Function Runners

Lightweight, stack-based utilities for function execution with error tracking

FunctionRunner - Sequential execution with early exit

ParallelRunner - Execute all and collect results

What are Function Runners?

- **Type-safe** function sequencers with compile-time size
- **Zero heap allocation** - uses fixed-size arrays
- Tracks **failure state** and provides diagnostics
- Clean API with **automatic template deduction**
- Two variants for different use cases

FunctionRunner: Sequential Execution

Executes functions **sequentially** until one fails (early exit)

```
#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;  // Stops here!
    }, "Connection failed"),

    step([]() {
        std::cout << "Starting...\n";  // Never runs
        return true;
    }, "Startup failed")
);
```

ParallelRunner: Collect All Results

Executes **all** functions regardless of failures

```
#include "parallel_runner.hpp"

auto runner = make_parallel_runner(
    parallel_step([]() {
        std::cout << "Check 1...\n";
        return true;
    }, "Check 1 failed"),

    parallel_step([]() {
        std::cout << "Check 2...\n";
        return false;  // Keeps going!
    }, "Check 2 failed"),

    parallel_step([]() {
        std::cout << "Check 3...\n";  // Still runs!
        return true;
    }, "Check 3 failed")
);

runner.run();  // Executes all three
```

Choosing Between Them

Use FunctionRunner when:

- Steps are **dependent** (later steps need earlier ones)
- Want to **stop immediately** on first failure
- Example: System initialization, deployment pipelines

Use ParallelRunner when:

- Steps are **independent**
- Need to see **all failures**, not just first
- Example: Validation checks, health monitoring, batch diagnostics

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";
    }
}
```

ParallelRunner: API Overview

Execution

- `void run()` - Execute all steps, stores results

Queries

- `bool result(size_t idx)` - Get individual step result
- `const array<bool, N>& results()` - Get all results
- `bool all_succeeded()` - Check if all passed
- `bool any_succeeded()` - Check if any passed
- `size_t success_count()` - Count successes
- `size_t failure_count()` - Count failures

Retry

FunctionRunner: 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

ParallelRunner: Batch Retry

```
auto health_checks = make_parallel_runner(
    parallel_step(check_disk, "Disk check failed"),
    parallel_step(check_network, "Network check failed"),
    parallel_step(check_memory, "Memory check failed")
);

health_checks.run();

if (!health_checks.all_succeeded()) {
    std::cout << "Some checks failed, retrying...\n";
    size_t recovered = health_checks.rerun_failed();
    std::cout << "Recovered " << recovered << " checks\n";

    if (!health_checks.all_succeeded()) {
        std::cout << "Failed checks:\n";
        for (size_t i = 0; i < health_checks.size(); ++i) {
            if (!health_checks.result(i)) {
                std::cout << "  - " << health_checks.error_message(i) << "\n";
            }
        }
    }
}
```

Implementation: FunctionRunner

```
template<typename... Funcs>
class FunctionRunner {
    std::tuple<std::pair<Funcs, std::string_view>...> m_steps;
    mutable int m_failed_step = -1;
};
```

- **Zero dynamic allocation** - everything on stack
- **No type erasure** - each callable stored with actual type
- **Lightweight** error messages via `std::string_view`
- Uses `std::tuple` to preserve individual types

Implementation: ParallelRunner

```
template<typename... Funcs>
class ParallelRunner {
    std::tuple<std::pair<Funcs, std::string_view>...> m_steps;
    mutable std::array<bool, sizeof...(Funcs)> m_results{};
    mutable bool m_executed = false;
};
```

- Same template-based design as FunctionRunner
- Additional result storage for all steps
- Both runners share similar API patterns
- Compile-time size determination

Performance Optimization: Replacing std::function

The Problem with std::function

```
// Old implementation (with std::function)
template<std::size_t N>
class FunctionRunner {
    struct Step {
        std::function<bool()> func;  // ❌ Type erasure overhead
        std::string_view error_msg;
    };
    Step m_steps[N];
};
```

Costs:

- Type erasure via virtual dispatch
- Heap allocation for non-trivial captures
- Copy overhead when constructing

Performance Optimization: Template-Based Solution

New Implementation

```
// New implementation (template-based)
template<typename... Funcs>
class FunctionRunner {
    std::tuple<std::pair<Funcs, std::string_view>...> m_steps;
};
```

Benefits:

- ✅ Each lambda stored with **actual type**
- ✅ **Zero heap allocations**
- ✅ Compiler can **inline** function calls
- ✅ Move semantics - no copying
- ✅ Smaller binary (no vtables)

Performance Comparison

Memory Layout

Old (std::function):

```
Step 0: [std::function (24-32 bytes)] [string_view (16 bytes)]
        ↓ (potential heap allocation)
        [Lambda object on heap]
```

New (template-based):

```
Step 0: [Lambda inline] [string_view (16 bytes)]
        (everything on stack, no indirection)
```

Result: Smaller, faster, cache-friendly!

Benchmark Results

```
Test: 5 sequential steps, all succeed
  Old (std::function): ~95 ns/iteration
  New (template-based): ~56 ns/iteration

Improvement: 41% faster! ⚡
```

Why?

- No virtual dispatch
- No heap allocation
- Better inlining
- Improved cache locality

Code Size Impact

Before (with std::function):

- Uses type erasure infrastructure
- Requires vtables for polymorphism
- Generic implementation shared

After (template-based):

- Generates specialized code per type combination
- No vtable overhead
- Compiler optimizes each instantiation

Trade-off: Slightly larger binary for heavily templated code, but much faster execution

Zero Overhead Abstraction

```
// Both approaches have identical syntax:
auto runner = make_function_runner(
    step([]() { return true; }, "Step 1 failed"),
    step([]() { return false; }, "Step 2 failed")
);
```

Old: Creates `std::function` wrappers

New: Stores actual lambda types

User code unchanged - optimization is transparent!

How It Works: Template Magic

```
template<typename... Funcs>
auto make_function_runner(StepWrapper<Funcs>&&... steps) {
    // Funcs = [Lambda1, Lambda2, Lambda3]
    // Each has unique type preserved!

    return FunctionRunner<Funcs...>{
        std::tuple<std::pair<Funcs, std::string_view>...>{...}
    };
}
```

Key: Variadic templates expand to actual types, not type-erased wrappers

Using std::bind and Lambdas

Both work perfectly with zero overhead:

```
// std::bind
auto r1 = make_function_runner(
    step(std::bind(connect, "localhost", 8080), "Failed")
);

// Lambda with capture
auto r2 = make_function_runner(
    step([&](){ return connect(host, port); }, "Failed")
);
```

Each gets its own type preserved:

- `std::bind` result type stored directly
- Lambda type stored directly
- No conversion to `std::function`

Use Cases

FunctionRunner: 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")
);
// Stops at first failure
```

ParallelRunner: Health Checks

```
auto health = make_parallel_runner(
    parallel_step(check_disk, "Disk check failed"),
    parallel_step(check_network, "Network check failed"),
    parallel_step(check_memory, "Memory check failed")
);
// Runs all checks, collects all results
```

Use Cases

FunctionRunner: 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")
);
// Early exit on first validation failure
```

ParallelRunner: Feature Compatibility

```
auto compat = make_parallel_runner(
    parallel_step(check_webgl, "WebGL not supported"),
    parallel_step(check_local_storage, "LocalStorage unavailable"),
    parallel_step(check_web_workers, "Web Workers not supported")
);
// Check all features, show comprehensive report
```

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

Sequential execution with early exit

- ✅ Stops at first failure
- ✅ Perfect for dependent operations
- ✅ Minimal overhead when stopping early
- ✅ Automatic template argument deduction
- ✅ Built-in error tracking and diagnostics
- ✅ Zero heap allocation
- ✅ Clean, modern C++17 API

Use for: Initialization sequences, deployment pipelines, dependent validation

Summary: ParallelRunner

Execute all and collect results

- ✅ Runs all operations regardless of individual failures
- ✅ Comprehensive result collection and analysis
- ✅ Batch retry of failed operations
- ✅ Automatic template argument deduction
- ✅ Built-in error tracking and diagnostics
- ✅ Zero heap allocation
- ✅ Clean, modern C++17 API

Use for: Health checks, independent validations, feature detection

Thank You!

Questions?

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