shared_ptr<square> temp = std::make_shared<square>();
}
```

vtables in shared_ptr

```
vtable for std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>:
        .quad   0
        .quad   typeinfo for std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>
        .quad   std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace () [base object destructor]
        .quad   std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace () [deleting destructor]
        .quad   std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
        .quad   std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
        .quad   std::_Sp_counted_ptr_inplace <square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter (std::type_info const&)
```

Why did we not use shared_ptr

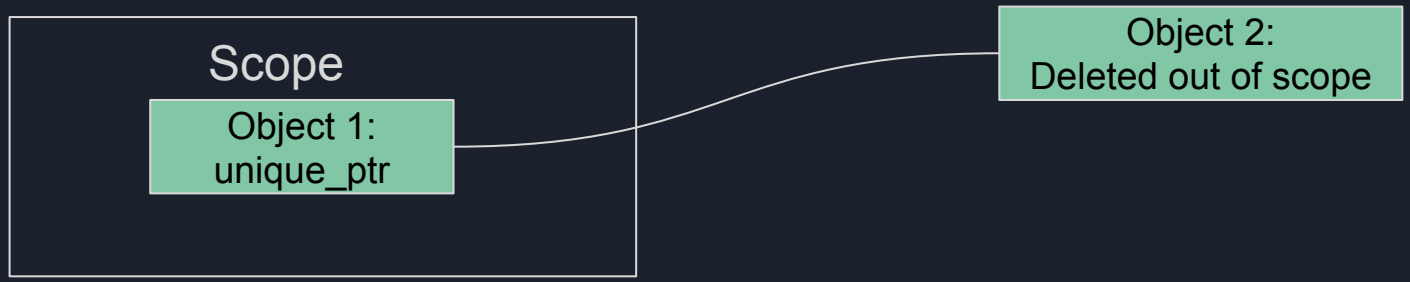
- Wanting something to not be nullable
- shared_ptr allows for a pointer to be null.
- Code bloat in how it generates vtables

What about unique_ptr?

What about using unique_ptr?

- Does not provide custom allocator support; nor was it designed for a custom allocator support in mind.
- High-risk of retrieving an invalid object with unique_ptr because of only allowing one object to reference that pointer.

How does unique_ptr work?



strong_ptr? What is it?

- Smart pointer that always gives you access to a not-null object and sharable

Goals of strong_ptr [original]

- Smart pointer that returned a non-null object
- Custom allocator support using pmr::polymorphic_allocator
- Memory-safe API's to never allow for accessing an invalid/potential null object from the pointer.
- **strong_ptr** operator* will always return a valid object
- **optional_ptr** interoperable with strong_ptr
- **optional_ptr** operator* checks and if object is invalid will throw bad_optional_access

Goals to using strong_ptr

- Smart pointer that returned a non-null object
- Memory-safe API's to never allow accessing a null object.
- **strong_ptr** operator* will always return a not-null object.
- **optional_ptr** interoperable with strong_ptr
- **optional_ptr** operator* checks and if object is null, then it will throw bad_optional_access

Not using shared_ptr

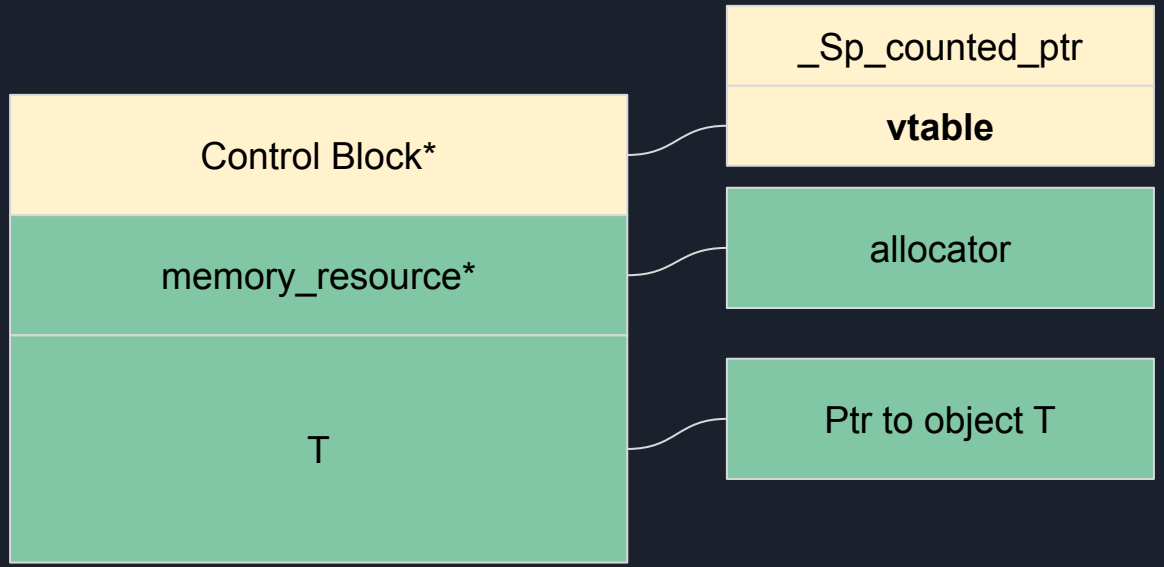
Not using shared_ptr

- Ability to be null
- Generates an extra vtable for every unique object of type T.

How does shared_ptr work?

How does shared_ptr work?

How is shared_ptr implemented (in libstdc++)?



Introducing of strong_ptr

What is strong_ptr?

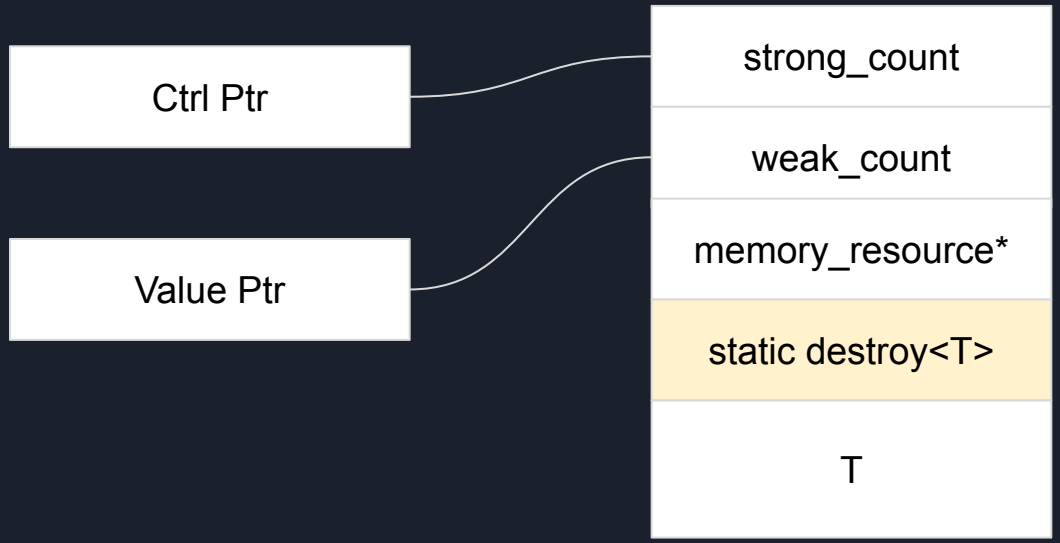
- Guarantees accessing a smart pointer to a non-null object
- Has custom allocator support with polymorphic_allocator.
- Provide memory-safe API’s to always access non-null objects.

We also have
Optional_ptr!

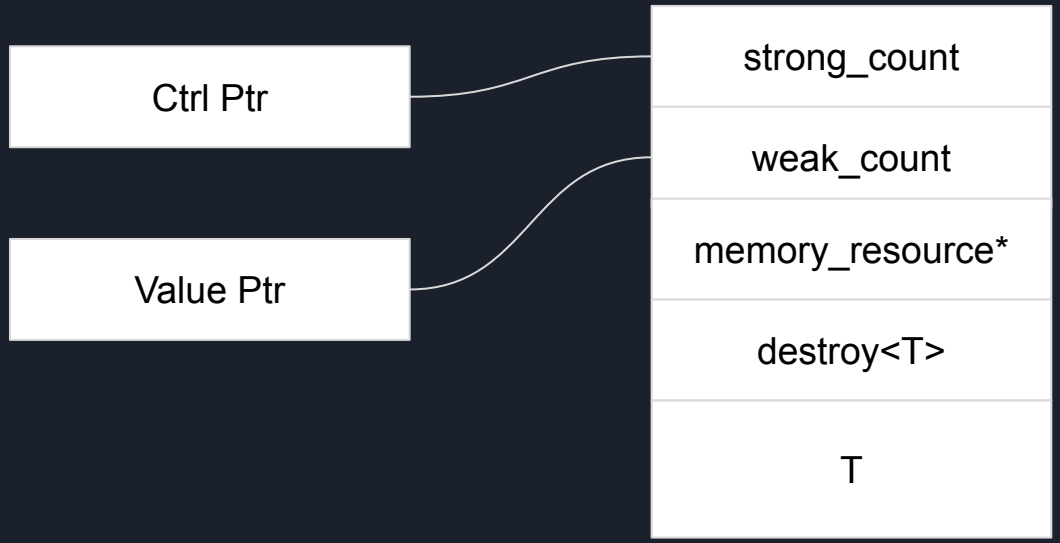
About optional_ptr

- optional_ptr is interoperable with strong_ptr.
- optional_ptr will check if null, then throw bad_optional_access or else access the non-null object.

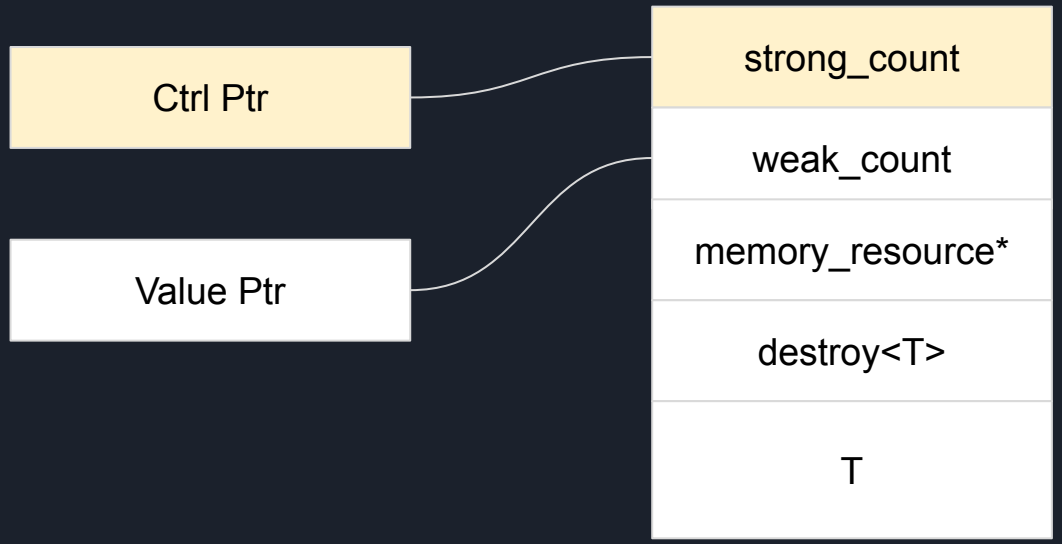
strong_ptr in TheAtlasEngine



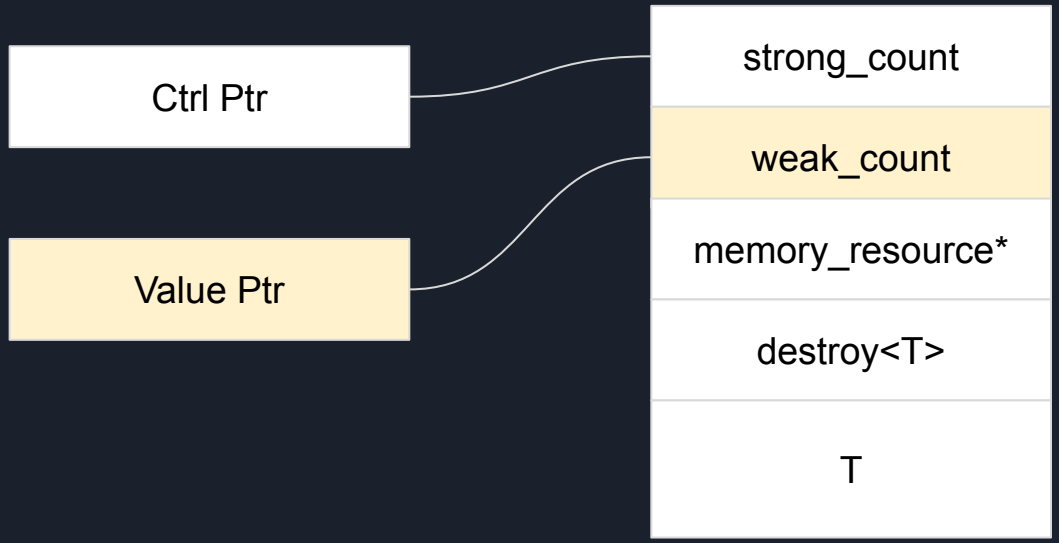
How does strong_ptr works?



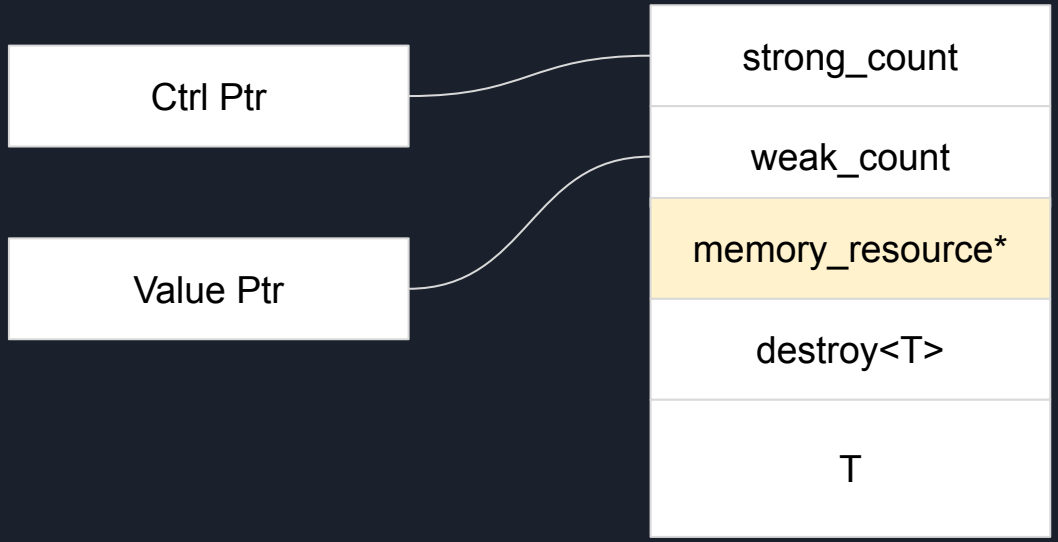
How does strong_ptr works?



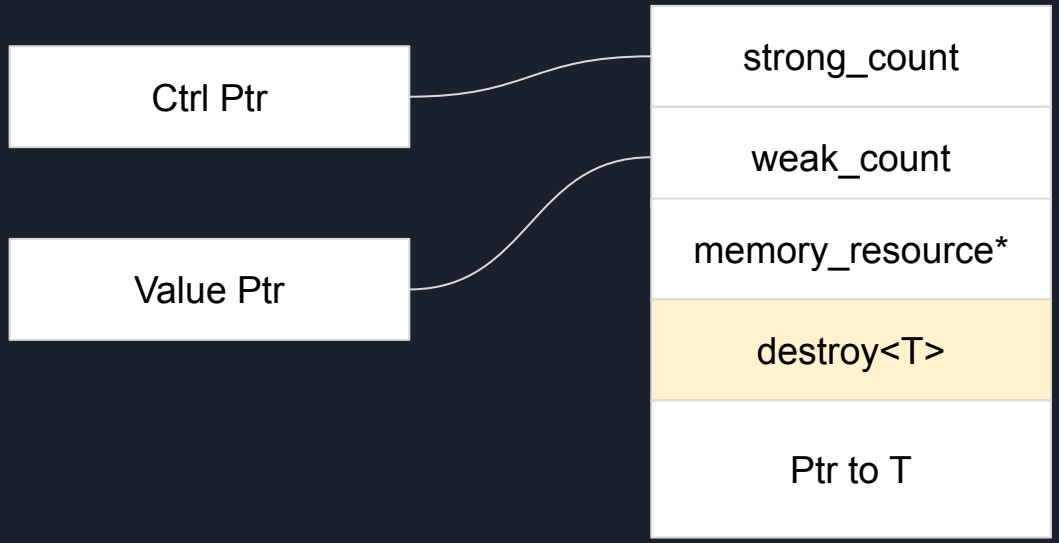
How does strong_ptr works?



How does strong_ptr works?

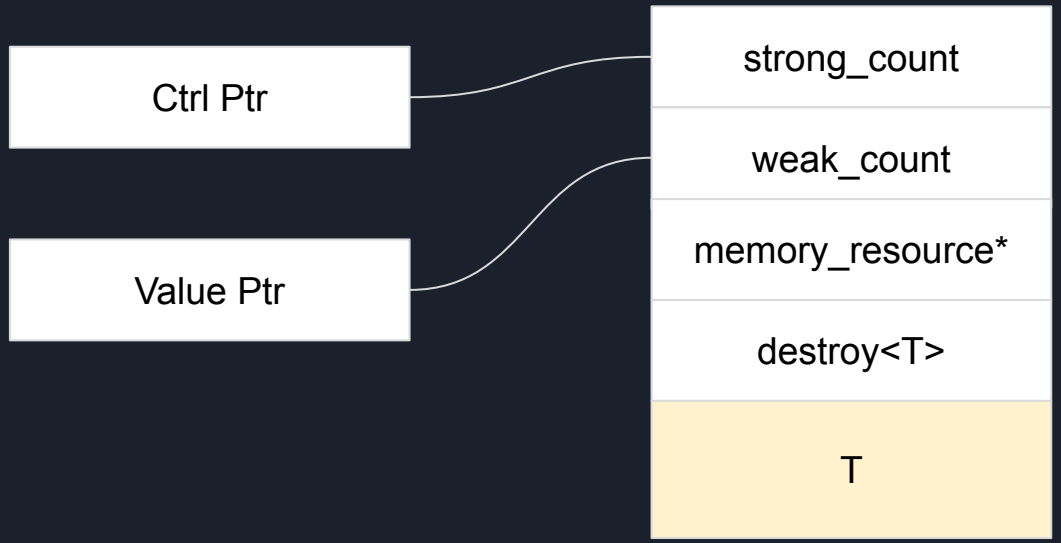


How does strong_ptr works?



What makes strong_ptr lightweight?

How does strong_ptr works?



strong_ptr destroy function

```
// Static function to destroy an instance and return its size
static size_t destroy_function(void const* p_object) {
      if (p_object != nullptr) {
            auto const* obj = static_cast<rc<T> const*>(p_object);
            // Call destructor for the object only
            obj->m_object.~T();
      }
      // Return size for future deallocation
      return sizeof(rc<T>);
}
```

strong_ptr destroy function

```
template<typename T>
struct rc {

      // Static function to destroy an instance and return its size
      static size_t destroy_function(void const* p_object) {
            if (p_object != nullptr) {
                  auto const* obj = static_cast<rc<T> const*>(p_object);
            // Call destructor for the object only
            obj->m_object.~T();
      }
      // Return size for future deallocation
            return sizeof(rc<T>);
      }
};
```

Creating a strong_ptr.

Creating a strong_ptr.

```
// Custom memory resource to allocate
std::array<std::byte, 1024> buffer;
std::pmr::monotonic_buffer_resource  my_buffer_resource(buffer.data(), buffer.size());

std::pmr::polymorphic_allocator<square> pmr(&my_buffer_resource);
strong_ptr<base_object> b = make_strong_ptr<base_object>(pmr);

strong_ptr<square> s = make_strong_ptr<square>(pmr);
```

Conclusion about strong_ptr

- Ensures objects can never be null.
- Reduce amount of state checking for a given object
- Does generate separate vtables

Then what became our conclusion?

Then what became our conclusion?

strong_ptr

- Always gives you a valid object or throws
- Resulting in a static call to the destroy function. No extra vtables created.
- Custom allocator support using pmr::polymorphic_allocator.
- Provide memory-safe API's to always access a valid object.

shared_ptr

- Gives you to access to an object that **maybe** alive.
- Code bloat due to vtables for dispose/destroy per unique object of type T

unique_ptr

- Higher risk of accessing an invalid object outside of its own scope.
- Lack of custom allocator support

Callbacks for Game State

What are goals in use of Callbacks

- Game objects do not store behavior code.
- Using callback for executing different behavior at different frame rates.
- Reduce code boilerplate and complexity with tracking object state.

Callback System

update_queue

physics_queue

Game object 1

Game object 2

Game object 3

Goals using callbacks for executing state

- Provide simple API’s to interact with TheAtlasEngine
- API’s to send state at different framerates simpler.
- Using callback for executing state at different framerates.

Callback system for game state

- Callbacks that can be executed at different frame rates.
- Minimize viral behavior for tracking game state.

Previously tracking state

Executing state

```
for(Layer* layer : _layerStack){
    layer->OnUpdate(timestep);
}

_imguiLayer->Begin();
for(Layer* layer : _layerStack){
    layer->OnUIRender();
}
_imguiLayer->End();
```

Introduces Viral Behavior

Viral Behavior in Code

```
class Layer {
public:
	Layer(const std::string& name = "Layer");
	virtual ~Layer();

	virtual void OnAttach() { }

	virtual void OnDetach() { }

	virtual void OnUpdate(Timestep ts) { }

	virtual void OnUIRender() { }

	virtual void OnEvent(Event& event){ }
};
```

Viral Behavior in Code

```
class EditorLayer : public Layer{
public:
	EditorLayer();

	void OnAttach() override;

	void OnDetach() override;

	void OnUpdate(Timestep ts) override;

	void OnEvent(Event& event) override;

	void OnUIRender() override;
};
```

Inspiration in using Callbacks

Inspiration using Callbacks

```
void Application::OnEvent(Event& event){
     EventDispatcher dispatcher(event);
     dispatcher.Dispatch<WindowCloseEvent>(bind(this, &Application::onWindowClose));
     dispatcher.Dispatch<WindowResizeEvent>(bind(this, &Application::onWindowResize));
}
```

Inspiration using Callbacks

```
void Application::OnEvent(Event& event){
    EventDispatcher dispatcher(event);
    dispatcher.Dispatch<WindowCloseEvent>(bind(this, &Application::onWindowClose));
    dispatcher.Dispatch<WindowResizeEvent>(bind(this, &Application::onWindowResize));
}
```

Registration Callbacks

Using Registration Callbacks

```
level_scene::level_scene(const std::string& p_name) {



    atlas::register_start(this, &level_scene::start);
    atlas::register_update(this, &level_scene::on_update);
    atlas::register_ui(this, &level_scene::on_ui_update);
}
```

Here is how it works!

Callbacks added to state queue

Adding callbacks to the state queue

```
template<typename UObject, typename UCallback>
void register_update(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::poll_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}

template<typename UObject, typename UCallback>
void register_physics(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::poll_physics_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}

template<typename UObject, typename UCallback>
void register_deferred(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::poll_defer_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}
```

Adding callbacks to the state queue

```
template<typename UObject, typename UCallback>
void register_update(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::add_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}

template<typename UObject, typename UCallback>
void register_physics(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::add_physics_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}

template<typename UObject, typename UCallback>
void register_deferred(UObject* p_instance, const UCallback& p_callable) {
     static_assert(std::is_member_pointer_v<UCallback>,
         "Cannot register a function that is not a member "
         "function of a class object");
     detail::add_defer_update([p_instance, p_callable]() { (p_instance->*p_callable)(); });
}
```

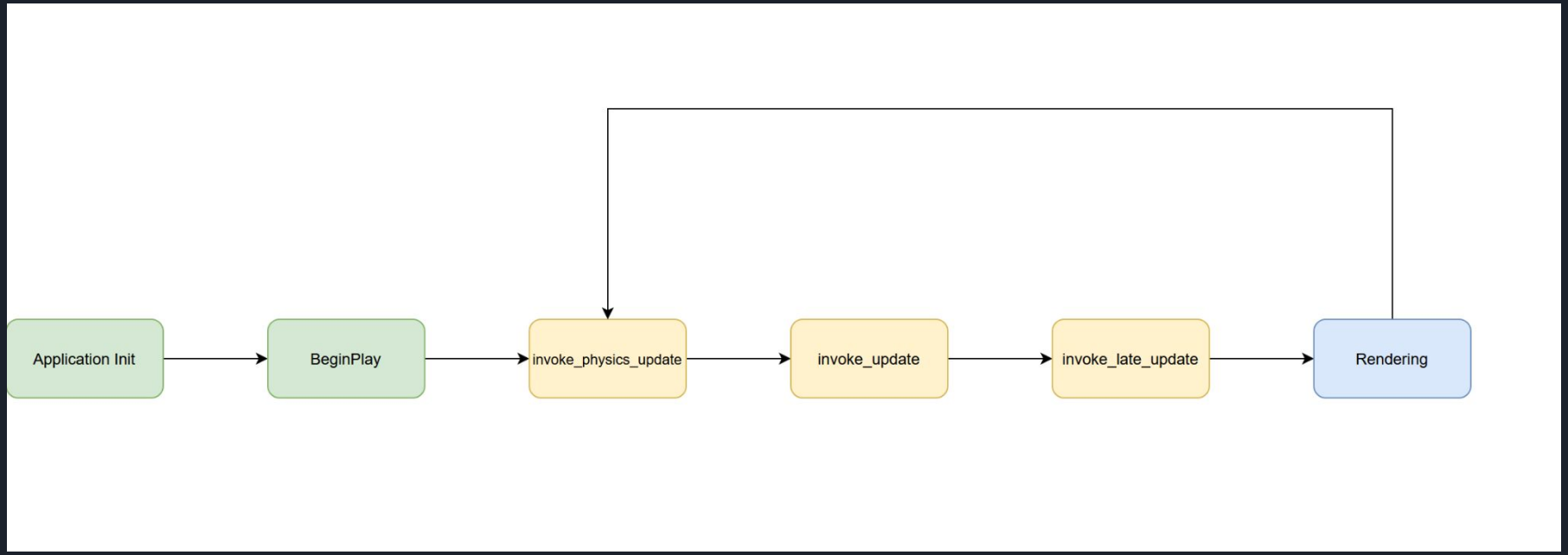
Adding callbacks to the state queue

```
void add_update(const std::function<void()>& p_callback) {
	update.emplace_back(p_callback);
}

void add_physics_update(const std::function<void()>& p_callback) {
	physics_update.emplace_back(p_callback);
}

void add_defer_update(const std::function<void()>& p_callback) {
	defer_update.emplace_back(p_callback);
}
```

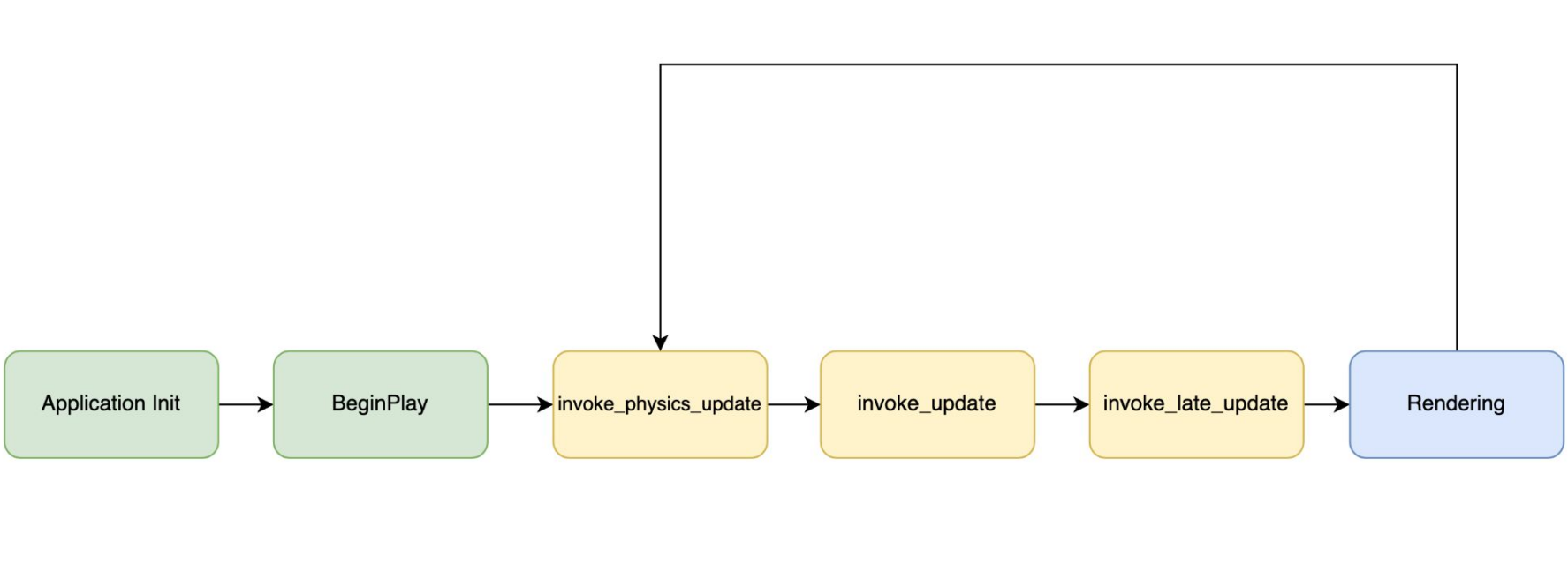
Executing Callbacks in the Game Loop

The Game Loop Execution Process

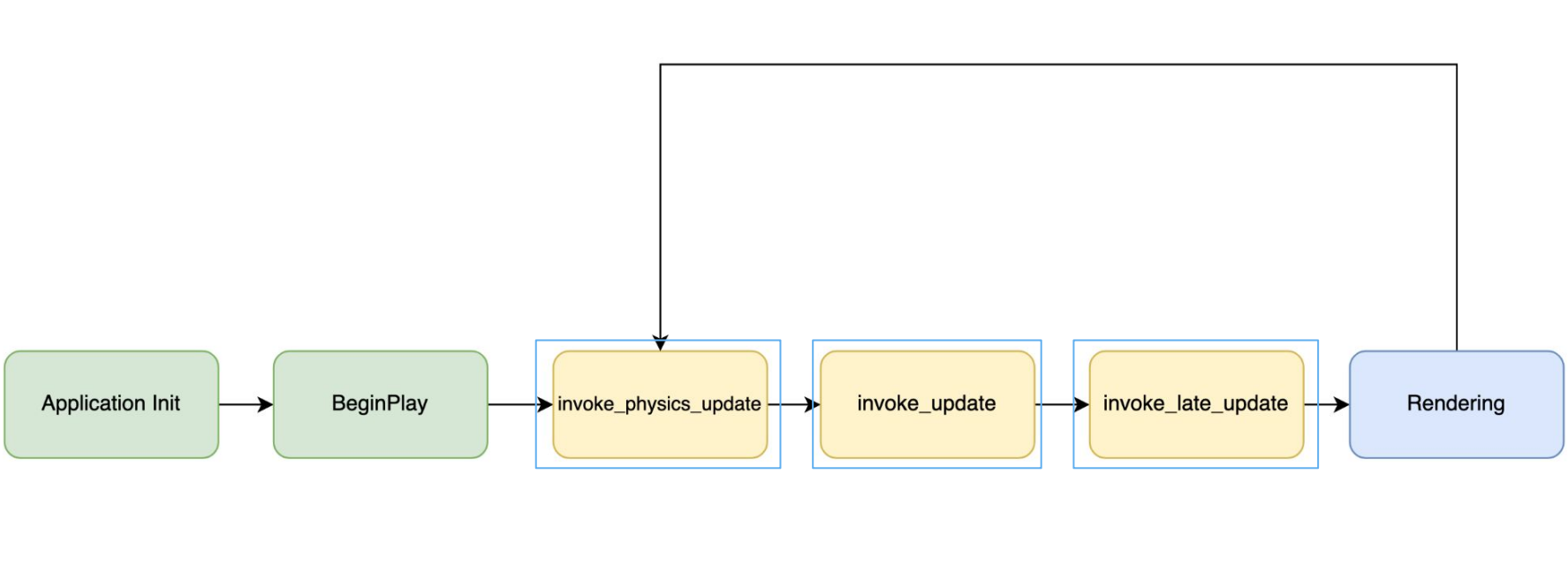
Executing state in Game Loop



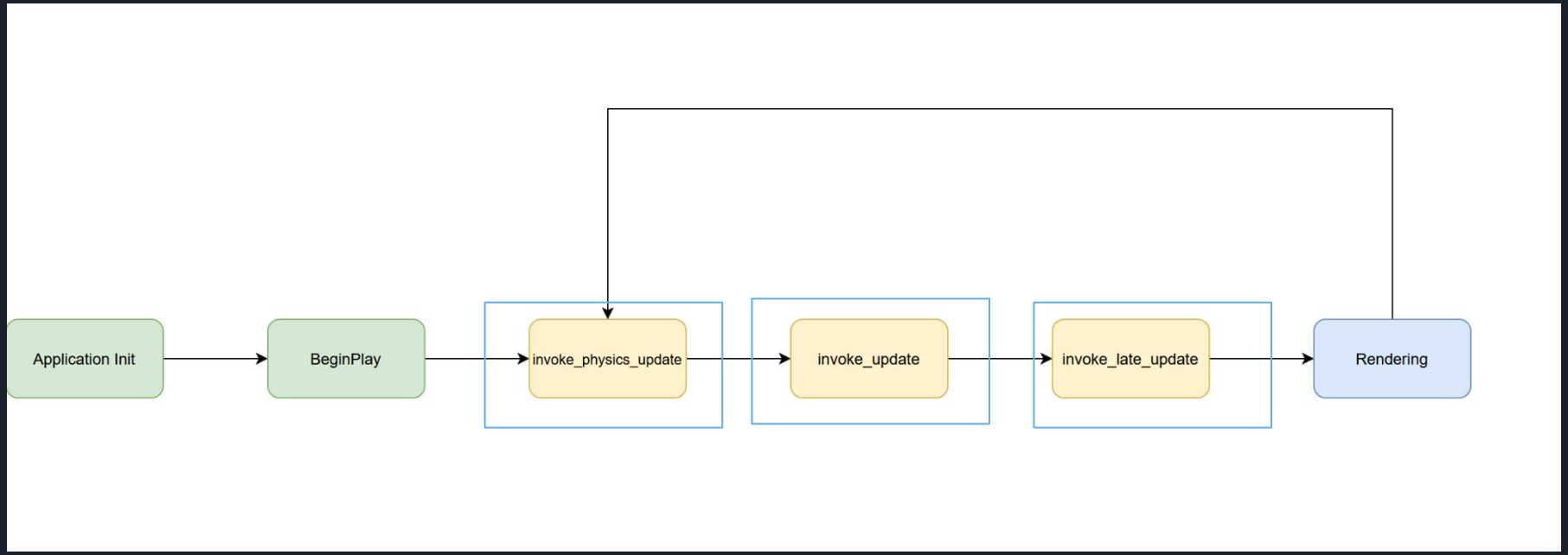
Executing state in Game Loop



Executing state in Game Loop



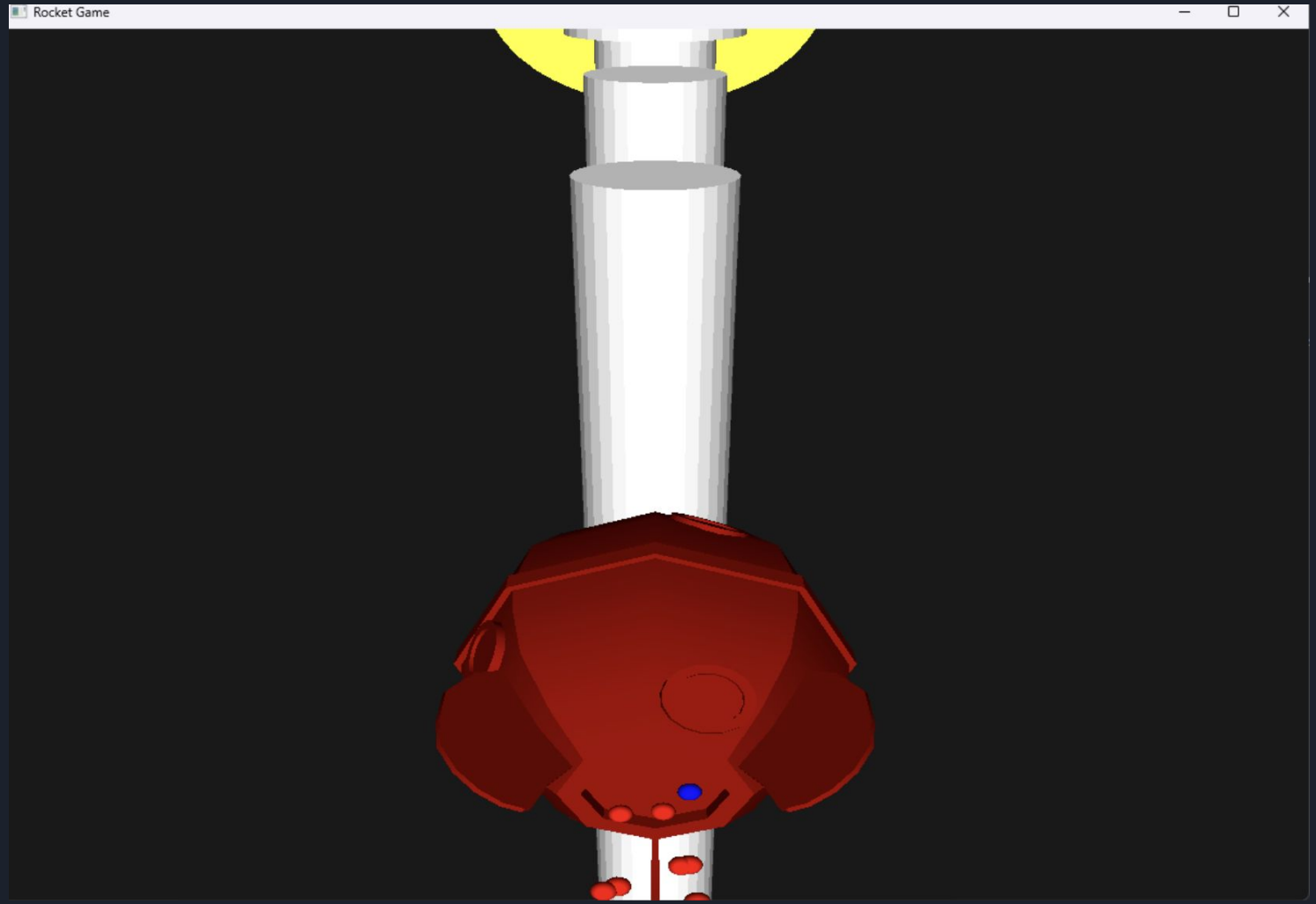
Executing state in Game Loop



Executing different callbacks

```
void invoke_on_update() {
      for (auto& on_update : update_queue) {
            on_update();
      }
}

void invoke_physics_update() {
      for (auto& on_update : physics_update_queue) {
            on_update();
      }
}

void invoke_defer_update() {
      for (auto& on_update : defer_update_queue) {
            on_update();
      }
}
```

Result in this Callback System


Rocket Game

Conclusion to Callbacks

Using this callback system

```
    some_object->set<atlas::material>(p_component: {
        .model_path = "assets/models/cube.obj",
        .texture_path = "assets/models/wood.png",
    });

    atlas::register_start(p_instance: this, p_callable: &level_scene::start);
    atlas::register_update(p_instance: this, p_callable: &level_scene::on_update);
    atlas::register_ui(p_instance: this, p_callable: &level_scene::on_ui_update);
}
```

Span for safe alternative for buffers

Using span<T> for contiguous buffers

- View and non-owning contiguous buffer
- Flexible for handling contiguous containers in C++.
- Reduce bug-prone situations in mismatching buffers and their size.

Here are some code examples

Flexible API’s with span<T>

```cpp
void print_span(std::span<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

void print_vec(std::vector<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    std::span<int> array_span(arr);  // Lightweight, no data copy
    std::vector<int> vec(arr, arr + 5);  // Allocates and copies data

    print_span(array_span);
    print_vec(vec);

    return 0;
}
```

Using span<T> for contiguous buffers

```cpp
void print_span(std::span<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

void print_vec(std::vector<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    std::span<int> array_span(arr);  // Lightweight, no data copy
    print_span(array_span);

    std::vector<int> vec(arr, arr + 5);  // Allocates and copies data
    print_vec(vec);

    return 0;
}
```

Using span<T> for contiguous buffers

```cpp
void print_span(std::span<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

void print_vec(std::vector<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    std::span<int> array_span(arr);  // Lightweight, no data copy
    print_span(array_span);

    std::vector<int> vec(arr, arr + 5);  // Allocates and copies data
    print_vec(vec);

    return 0;
}
```

Reduce code boilerplate

Using span<T> for contiguous buffers

```cpp
void print_span(std::span<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

void print_vec(std::vector<int> p_data) {
    for(auto val : p_data) {
        std::println("{}", val);
    }
}

int main() {
    int std::array<int, 5> arr = {1, 2, 3, 4, 5};
    std::span<int> array_span(arr);  // Lightweight, no data copy
    std::vector<int> vec(arr, arr.size());  // Allocates and copies data

    print_span(array_span);
    print_vec(vec);

    return 0;
}
```

Copying with vector<T>

```
print_vec(std::vector<int, std::allocator<int>>):          #
@print_vec(std::vector<int, std::allocator<int>>)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 112
        mov     qword ptr [rbp - 56], rdi
        mov     qword ptr [rbp - 64], rdi
        mov     rdi, qword ptr [rbp - 64]
        call    std::vector<int, std::allocator<int>>::begin()
        mov     qword ptr [rbp - 72], rax
        mov     rdi, qword ptr [rbp - 64]
        call    std::vector<int, std::allocator<int>>::end()
        mov     qword ptr [rbp - 80], rax
```

Copying with span<T>

```
print_span(std::span<int, 18446744073709551615ul>): # @print_span(std::span<int,
18446744073709551615ul>)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 112
        mov     qword ptr [rbp - 64], rdi
        mov     qword ptr [rbp - 56], rsi
        lea     rax, [rbp - 64]
        mov     qword ptr [rbp - 72], rax
        mov     rdi, qword ptr [rbp - 72]
        call    std::span<int, 18446744073709551615ul>::begin() const
        mov     qword ptr [rbp - 80], rax
        mov     rdi, qword ptr [rbp - 72]
        call    std::span<int, 18446744073709551615ul>::end() const
        mov     qword ptr [rbp - 88], rax
```

Span reduces code boilerplate

```
class buffer {
public:

    void write(const std::vector<uint32_t>& p_data) {}

    void write(uint32_t* p_data, size_t p_size) {}

    void write(const uint32_t p_arr[], size_t p_size) {}
};
```

Span reduces code boilerplate

```
class buffer {
public:

    void write(const std::vector<uint32_t>& p_data) {}

    void write(uint32_t* p_data, size_t p_size) {}

    void write(const uint32_t p_arr[], size_t p_size) {}
};
```

Span reduces code boilerplate

```
class buffer {
public:
    void write(std::span<uint32_t> p_data) {}
};
```

Conclusion in using span.

Using std::source_location

The usefulness of std::filesystem

- Cross-platform handler for filepaths
- Simple API for getting absolute or relative paths to your current directory.

API Functions

- **std::source_location**
- **std::filesystem::absolute**
- **std::filesystem::relative**

Let us talk about std::source_location

Code without using source_location

```
void vk_check(const VkResult& p_result, const std::string& p_name, uint32_t p_line, const
std::string& p_function_name)  {
      if (p_result != VK_SUCCESS) {
       console_log_error_tagged(
            "vulkan",
            "File {} on line {} failed VkResult check",
            p_line);
       console_log_error_tagged("vulkan",
            "Current Function Location = {}",
            p_function_name);
      console_log_error_tagged(
            "vulkan", "{} VkResult returned: {}", p_name, (int)p_result);
      }
}
```

Code without using source_location

```
vk_check(
          vkCreateCommandPool(m_driver, &pool_ci, nullptr, &m_command_pool),
          __FILE__,
          __LINE__,
          __FUNCTION__);
```

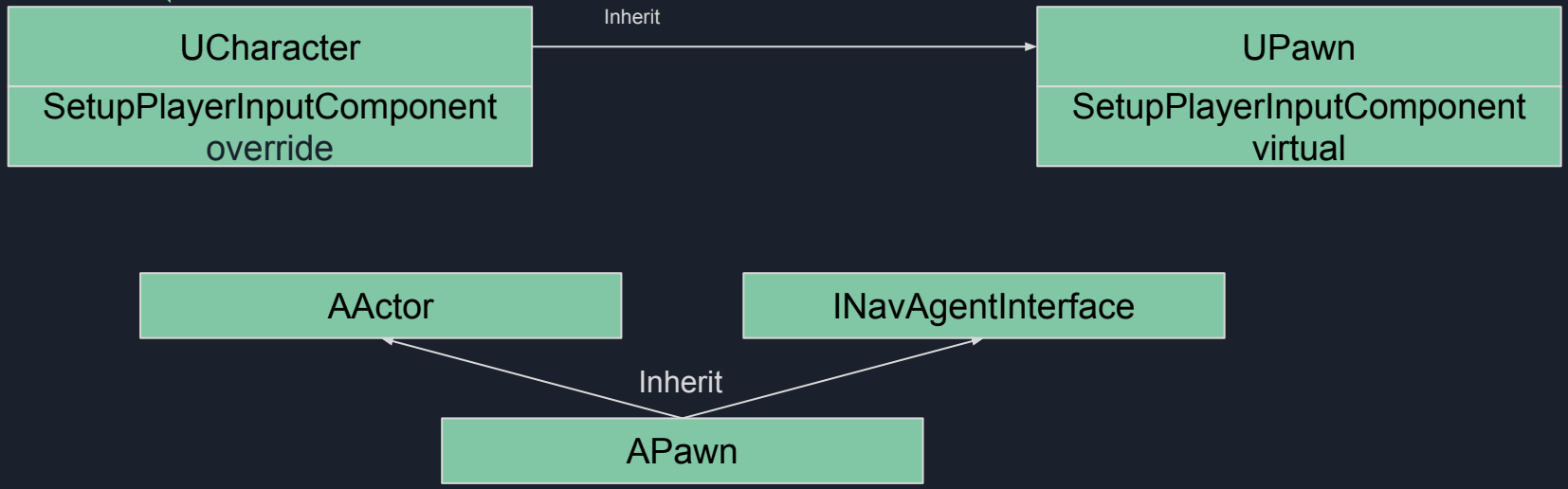
Simplify debugging with source_location

```
void vk_check(const VkResult& p_result,
                  const std::string& p_name,
                  const std::source_location& p_source) {
        if (p_result != VK_SUCCESS) {
            console_log_error_tagged(
              "vulkan",
              "File {} on line {} failed VkResult check",
              std::filesystem::relative(p_source.file_name()).string(),
              p_source.line());
            console_log_error_tagged("vulkan",
                                     "Current Function Location = {}",
                                     p_source.function_name());
            console_log_error_tagged(
              "vulkan", "{} VkResult returned: {}", p_name, (int)p_result);
        }
    }
```

Simplify debugging with source_location

```
vk_check(
          vkCreateCommandPool(m_driver, &pool_ci, nullptr, &m_command_pool),
          "vkCreateCommandPool");
```

Hierarchies Unreal Engine Example



Data-Oriented for Game Objects

- Store only data that represent game objects
- Do not store state within themselves
- Users are given a simpler API’s to have their state be invoked by TheAtlasEngine.

Looking forward to C++26

Upcoming C++26 Features

- std::hive
- reflections

Any Questions?

Thank You for Attending!

Skipped Slides after this slide...!

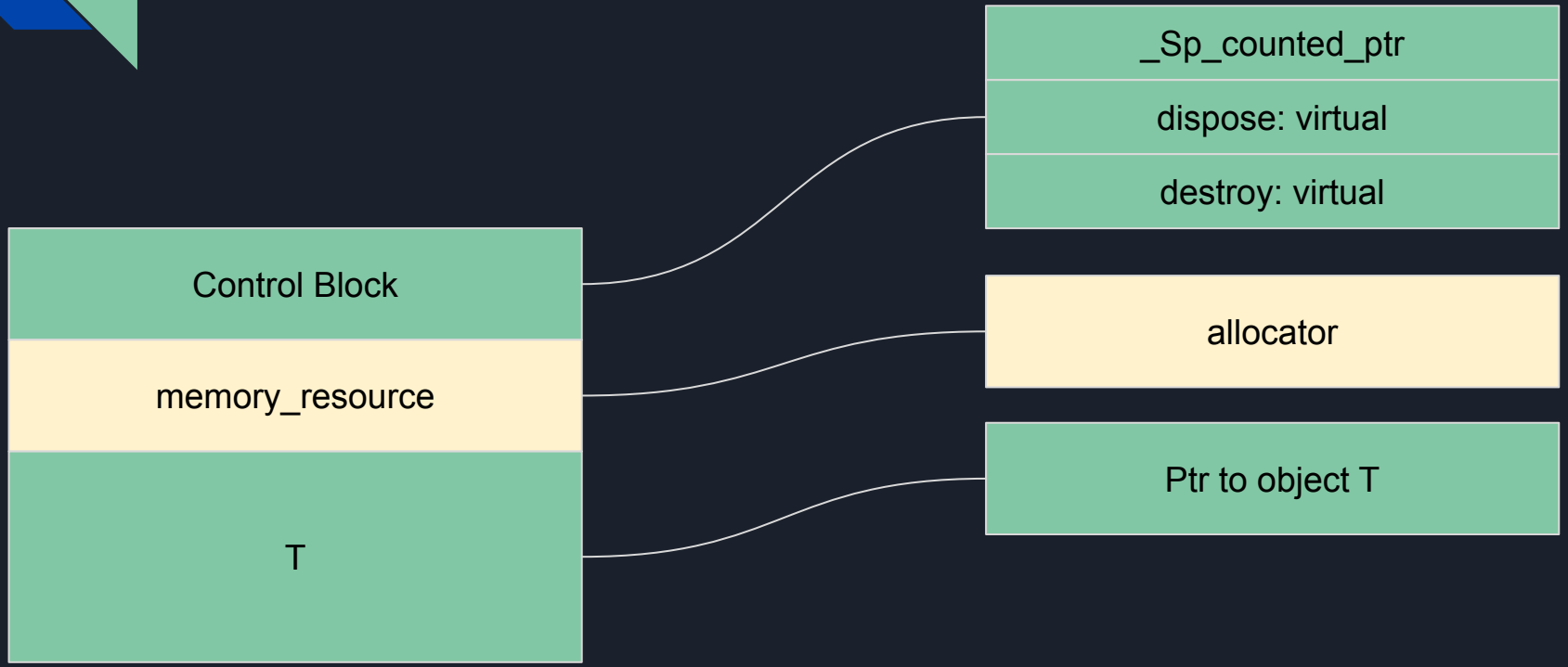
Vtable layout

```
.L__const._ZNSt8__detail10__to_charsIoEESt15to_chars_resultPcS2_T_i._digits:
        .ascii  "0123456789abcdefghijklmnopqrstuvwxyz"

.L.str.51:
        .asciz  "format error: unmatched '}' in format string"

vtable for std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>:
        .quad   0
        .quad   typeinfo for std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>
        .quad   std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [base object destructor]
        .quad   std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [deleting destructor]
        .quad   std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
        .quad   std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
        .quad   std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter(std::type_info const&)

typeinfo for std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>:
        .quad   vtable for __cxxabiv1::__si_class_type_info+16
        .quad   typeinfo name for std::_Sp_counted_ptr_inplace<sub_class, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>
        .quad   typeinfo for std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>
```

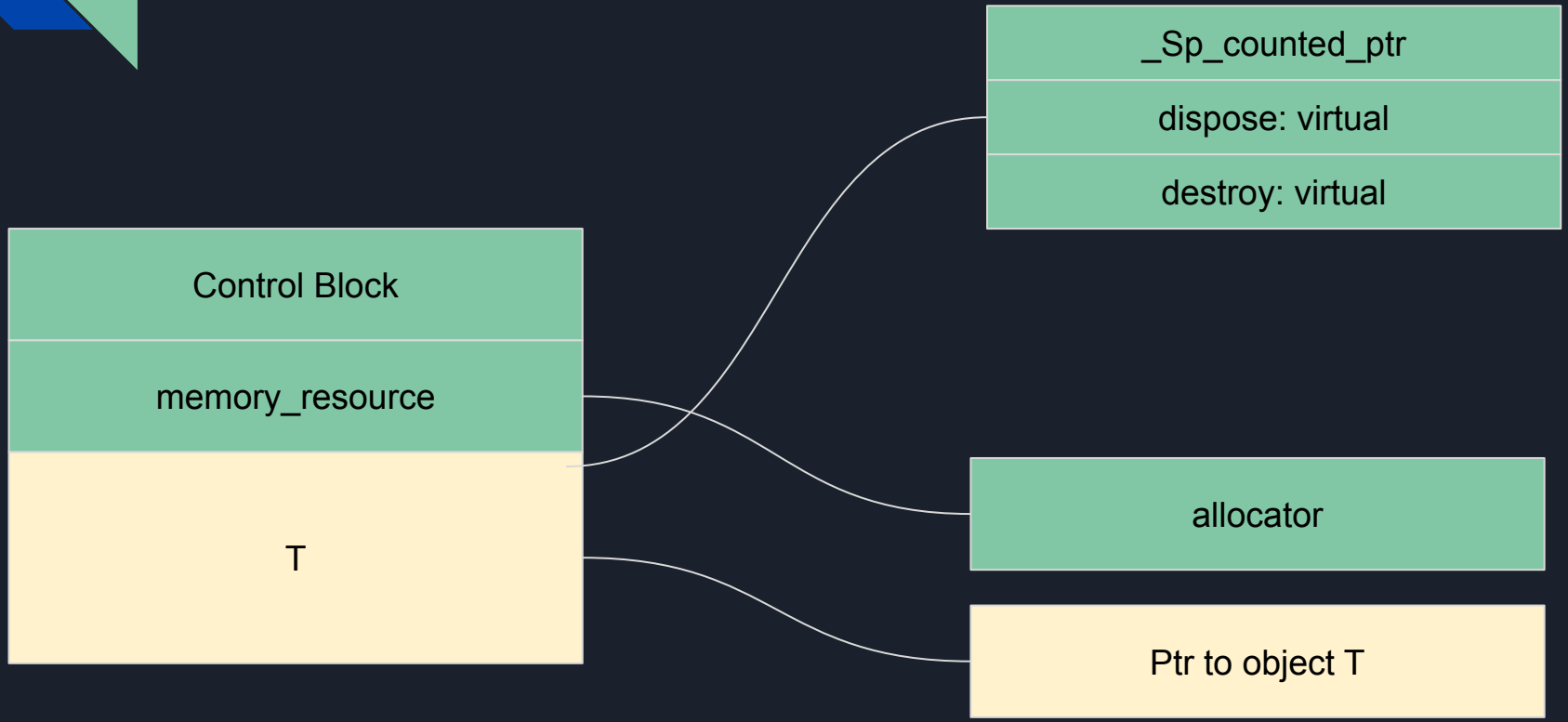
Offset_from top

rtti Info

Ref Counting System in shared_ptr (libstdc++)

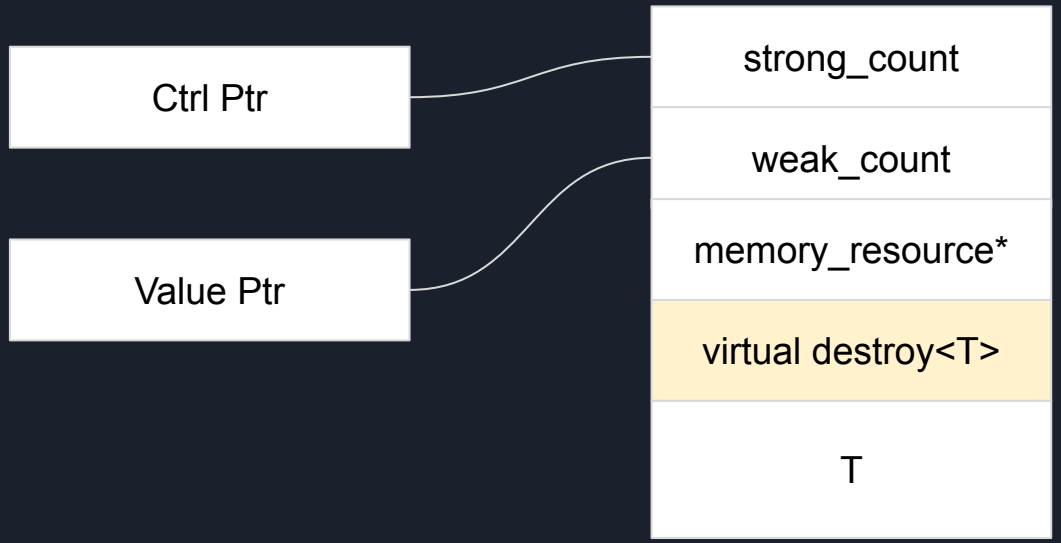


Ref Counting in shared_ptr (libstdc++)



Code bloat with vtables in shared_ptr.

Code bloat in shared_ptr?



Code bloat in shared_ptr?

```cpp
class test {
public:
    test() {
        std::println("Explicitly calling default constructor");
    }

    test(const std::string& p_name) {
        std::println("Explicitly calling constructor with p_name param = {}", p_name);
    }

    ~test() {
        std::println("explicitly calling destructor!");
    }
};

class square {
public:
    square() {
        std::println("Explicitly calling default constructor");
    }

    square(const std::string& p_name) {
        std::println("Explicitly calling constructor with p_name param = {}", p_name);
    }

    ~square() {
        std::println("explicitly calling destructor!");
    }
};

int main() {
    std::shared_ptr<test> test_ptr = std::make_shared<test>("custom name");

    std::shared_ptr<square> square_ptr = std::make_shared<square>("custom Square");
}
```

Code bloat in shared_ptr?

```
vtable for std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>:
        .quad   0
        .quad   0
        .quad   std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [complete object destructor]
        .quad   std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [deleting destructor]
        .quad   std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
        .quad   std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
        .quad   std::_Sp_counted_ptr_inplace<square, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter(std::type_info const&)
vtable for std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>:
        .quad   0
        .quad   0
        .quad   std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [complete object destructor]
        .quad   std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr_inplace() [deleting destructor]
        .quad   std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
        .quad   std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
        .quad   std::_Sp_counted_ptr_inplace<test, std::allocator<void>, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter(std::type_info const&)
vtable for std::__format::_Formatting_scanner<std::__format::_Sink_iter<char>, char>:
        .quad   0
        .quad   0
        .quad   std::__format::_Formatting_scanner<std::__format::_Sink_iter<char>, char>::_M_on_chars(char const*)
        .quad   std::__format::_Formatting_scanner<std::__format::_Sink_iter<char>, char>::_M_format_arg(unsigned long)
vtable for std::__format::_Iter_sink<char, std::__format::_Sink_iter<char>>:
        .quad   0
        .quad   0
        .quad   std::__format::_Iter_sink<char, std::__format::_Sink_iter<char>>::_M_overflow()
        .quad   std::__format::_Sink<char>::_M_reserve(unsigned long)
        .quad   std::__format::_Sink<char>::_M_bump(unsigned long)
vtable for std::__format::_Scanner<char>:
        .quad   0
        .quad   0
        .quad   std::__format::_Scanner<char>::_M_on_chars(char const*)
        .quad   __cxa_pure_virtual
vtable for std::__format::_Seq_sink<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>:
        .quad   0
        .quad   0
        .quad   std::__format::_Seq_sink<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>::_M_overflow()
        .quad   std::__format::_Seq_sink<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>::_M_reserve(unsigned long)
        .quad   std::__format::_Seq_sink<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>::_M_bump(unsigned long)
```

Virtual destroy

Virtual destroy

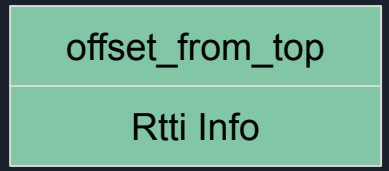
What about unique_ptr??

- unique_ptr does not have custom allocator support.
- Does not have share semantics as shared_ptr. Want to have ability to share things.
- **safety thing**: Accessing a moved unique_prt is unsafe.
- **shared_ptr**: has a move and copy that does the same thing. So, you can never have a strong_ptr that is invalid to access within a scope that can access it.
- If you have A (that is unique). Then move that to B. A is now invalid.
- Whereas with strong_ptr you can always reference it. Unique_ptr does not guarantee.
- **performance note**: Because we make copies each time, we have to perform an atomic increment/decrement for all valid objects around. It will always make a copy to increment/decrement. Whereas shared_ptr skips increment/decrementation when you move.

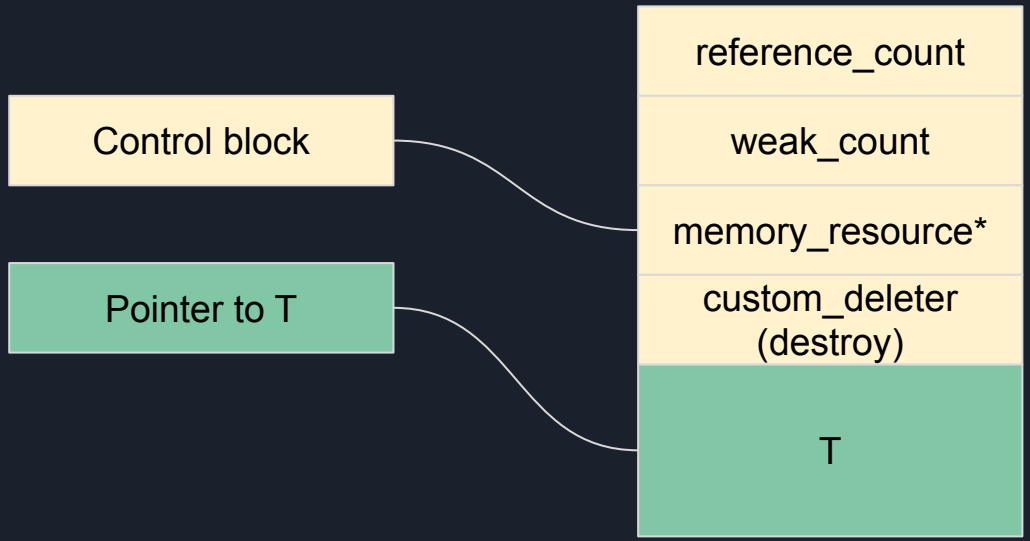
Unique_ptr incompatibility custom allocator support

- Difficult for enabling custom allocator support
- Unique_ptr does not support pmr::polymorphic_allocator out of the box.

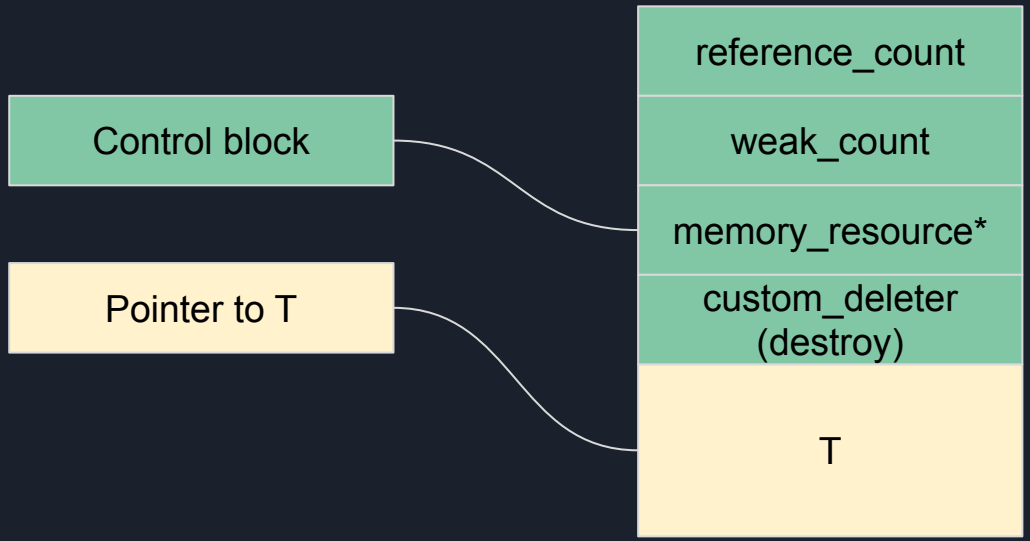
Diagram of explaining vtable



How does shared_ptr work?



How does shared_ptr work?



Issues with unique_ptr

C++ code use of strong_ptr

Game object designed

- Data-oriented
- Do not execute state themselves
- Use data to represent type of game objects in a game.

API Designs

- Simpler API Design
- Provide guarantees in the utilization of those API’s.
- Minimize side effects using these API’s.

Contributor Project Workflow

- Working CI for pull-requests
- Ensure code quality assurance through the use of CI
- Approach at writing tests for code testability

Heavy Focus in Memory Safety in C++

- Thinking about how we can write code in a memory-safe manner.
-

Using Conan + CMake

- Reduce CMake maintenance in TheAtlasEngine
- Reduce maintaining CMake modules managing dependencies.
- Simplify build system workflow

Designs in TheAtlasEngine

- Simple API design
- Ensure creation and manage lifetime of objects.

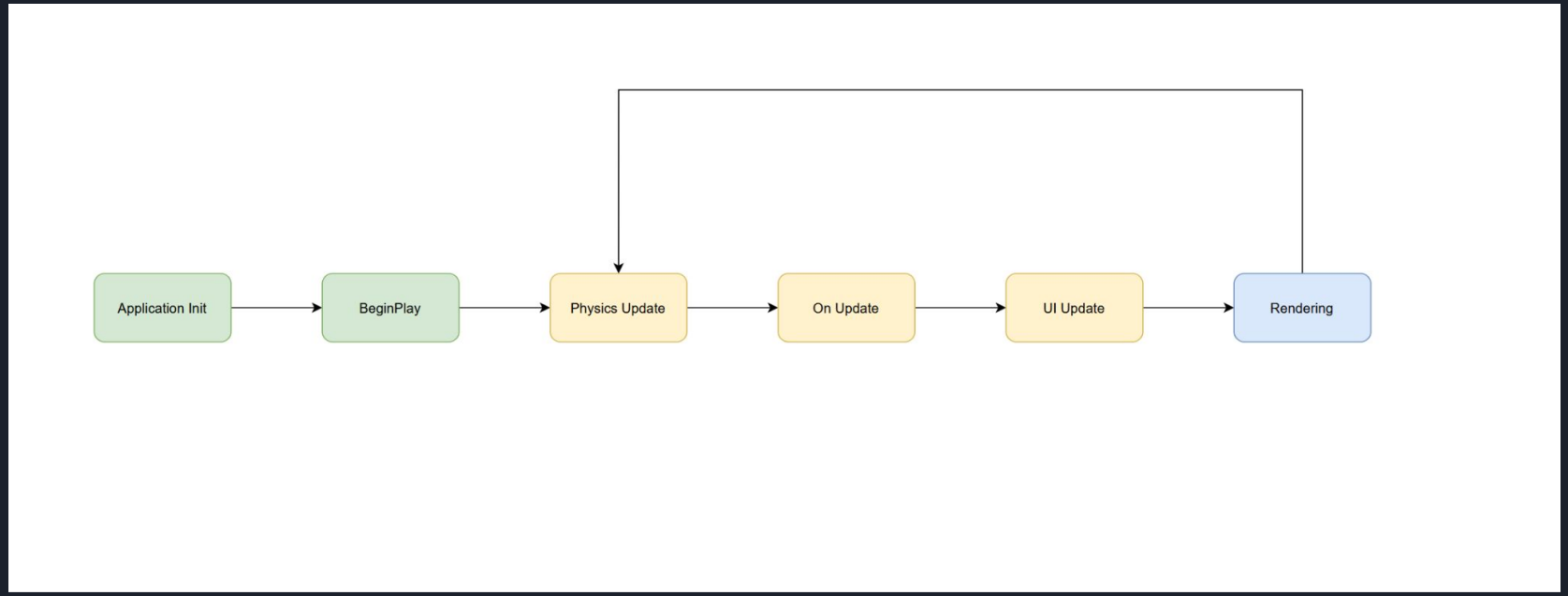
Objects in TheAtlasEngine

- Goal to reduce object lifetime issues
- Guarantees of objects are alive until the end of their scope
- Reduce needs to state-checks your objects

Refresher: Mechanism of Game Mainloop’s

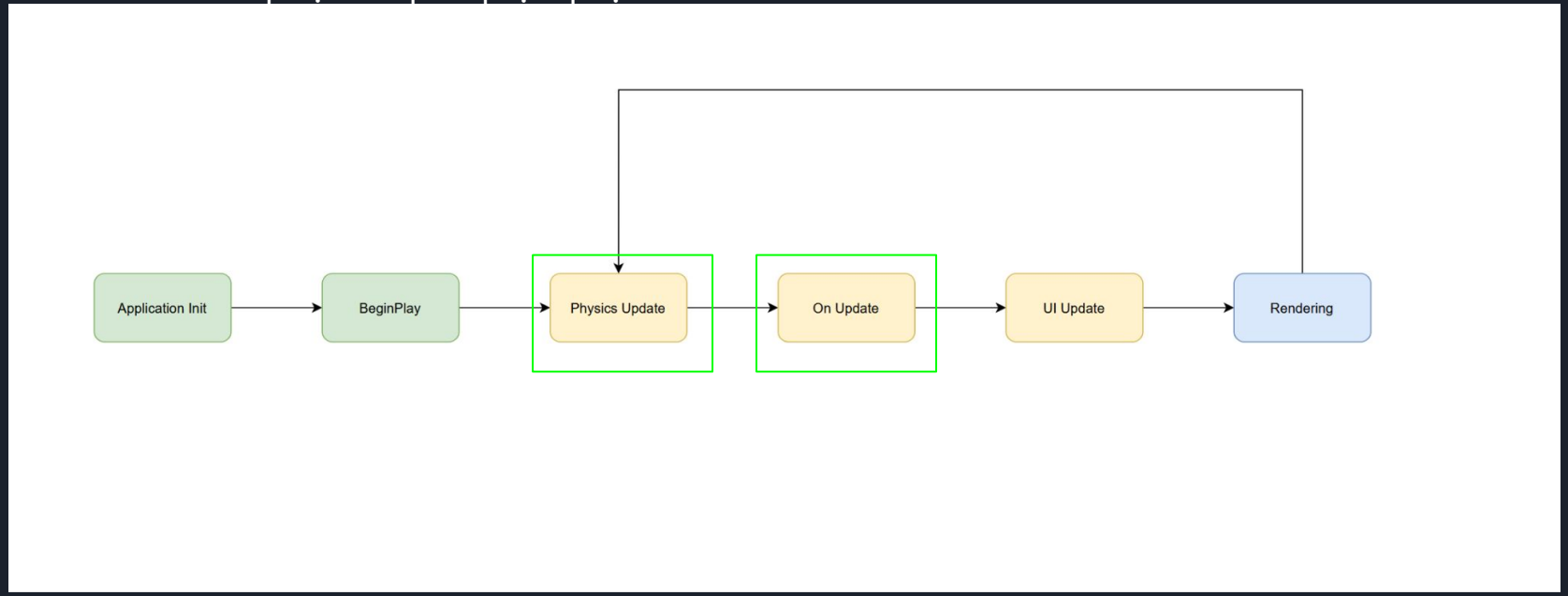
Define Game Mainloop

- Control your game and physics logic at different intervals in the game mainloop.



Mechanism Executing in Game Mainloops

- Game mainloop actual event execution process demonstration for game input logic,



Importance of Updates Between Frame Rates

- Physics need to run at a fixed framerate and not as fast as the game loop.
- Game logic relies on game loop frame rate for smooth interaction such as with keyboard inputs.
- Example: Physics most likely will be computationally more heavier. Then the update logic that simply handles user inputs.
- Example: Game engines could have opportunity to optimize certain logic that may be computationally expensive.

Representing game objects in TheAtlasEngine

- Creating game objects return strong_ptr/optional_ptr.
- Data-oriented design applied to game objects.
- memory-safety with strong_ptr

```
class scene_scope {
public:
    scene_scope() = default;
    scene_scope(const std::string& p_name)
      : m_name(p_name) {}

    strong_ref<scene_object> create_object(const std::string& p_tag) {
        return create_strong_ref<scene_object>(
          p_allocator: m_allocator, &m_registry, p_tag);
    }
```

What is strong_ptr?

- Pointer wrapper ensures pointer points to alive object
- Takes in a pmr::polymorphic_allocator
- Not use the global allocation to allocate objects.

```
template<typename T, typename... Args>
strong_ref<T> create_strong_ref(
  std::pmr::polymorphic_allocator<> p_allocator,
  Args&&... args) {
    return memory::make_strong_ptr<T>(p_allocator,
                                      std::forward<Args>(args)...);
}
```

Rationale: Not using shared_ptr/unique_ptr

- Not guarantee the object is always alive
- Lack of custom allocator support
  - Has support for custom deleters. Learned it can be much work to bend shared_ptr/unique_ptr to work with custom allocators which can take a lot of work.
  - Lot of work of trying to get custom deleters to work with specialized types.

shared_ptr incompatibility with custom allocators

- Custom deleters exist, issues is trying to conform shared_ptr/unique_ptr to work with custom allocators.
- Lot of work of trying to get custom deleters to work with specialized types.

strong_ptr in TheAtlasEngine

- Guarantees pointer always points to an alive object.
- Does not use global allocator for memory allocations.
- Flexibility supporting custom allocators using polymorphic_allocator

Guarantees of strong_ptr

- Guarantee pointer is referencing to a valid object.
- Reduce state-checks, if the object can be null.
- Remove any potential a pointer can be null, when accessing that object.

strong_ptr supports custom allocators

- Strong_ptr implemented with custom allocators in mind.
- Using pmr::polymorphic_allocator as part of its implementation

How does strong_ptr work?

- Strong_ptr requires pass of a pmr::polymorphic_allocactor
- Removing use of the global allocator in TheAtlasEngine

```
template<typename T, typename... Args>
strong_ref<T> create_strong_ref(
  std::pmr::polymorphic_allocator<> p_allocator,
  Args&&... args) {
    return memory::make_strong_ptr<T>(p_allocator,
                                      std::forward<Args>(args)...);
}
```

Game Object State in TheAtlasEngine

- Looking at our options.
  - What is an ECS?
  - Using Inheritance
  - Using Hybrid of Inheritance + ECS.
- Using callbacks to represent state
- Reduce code boilerplate

Game Object State in TheAtlasEngine

- Reduce code boilerplate
- Data-oriented design for game objects.
- Remove conforming user-defined state to some hierarchical system.

Game Object State in TheAtlasEngine

- Using callbacks to store in state, that get called at different frame rates.
- API's with `register_*` functions are uploading state that get invoked in the game loop.
- Example: `register_physics` would register callback at physics logic frame interval.

Looking at our options.

- Inheritance
- Hybrid of Inheritance + ECS

Option 1: Rationale not using Inheritance

Option 1: Rationale not using Inheritance

- Complexity increases modeling game object state in a one to many relationship hierarchy system.
- Requires users to conform to the hierarchical system.
- Difficulty in modeling one to many relationship of every state a game object can potentially be in.
- Reduce cost of vtable/thunks getting created
- Invites the use of multiple inheritance.

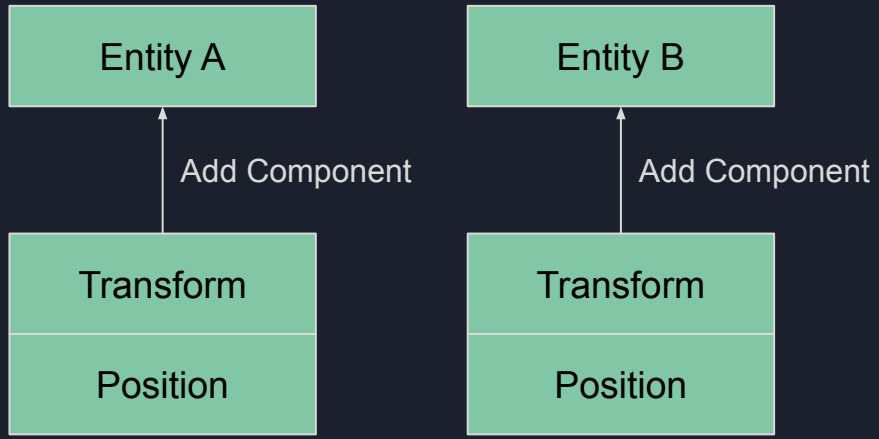
Option 1: Rationale not using Inheritance

Option 2: Rationale not using Inheritance + ecs

What is an ECS (Entity-Component-System)?

- Game objects referred to as entities.
- Separate data from behavior (state)
- Attaching components (data) to game objects
- Manage complex interactions for processing data during specific game logic execution.

What is an (Entity-Component-System)?



Hybrid of Inheritance + Ecs

- Still represents a one to many relationship + components.
- Game object themselves inherit from base class to represent specific gameobject state of access.
- Components are data, containing their own state represented using inheritance.
- [add visual diagram to show hierarchy]

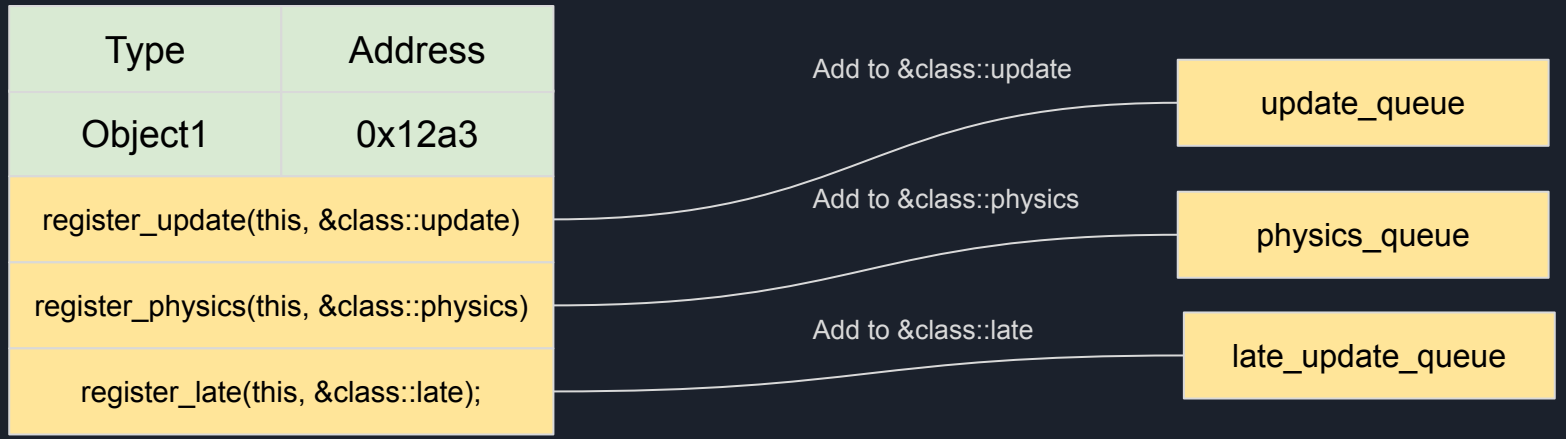
Using callbacks to representing state.

- Reduce boilerplate and requirement of the user to conform to a hierarchical system.
- Allow users to choose where their state get executed and how choose to interact with TheAtlasEngine.

Why using callbacks?

- Reduce code complexity/boilerplate.
- Reduce cost of vtable/thunks get created.
  - Reduce vtable sizes that get created per game objects
  - Reduce thunks creation & minimize lookup times for inherited game objects.
  - Overlapping of the same/similar states
- Reduce game object state coupling.
- Simpler to API design with state interaction.

Using callbacks to representing state.



Using callbacks to representing state.

Centralizing containers using span

Centralized containers using span

- Safe alternative view of your data buffers
- Reducing code boilerplate to support contiguous containers
- Remove needing to copy from temporary data buffers
- Reduce situations that are bug-prone with varied sizes in contiguous buffers

Reduce code boilerplate

```
/**
 * @brief Maps buffer handler to chunk of data of type, that is
 * std::span<uint32_t>.
 */
void write(const vk_buffer& p_buffer,
           const std::span<uint32_t>& p_in_buffer);

/**
 * @brief Maps buffer handler to data chunks that contain vertices
 */
void write(const vk_buffer& p_buffer,
           const std::span<vertex_input>& p_in_buffer);
```

Issues without using std::span

- Increase code boilerplate to handle any form of contiguous buffer
- Bug-prone to handle contiguous buffers with varied lengths in sizes.
- Require allocation to view data buffers that only need to be read rather then written to.

Use of std::span

- Ability to provide flexible API's for any contiguous buffer supported in the standard.
- view-only to data buffers
- Reduce amount of temporary variables needed to be created

Cross-platform path handling with std::filesystem

- Portability across cross-platform
- Simplifying code maintenance and validation
- Advantages of not needing to write cross-platform code for handling file paths
- Improves readability and expressiveness with logic involving file paths
- Functionality
  - **filesystem::regular_path**
  - **filesystem::exists**
  - **filesystem::absolute_path**
  - **filesystem::relative_path**

filesystem::regular_file

- Allowed for verifying for valid input files

```
shaderc::Compiler compiler;

//! @note Read from that file if the parameter is a file itself
if(std::filesystem::is_regular_file(p_filename)) {
    std::string shader_raw_source_text = read_file(p_filename);
```

Conclusion

FAQ

Thank you for attending!

[Slides afterward are skipped…]

What changes are in TheAtlasEngine?

- Ways we handle renderable objects
- Expressing game object state with a different approach in mind.
- Using Conan + CMake for handling all of our dependencies.
- Creating external tools to help with project management.
- Re-designing the physics system and renderer.

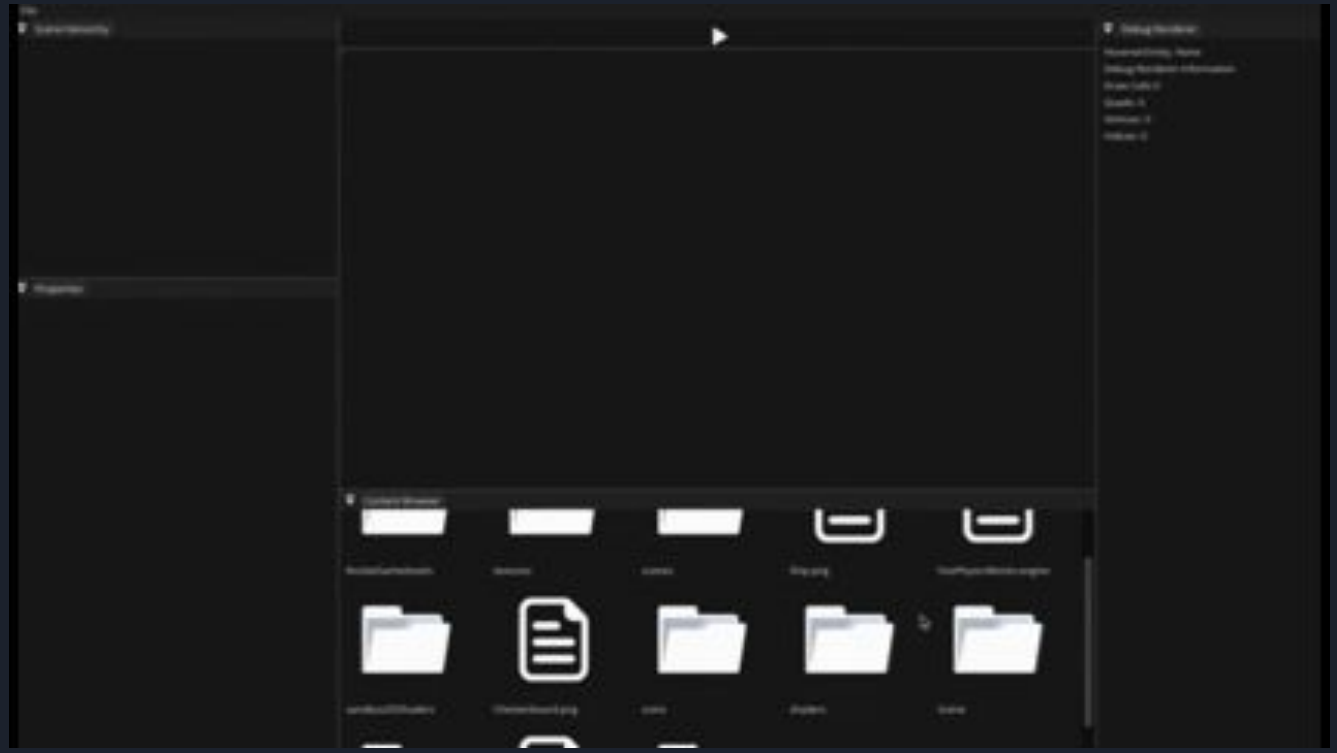
Extensive Toolchain Usability

- Using conan to handle TheAtlasEngine's third party dependencies
- Simplifying CMake modules through the use of Conan.

The Engine3D Project

- Engine3D was a project that I started a few years back. This project started off, when I first began to watch TheCherno's game engine series.
- I was following along trying to get a grasp at how I, an undergraduate student could begin to start a project so complex as a game engine.
- This project began as just a way for me to understand what is involved when building a game engine.
- I went through many iteration of learning, researching, YouTube video watching, and lot of learning of on how this could be done.
- As an undergraduate student, one of the pitfalls any student like myself fall into, is saying we want to start a project but never really pursuing it due to burnout among other things. This project was a project that helped me break out of that cycle.

The Engine3D Project



Purpose of this project?

- Get experiences in familiarizing myself with a system that makes up building a game engine and learning how they work.
- Lear widely about how ecs's, scene management, serialization, instrumentation, and profiling work in game engines.
- Learning through various tutorials about how to implement these various systems to work as a whole.
- At the same time, I was using raw CMake as the build system with all of my dependencies before I learned of Conan.

What Engine3D taught me?

- How to coordinate between scene management, physics simulation, and rendering altogether.
- It taught me in implementing my own serializer for multiple scenes.
- I learned about instrumentation and profiling; to benchmark parts of Engine3D to a json file format.

Inheritance – show diagram of just inheritance

- Our options that we have seen many times was using inheritance to represent game objects.
- These options involved a variety of approaches that had inheritance involved in the process
- This included representing game objects themselves, where if you wanted to consider every known factor for that object was handled. You'd express it in a relationship model. Where an object can be one to many things.
- Additionally including, using a mixture of inheritance + game objects.
    - Where you may use components to contain the state and data of what the object could be.
    - Where components may have it's own sets of hierarchical structure on how it is handles its state.
    - Such as that the component if being something as a collider. Can contain state both renderable and physics-based.
    - Which is something that can be added to a game object, where potentially that game object may also inherit from some sets of base classes to ensure that the object also has known state to work enough with physics and to imply it is a renderable object.

Ways game objects state are defined

- Game objects were something I brought up in the early development of TheAtlasEngine.
  - These discussions involved looking at our options for handling game object state.
  - Diving into the complexities between using inheritance and looking at how we may want to define them within the project.
  - How this is going to be represented. Looking at what things that you would want your object to be known?
  - Which was how its state gets updated? And when these updates happen relative to the object itself.
- I spent majority of the development in defining how TheAtlasEngine utilizes and handle game object state in a data-oriented design using an ECS.

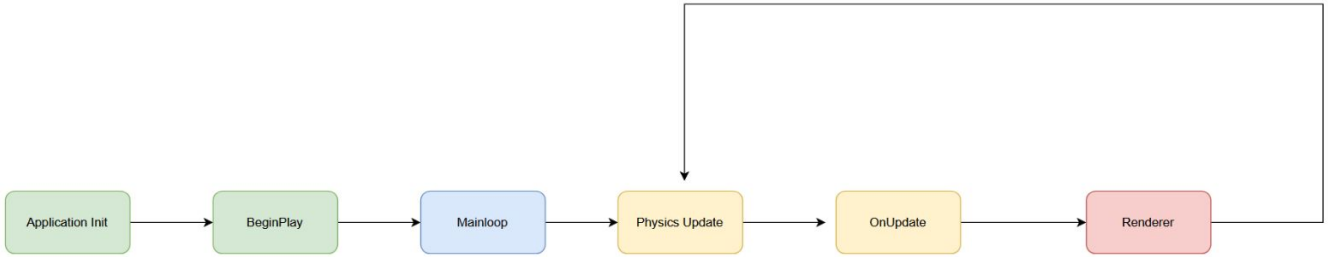
Architectural Designs in TheAtlasEngine

Before we do.....

Game mainloop

- For those coming from a non-game development or game engine developer background.
- In game engines, their typically are a variety of ways they handle state in their own respective mainloops.
- This also includes how game object updates as well
- As shown in this image, this is what a game mainloop at the highest level of importance works.

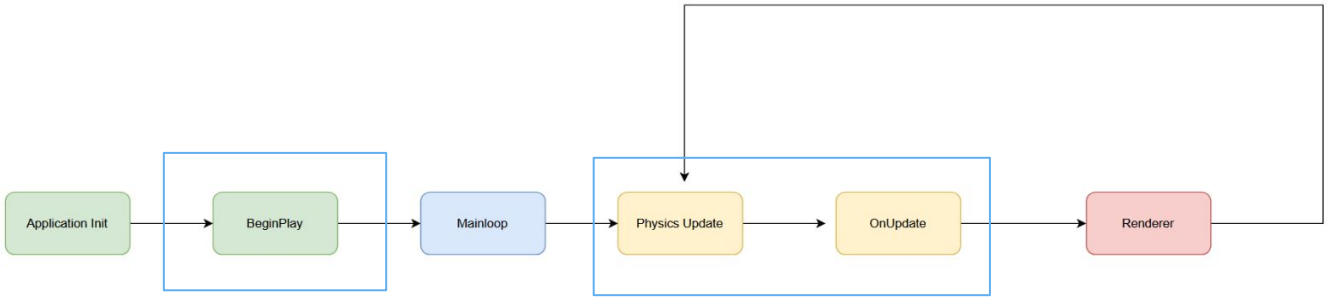
Game mainloop



Why is update frequency important?

- Why does update frequency matter?
  - Separation between non-physics and physics-based logic.
  - Ensures determinism when running these logics in the mainloop.
  - Can be used to perform optimizations if the game engine needs to. Whether that is running on a different thread.
- Importance comes from ensuring that variety of specific game logic are separated in cases where some logic are computationally expensive.

Why is update frequency important?



Rethinking Game Objects in TheAtlasEngine

What were our options?

Exhibit A (diagram of simple yet complex diagrams demo)

Exhibit B (diagram of complex practical diagrams demo)

Exhibit C (exaggerated yet complex visual + diagram demo)

NOTE TO SELF:

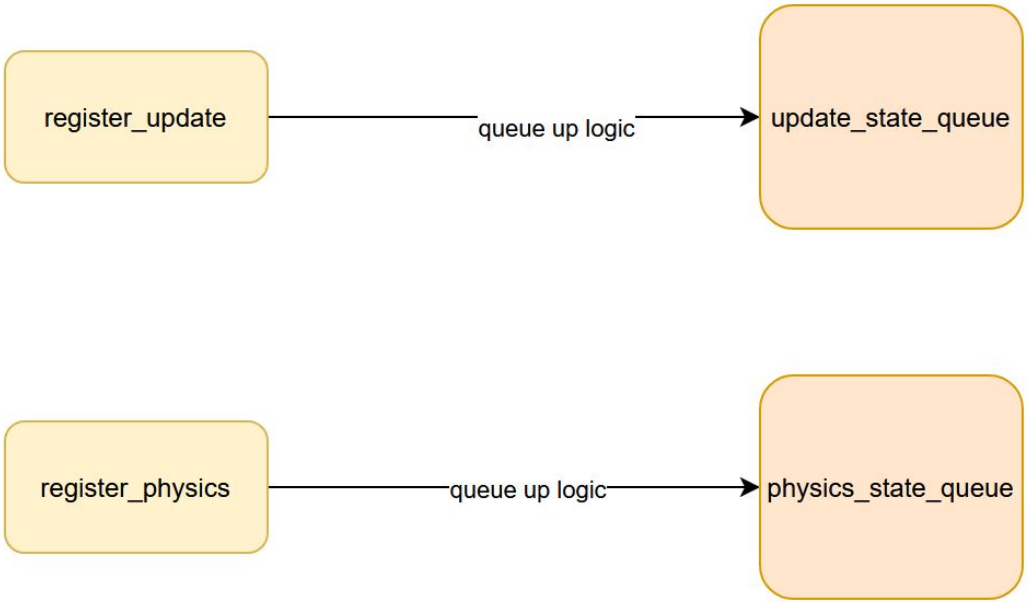
- Simple example: Show just a wooden chair that is a static game object
- Complex example: Show wooden chair, that has a physics body. That also has two hands that have inverse kinematics applied + they shoot a gun, and can turn into cloth.
- Then model the inheritance structure of how this looks on these two occasions.

Game objects in TheAtlasEngine

Registration Callback for Game Object State

- In TheAtlasEngine, considering that there were many options. Which included using inheritance, mixture of inheritance and an entity component system.
- My consideration when it came to game objects were a few things.
  - How do we want to handle state?
  - How can we tell TheAtlasEngine which object its associated with? (the game object)
  - Without using inheritance, how can we provide an API that gives users the same level of control where their logic gets invoked by TheAtlasEngine.
  - While still maintaining some level of simplicity to the ways these states get handled.
- That is when I thought back at what I had for Engine3D previously.
  - Which was me implementing my very own version of std::bind for the event system.
  - Purpose was to bind an event that dispatch a callback associated with the event.
  - This brought up the idea of using a callback system to register state of the game objects.
- Things that I wanted to look into once I had this in mind were:
  - How can users register state?
  - How will TheAtlasEngine know and associate where this states coming from? Would it even matter?
  - Trimming down things of what you wanted TheAtlasEngine to know when it came to game object state.

Registration Callback for Game Object State



Registration Callbacks Invocation Flow

- How this works in TheAtlasEngine for game object state?
  - Whenever you want your state of the game object to be known. You would call one of our register functions that we provide to you.
  - register_physics to tell TheAtlasEngine to invoke this object's state during the physics timestep interval.
  - Register_update tells TheAtlasEngine to invoke the object's state during the game state to run in a non-physics based game logic time interval.
- The flow is whenever these callbacks get submitted. They get sent to our state queues. Which are just std::deque's.
- That then get ran at specific update frequency in the application's mainloop.
- Preface: Now, I know I have mentioned game objects are not represented in an object-oriented fashion. I do want to preface by saying. I do not think inheritance doesn't have a place in the project. I think game objects in itself are highly complex to model using inheritance. As in TheAtlasEngine, we use inheritance when involving scenes. To create an object you need to have a scene to associate the object's lifetime to.

Wait?! Then how are game objects created?

Inheriting when involving scenes

- To create a custom scene in TheAtlasEngine, you must inherit from the atlas::scene_scope class.
- This scene_scope class provides you functionality to create a scene object and associate its lifetime to the scene class.
- This is a requirement because no game objects should not be created without an associated scene.
- This is to ensure the lifetime of game objects are based on the lifetime of the scene itself.

Creating objects required associated with a scene

- To create a game object in a scene you inherit from the scene_scope class.
- Then are provided with functions such as create_object(const std::string& p_name).
- This function when you call, that is provided to you return a type called strong_ptr.
- The use of type strong_ptr is to ensure that any game object that you create must be a valid game object to prevent lifetime issues or invalidated objects.
- Minimizing the potential of having null objects.

Game object returns type called strong_ptr

- Yes, game objects are created as the following, strong_ptr<scene_object>
- Strong_ptr is to prevent lifetime doubts in our code.
- Whenever you are receiving a strong_ptr. This should always point you to a valid object.
- Minimize state checking if a pointer points to a null object.

What is a strong_ptr?

- Strong_ptr allows you to remove checks of having a null pointer.
- It is to ensures that the object is always alive when being pointed to.
- Uses polymorphic allocator to handle memory allocations instead of using the global memory allocator of new/delete. Preventing issues such as memory fragmentation.
- If you need to have an absent value for the case that an object needs to live through an entire scope of a class. There is a type called optional_ptr<T>.
- optional_ptr<T> works well with strong_ptr. This does not expect an immediate value, and if you try to access this pointer. You will receive a throw to indicate bad memory access violation.
- Trying to debug an invalid pointer access is much difficult to debug considering even using a debugger. Whereas if you just throw that an error happened when the error happens. Then you will most likely catch the error and where it is happening much earlier then later on.

Why not use shared_ptr?

- Now, you are probably wondering. Why not use shared_ptr?
- Reduce custom allocator support.
- What does that mean?
  - It means the amount of work you need to bend shared_ptr to work with your custom allocator can take up quite a lot of time.
  - Even then, it does not work quite well if you decide you wanted to use pmr::polymorphic_allocator.
  - You would have to make a custom allocator for majority of any type that you potentially might want to do memory allocations with. (TBD. Need to ask about this)
- Shared_ptr can be nullify which is what strong_ptr guarantees the object wont be null. Whereas shared_ptr can be null.
- Preface: shared_ptr is fine if you want to use it. I want to ensure memory safety that the object being pointed to is always valid.

What does this mean for TheAtlasEngine?

- This means that game objects are always going to be valid throughout the duration of the scene they are associated with.
- Utilizing polymorphic allocator for any sort of memory allocations. Minimize the use of global allocators of new/delete within the codebase. Minimize potential memory unsafe situations that can lead to memory fragmentation.
- Reduces amount of lifetime violations. In the sense, that everytime your parameters are of strong_ptr<T>. You can be guaranteed that object is always valid when you see it.

Ensuring memory safety with strong_ptr

Modern C++ Features

span<T>

Without using span<T>

- Increase code boilerplate.
- Less flexible with contiguous sequence buffers such as arrays, c-style arrays, and vector<T>.
- Bug-prone if you have multiple buffers with different lengths that needs to be handled. Leading to out of bounds accesses.

Without using span<T>

```
template<size_t arr_size>
class index_buffer32_without {
public:
    index_buffer32_without() = default;

    // boilerplate without std::span<uint32_t>
    index_buffer32_without(const std::vector<uint32_t>& p_indices) {}
    index_buffer32_without(const uint32_t p_arr[], size_t p_size) {}
    index_buffer32_without(uint32_t* p_arr, size_t p_size) {}
    index_buffer32_without(std::array<uint32_t, arr_size> p_array) {}

    void bind(const VkCommandBuffer& p_current);

    void destroy();
};
```

With use of span

- Reduce code boilerplate.
- Flexible and compatible with any contiguous container in the C++ standard.
- Reduce bugs when dealing with multiple contiguous buffers that may have vary lengths.
- Centralized contiguous that expresses what sequence of data are your inputs and outputs.

With use of span

```
class index_buffer32 {
public:
    index_buffer32() = default;

    // With std::span<uint32_t>
    index_buffer32(const std::span<uint32_t>& p_indices) {}

    void bind(const VkCommandBuffer& p_current);

    void destroy();
};
```

Using std::filesystem

- Handles cross-platform filepath
- Provides logic to help with handling absolute and relative paths to your current working directory.
- Minimize reinventing the wheel and cost of maintainability within our own codebase to not needing to reimplement these features std::filesystem provides.

Using std::filesystem

```
if (sphere_filepath != "") {
    std::filesystem::path relative_path =
      std::filesystem::relative(Path: sphere_filepath, Base: "./");
    console_log_trace(fmt: "Sphere Filepath = {}", sphere_filepath);
    viking_room_material->model_path = { relative_path.string() };
```

Shout out to std::println

- The last thing I want to mention and shout out to is finally being able to use print/println being in the standard.
- Ive been using std::print/std::println. They both have been quite amazing and simplify whenever I do debugging in smaller projects.
- I have not needed to touch iostream as ive been using print/println as my replacement of std::cout.

FAQ

Thank you for attending!

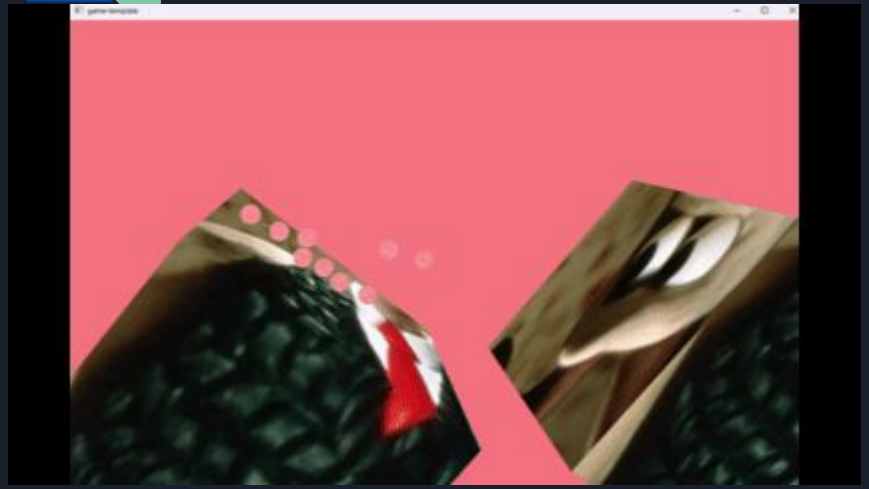
Below are hidden slides....

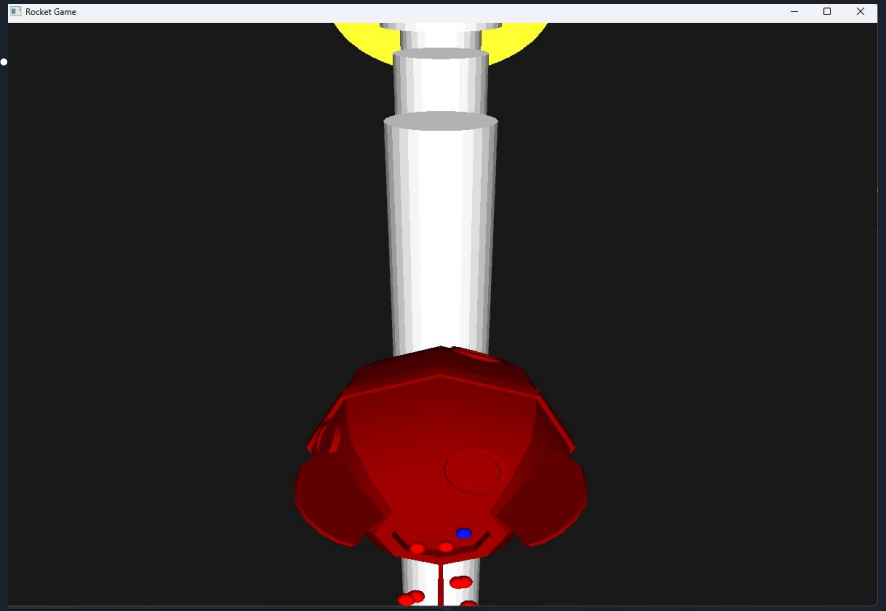
Motivation for handling game object state

- Game object states are complex to model using inheritance to express a relationship with their state
- Increase complexities in users code to define game objects.
- Difficult to model every possible known factor of how a game object can be represented, with inheritance.
- Simple API for specifying when object state gets invoked by TheAtlasEngine.
- Examples
    - Two diagrams showing traditionally designed game objects using inheritance
    - This is a bit exaggeration. So follow with me on this example, you can have a game object that can be as simple as a static chair to a complex game object that is not only a chair. BUT a chair + physics body + rigidbody + collider + flammable + have hands that can shoot a gun.

How does TheAtlasEngine update game objects?

TheAtlasEngine Now.





What I learned?

Moving to TheAtlasEngine

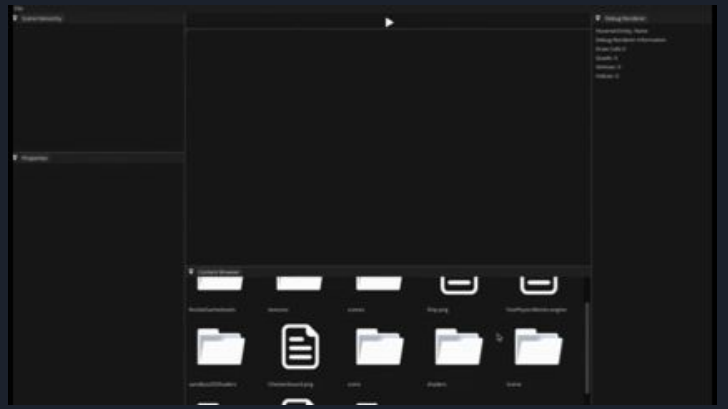
- Changing from Engine3D to TheAtlasEngine.
- What changed and why?

What were architecture designs in TheAtlasEngine?

Game Object States

Let’s go over game main loops.

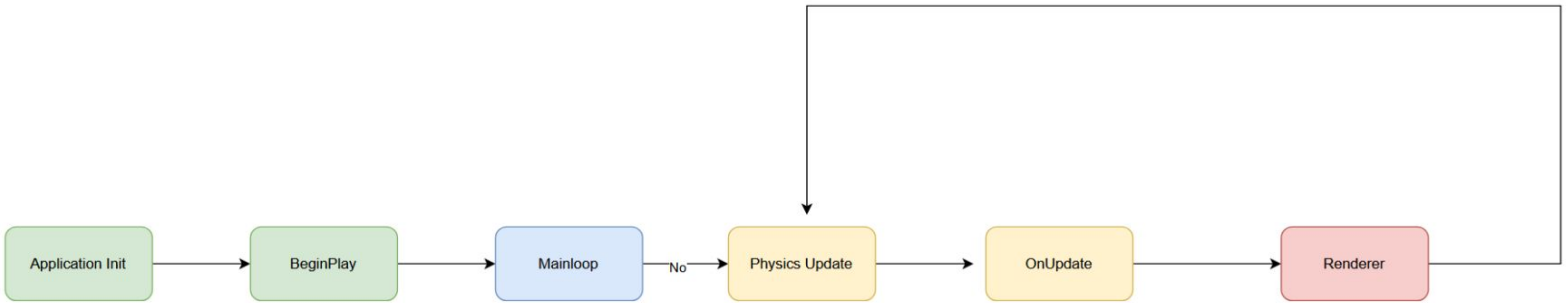
3 years of game engine development



The Engine3D Project

Entering Open-Source

Mainloop


Application Init
BeginPlay
Mainloop
No
Physics Update
OnUpdate
Renderer

Game Object Design

Exhibit A

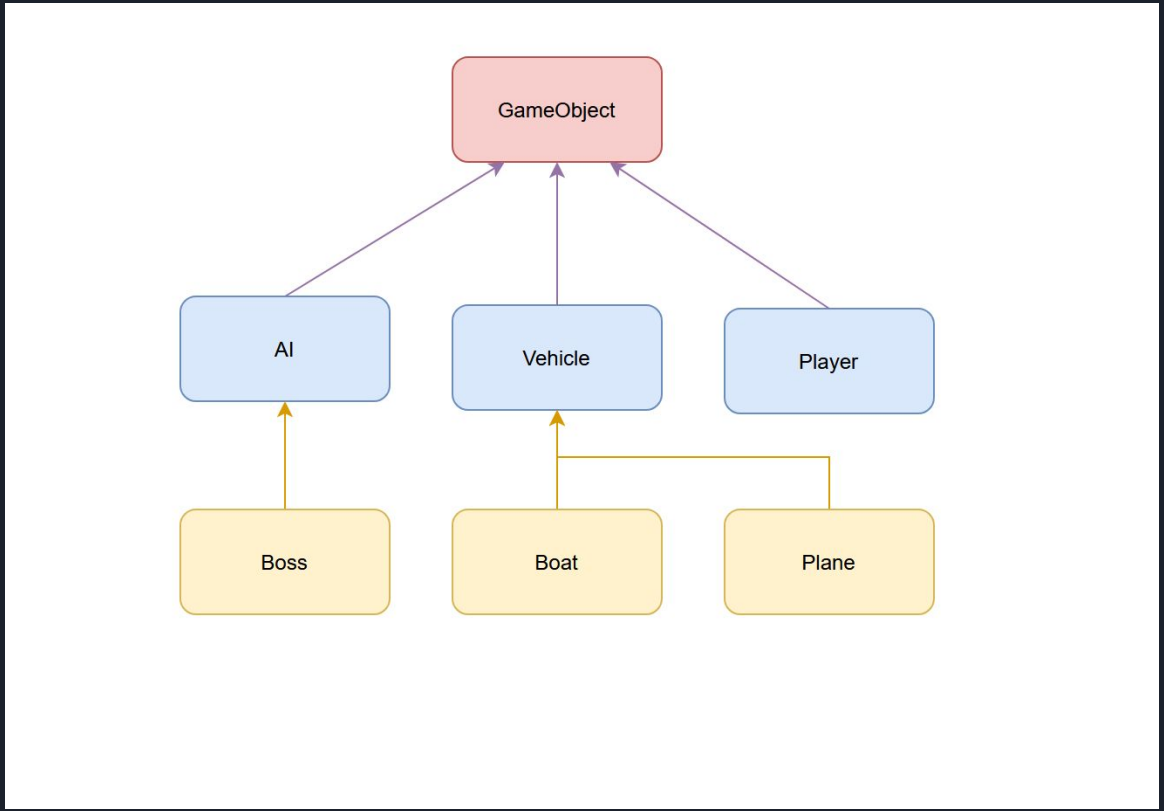
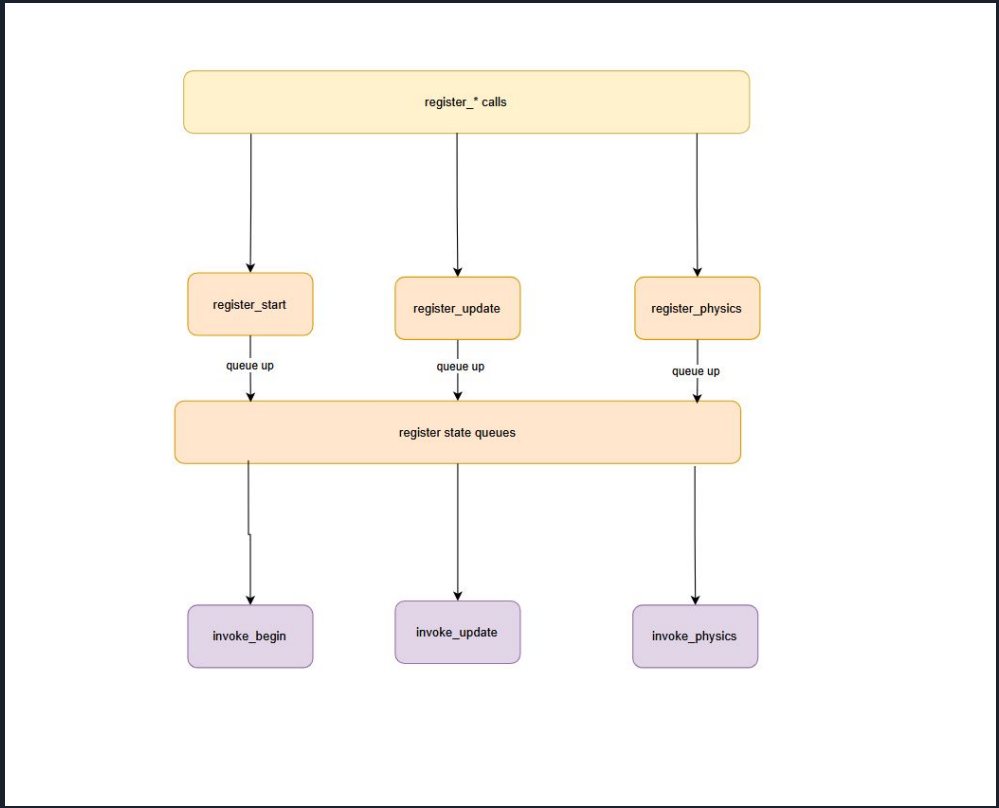


Exhibit B – [add another example of diagram]

How TheAtlasEngine update game objects?



Game Object State

Register those updates?!

```
    atlas::register_update(p_instance: this, p_callable: &main_scene::on_update);
    atlas::register_start(p_instance: this, p_callable: &main_scene::start_game);
    atlas::register_physics(p_instance: this, p_callable: &main_scene::on_physics_update);
    atlas::register_ui(p_instance: this, p_callable: &main_scene::on_ui_update);
}
```

Game Object Creation

Game objects return strong_ptr

```
class scene_scope {
public:
    scene_scope() = default;
    scene_scope(const std::string& p_tag)
      : m_tag(p_tag) {}

    strong_ref<scene_object> create_object(const std::string& p_tag) {
        return create_strong_ref<scene_object>(
          p_allocator: m_object_allocator, &m_registry, p_tag);
    }
```

Why not use shared_ptr?

What shared_ptr doesn’t guarantee

- Has ability to be nullptr/doesn’t always guarantee alive
- Custom allocator incompatibility
-

[add code snippet for writing a custom allocator \w shared]

[add example for memory-safety gurantees

[mentions of issues \w custom allocator support + code]

[mention strong_ptr guarantees

[add] Diagram game object state (traditional)

[add] diagram for registration callback system

[add] diagram of Atlas mainloop + registration

[add] minimal benefit-cost analysis (preface still undergoing current benchmarks)

Embracing Modern C++

Reasons to use span<T>

- Bug-prone to passing larger sizes of data, leading to buffer overrun.
- Verbosity/Boilerplate
- No type-safety or expressiveness
- Incompatible with different containers

examples without using span

```
template<size_t arr_size>
class index_buffer32_without {
public:
    index_buffer32_without() = default;

    // boilerplate without std::span<uint32_t>
    index_buffer32_without(const std::vector<uint32_t>& p_indices) {}
    index_buffer32_without(const uint32_t p_arr[], size_t p_size) {}
    index_buffer32_without(uint32_t* p_arr, size_t p_size) {}
    index_buffer32_without(std::array<uint32_t, arr_size> p_array) {}

    void bind(const VkCommandBuffer& p_current);

    void destroy();
};
```

with the use of span<T>

```
class index_buffer32 {
public:
    index_buffer32() = default;

    // With std::span<uint32_t>
    index_buffer32(const std::span<uint32_t>& p_indices) {}

    void bind(const VkCommandBuffer& p_current);

    void destroy();
};
```

```
/**
 * @brief Maps buffer handler to chunk of data of type, that is
 * std::span<uint32_t>.
 */
void write(const VkDevice& p_device, const buffer_handle& p_buffer,
           const std::span<uint32_t>& p_in_buffer);

/**
```

Problems without using std::span<T>

- Bug-prone to passing larger sizes of data, leading to buffer overrun.
- Verbosity/Boilerplate
- No type-safety or expressiveness
- Incompatible with different containers

```
std::vector<int> v = {1, 2, 3};
    process_data(v.data(), v.size()); // For vector

    int arr[] = {4, 5, 6};
    process_data(arr, 3); // For C-style array
```

Problems Before span<T>

- Bug-prone to passing larger sizes of data, leading to buffer overrun.
- Verbosity/Boilerplate
- No type-safety or expressiveness
- Incompatible with different containers

[add] code snippet span<T> address those problems

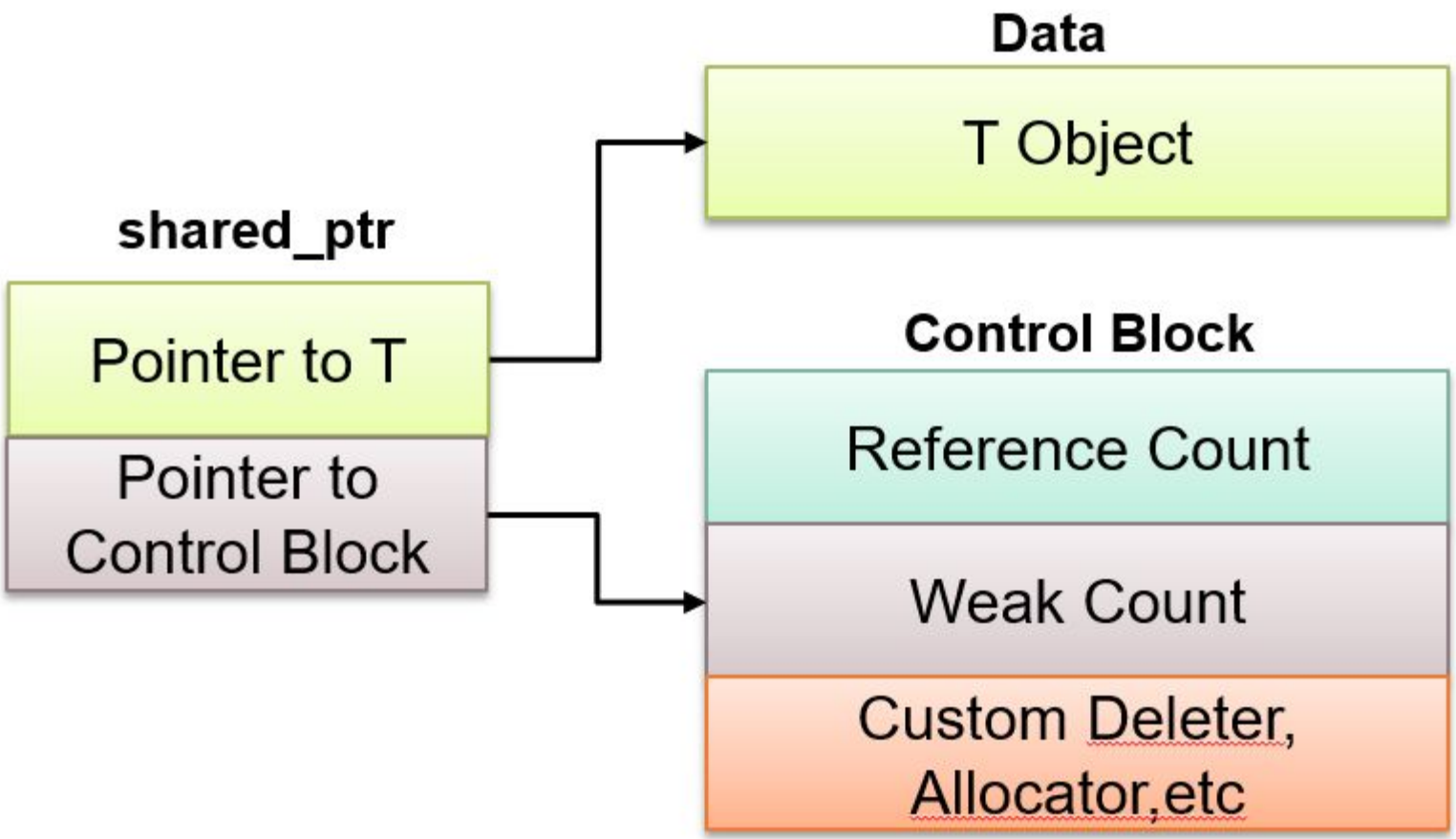
```
void process_data(std::span<const int> data); // Accepts any contiguous range
```

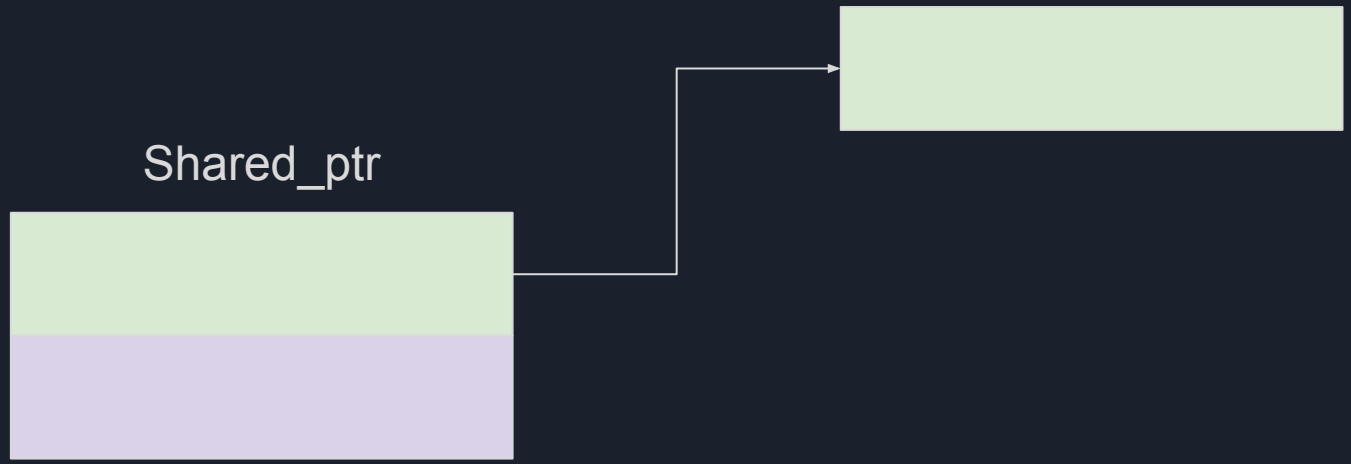
[add] code snippet more span usage

Pointers in TheAtlasEngine

- Game objects must be created with an associated scene.
- Game objects return what we call a strong_ptr and if you have an absent value then we use an optional_ptr.
- Now, you're probably wondering why we didn't use shared_ptr/unique_ptr?
-

Why game objects not return shared_ptr?


Data
T Object
shared_ptr
Pointer to T
Pointer to Control Block
Control Block
Reference Count
Weak Count
Custom Deleter, Allocator,etc


Shared_ptr

[show example of creating a custom allocator with shared_ptr

strong_ptr, optional_ptr + pmr::polymorphic_allocator

How we handle pointers in TheAtlasEngine?

[game objects return strong_ptr]

[add images to show how we create game objects]

[show what the game object return – strong_ptr]

[what is a strong_ptr?]

[why I chose strong_ptr over shared_ptr/unique_ptr?]

- Pass polymorphic allocator to ensure custom allocator support
- Remove doubt of a pointer being invalid (other words nullified)

[add show what the game object pointer returns]

[add diagram to show why we didn't use shared_ptr or unique_ptr]

Problems strong_ptr addresses

What is strong_ptr and optional_ptr?

- Not-nullified
- Does not allow you to access invalid objects. Will THROW
- Requires to pass in the pmr::polymorphic_allocator instead of using global allocator (new/delete)

filesystem::path

[add] filesystem::path::filename code snippet

[add] filesystem::path::extension code snippet

[add] filesystem::path::parent_path code snippet