FunctionRunner

A lightweight, stack-based utility for sequential function execution with error tracking

What is FunctionRunner?

- **Type-safe** function sequencer with compile-time size
- Executes functions **sequentially** until one fails
- **Zero heap allocation** - uses fixed-size array
- Tracks **failure state** and provides diagnostics
- Clean API with **automatic template deduction**

Basic Usage

```cpp
#include "function_runner.hpp"

auto runner = make_function_runner(
    step([]() {
        std::cout << "Initializing...\n";
        return true;
    }, "Initialization failed"),

    step([]() {
        std::cout << "Connecting...\n";
        return false;  // Simulates failure
    }, "Connection failed"),

    step([]() {
        std::cout << "Starting...\n";
        return true;
    }, "Startup failed")
);

int failed_idx = runner.run();  // Returns index of failed step
```

Key Features

1. No Template Parameter Required

```
// ❌ Old way: explicit template parameter
FunctionRunner<3> runner{{...}};

// ✅ New way: automatic deduction
auto runner = make_function_runner(
    step(fn1, "Error 1"),
    step(fn2, "Error 2"),
    step(fn3, "Error 3")
);
```

Key Features

2. Clean `step()` Helper

```
auto runner = make_function_runner(
    step(initialize_system, "Init failed"),
    step(connect_database, "DB connection failed"),
    step(load_config, "Config load failed")
);
```

No need for verbose `std::pair{}` or custom wrappers!

Key Features

3. State Tracking

```
int result = runner.run();

if (result >= 0) {
    // Query which step failed
    int failed = runner.failed_step();

    // Get error message
    auto msg = runner.error_message(failed);
    std::cout << "Failed: " << msg << "\n";

    // Retry the failed step
    if (runner.rerun(failed)) {
        std::cout << "Retry succeeded!\n";
    }
}
```

API Overview

Execution

- `int run()` - Execute all steps, returns `-1` or failed index

Queries

- `int failed_step()` - Get index of failed step
- `size_t size()` - Get total number of steps
- `string_view error_message(size_t idx)` - Get error by index
- `string_view error_message(function*)` - Get error by pointer

Retry

- `bool rerun(size_t idx)` - Retry step by index
- `bool rerun(function*)` - Retry step by pointer

Implementation Details

```
template<std::size_t N>
class FunctionRunner {
    struct Step {
        std::function<bool()> func;
        std::string_view error_msg;
    };

    Step m_steps[N];                // Fixed-size array
    mutable int m_failed_step = -1;  // Tracks failure
};
```

- **Zero dynamic allocation**
- **Type-erased** via `std::function<bool()>`
- **Lightweight** error messages via `std::string_view`

Use Cases

System Initialization

```
auto startup = make_function_runner(
    step(init_logging, "Logging init failed"),
    step(init_network, "Network init failed"),
    step(init_database, "Database init failed"),
    step(load_plugins, "Plugin load failed")
);
```

Validation Pipelines

```
auto validator = make_function_runner(
    step([]() { return validate_input(); }, "Invalid input"),
    step([]() { return check_permissions(); }, "Access denied"),
    step([]() { return verify_resources(); }, "Resources unavailable")
);
```

Advanced: Retry Logic

```
auto runner = make_function_runner(
    step(connect_to_server, "Connection failed")
);

int result = runner.run();

if (result >= 0) {
    // Retry with exponential backoff
    for (int i = 0; i < 3; i++) {
        std::this_thread::sleep_for(std::chrono::seconds(1 << i));
        if (runner.rerun(result)) {
            std::cout << "Retry successful!\n";
            break;
        }
    }
}
```

Design Principles

1. **Stack-based** - No dynamic allocation overhead
2. **Type-safe** - Compile-time size checking
3. **Modern C++17** - Uses `string_view`, template deduction
4. **Clean API** - Minimal boilerplate via `step()` helper
5. **Flexible** - Both index and pointer-based queries
6. **Namespace hygiene** - Internal helpers in private namespace

C++17 Features Used

1. Class Template Argument Deduction (CTAD)

```
// Before C++17: Need to specify types
std::pair<Lambda1, const char*>{lambda, "error"};

// C++17: Types deduced automatically
std::pair{lambda, "error"};
```

Our `step()` helper leverages CTAD:

```
template<typename Func>
StepWrapper<Func> step(Func&& f, std::string_view msg) {
    return StepWrapper<Func>{std::forward<Func>(f), msg};
}
```

C++17 Features Used

2. std::string_view

```
struct Step {
    std::function<bool()> func;
    std::string_view error_msg;  // Non-owning, lightweight
};
```

Benefits:

- No string copying or allocation
- Works with string literals, `std::string`, and char arrays
- Cheap to pass and store (just pointer + size)

C++17 Features Used

3. Variadic Template Expansion

```
template<typename... Funcs>
auto make_function_runner(StepWrapper<Funcs>... steps) {
    constexpr std::size_t N = sizeof...(Funcs);
    return FunctionRunner<N>{{
        typename FunctionRunner<N>::Step{
            steps.func,
            steps.error_msg
        }...  // Pack expansion
    }};
}
```

Automatically deduces `N` from number of arguments!

C++17 Features Used

4. Fold Expressions (Implicit)

```
// sizeof...(Funcs) counts parameter pack elements at compile-time
constexpr std::size_t N = sizeof...(Funcs);
```

Enables automatic array size without manual counting.

5. constexpr if (Future Enhancement)

```
// Could be used for compile-time optimization
if constexpr (N == 1) {
    // Single-step fast path
} else {
    // Multi-step path
}
```

C++17 Features Used

6. Structured Bindings (User Side)

```
auto runner = make_function_runner(
    step(fn1, "error1"),
    step(fn2, "error2")
);

// C++17 structured bindings in user code
if (auto [success, idx] = std::pair{runner.run() == -1,
                                    runner.failed_step()};
    !success) {
    std::cout << "Failed at: " << idx << "\n";
}
```

C++17 Features Used

7. Perfect Forwarding with auto

```
template<typename Func>
auto step(Func&& f, std::string_view msg) {
    //         ^^ Universal reference
    return StepWrapper<Func>{std::forward<Func>(f), msg};
    //                        ^^^^^^^^^^^^^^^^^^^ Perfect forwarding
}
```

Preserves value category (lvalue/rvalue) without template bloat.

C++17 vs Older Standards

Feature	Pre-C++17	C++17
Template deduction	`make_function_runner<Fn1, Fn2>(...)`	`make_function_runner(...)`
String view	`const char*` + length	`std::string_view`
Pack size	Manual counting	`sizeof...(pack)`
Type deduction	Explicit helpers	CTAD + auto
Aggregate init	Limited	Enhanced with deduction

Summary

FunctionRunner provides:

- ✅ Sequential execution with early exit on failure
- ✅ Automatic template argument deduction
- ✅ Built-in error tracking and diagnostics
- ✅ Retry capability for failed steps
- ✅ Zero heap allocation
- ✅ Clean, modern C++17 API

Perfect for initialization sequences, validation pipelines, and multi-step workflows!

Thank You!

Questions?

Repository: https://github.com/Anluren/stack_vec