"IOC_Config Documentation"

"Michele Bigi"

IOC_Config - Implementation Guide for Projects

Version: 1.2.0
Date: December 2, 2025
Target Audience: Developers integrating IOC_Config

Table of Contents

Integration Overview

IOC_Config is designed as a reusable library for configuration management. It can be integrated into any C++17 project through:

1. **Source Integration**: Include source directly in project
2. **CMake Integration**: Link as CMake target
3. **System Library**: Install system-wide and link
4. **Package Manager**: Integration via conan/vcpkg

Prerequisites

- **C++ Standard**: C++17 or later
- **Compiler**: GCC 7+, Clang 5+, MSVC 2017+
- **Dependencies**: nlohmann/json (header-only)
- **Optional**: YAML support (yaml-cpp), TOML support (toml++)

Setup & Build

Option 1: Direct Source Integration

```
# 1. Copy IOC_Config source to your project
cp -r IOC_Config/include/ioc_config YourProject/thirdparty/
cp -r IOC_Config/src/oop_parser.cpp YourProject/src/

# 2. Include in your CMakeLists.txt
add_library(ioc_config src/oop_parser.cpp)
target_include_directories(ioc_config PUBLIC include/)
target_compile_features(ioc_config PUBLIC cxx_std_17)

# 3. Link to your target
target_link_libraries(your_target PRIVATE ioc_config)
```

Option 2: Git Submodule

```
# 1. Add as submodule
git submodule add https://github.com/manvalan/IOC_Config vendor/
        IOC_Config

# 2. In CMakeLists.txt
add_subdirectory(vendor/IOC_Config)
target_link_libraries(your_target PRIVATE ioc_config)
```

Option 3: System Installation

```
# 1. Build and install IOC_Config
cd IOC_Config
mkdir build && cd build
cmake ..
cmake --build . --config Release
sudo cmake --install . --prefix /usr/local

# 2. In your CMakeLists.txt
find_package(ioc_config REQUIRED)
target_link_libraries(your_target PRIVATE ioc_config::ioc_config)
```

Build Configuration

```
# CMakeLists.txt example
cmake_minimum_required(VERSION 3.10)
project(MyProject)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# IOC_Config
add_subdirectory(vendor/IOC_Config)

# Your executable
add_executable(my_app src/main.cpp)
target_link_libraries(my_app PRIVATE ioc_config)
```

Optional Features

```
# Enable YAML support
set(WITH_YAML ON)

# Enable TOML support
set(WITH_TOML ON)

# Add to project
add_subdirectory(vendor/IOC_Config)
```

Basic Integration

Step 1: Include Header

```
#include <ioc_config/oop_parser.h>

using namespace ioc_config;
```

Step 2: Initialize Parser

```
OopParser parser;
```

Step 3: Load Configuration

```
if (!parser.loadFromOop("config.oop")) {
    std::cerr << "Failed to load config: " <<
        parser.getLastError() << std::endl;
    return false;
}
```

Step 4: Access Data

```
// Get parameter
std::string value = parser.getParameter("section", "parameter",
        "default");

// Check existence
if (parser.parameterExists("section", "parameter")) {
    // Process...
}

// Get all sections
auto sections = parser.getSectionNames();
for (const auto& section : sections) {
    auto keys = parser.getParameterKeys(section);
    for (const auto& key : keys) {
        std::string val = parser.getParameter(section, key);
    }
}
```

Step 5: Complete Example

```
#include <ioc_config/oop_parser.h>
#include <iostream>

using namespace ioc_config;

int main() {
    OopParser parser;

    // Load configuration
    if (!parser.loadFromOop("orbital_data.oop")) {
        std::cerr << "Error: " << parser.getLastError() <<
         std::endl;
        return 1;
    }

    // Access orbital parameters
    if (parser.sectionExists("object")) {
        auto* section = parser.getSection("object");
        if (section) {
            for (const auto& [key, param] : section->params) {
                std::cout << key << " = " << param.value <<
         std::endl;
            }
        }
    }

    return 0;
}
```

Advanced Integration

Pattern 1: Configuration Facade

Create a wrapper class for your domain model:

```
class OrbitalConfiguration {
private:
    ioc_config::OopParser parser_;

public:
    bool load(const std::string& filepath) {
        return parser_.loadFromOop(filepath);
    }

    // Domain-specific accessors
    int getObjectId() const {
        return std::stoi(parser_.getParameter("object", "id",
        "0"));
    }

    std::string getObjectName() const {
        return parser_.getParameter("object", "name", "Unknown");
    }

    void setObjectName(const std::string& name) {
        parser_.setParameter("object", "name", name);
    }

    bool save(const std::string& filepath) {
        return parser_.saveToOop(filepath);
    }
};

// Usage
OrbitalConfiguration config;
config.load("data.oop");
std::cout << config.getObjectName() << std::endl;
config.setObjectName("Vesta");
config.save("data_updated.oop");
```

Pattern 2: Configuration Validator

Create validation layer:

```
class ConfigValidator {
private:
    ioc_config::ConfigSchema schema_;

public:
    bool loadSchema(const std::string& filepath) {
        return schema_.loadFromJson(filepath);
    }

    bool validate(const ioc_config::OopParser& config) {
        if (!schema_.validateConfiguration(config)) {
            auto errors = schema_.getValidationErrors();
            for (const auto& error : errors) {
                std::cerr << "Validation error: " << error <<
        std::endl;
            }
```

```cpp
            return false;
        }
        return true;
    }
};

// Usage
ConfigValidator validator;
validator.loadSchema("schema.json");

OopParser parser;
parser.loadFromOop("config.oop");

if (validator.validate(parser)) {
    std::cout << "Configuration is valid" << std::endl;
}
```

Pattern 3: Multi-Format Support

```cpp
class ConfigurationManager {
private:
    std::string configFile_;
    std::string configFormat_;
    ioc_config::OopParser parser_;

public:
    bool loadConfiguration(const std::string& filepath) {
        configFile_ = filepath;

        // Detect format from extension
        size_t dotPos = filepath.rfind('.');
        if (dotPos != std::string::npos) {
            configFormat_ = filepath.substr(dotPos + 1);
        }

        // Load appropriate format
        if (configFormat_ == "json") {
            return parser_.loadFromJson(filepath);
        } else if (configFormat_ == "xml") {
            return parser_.loadFromXml(filepath);
        } else if (configFormat_ == "csv") {
            return parser_.loadFromCsv(filepath);
        } else {
            return parser_.loadFromOop(filepath);
        }
    }
    bool convertTo(const std::string& targetFormat, const
        std::string& outputFile) {
        if (targetFormat == "json") {
            return parser_.saveToJson(outputFile);
        } else if (targetFormat == "xml") {
            return parser_.saveToXml(outputFile);
        } else if (targetFormat == "csv") {
            return parser_.saveToOop(outputFile);  // OOP is
        default
        } else {
            return parser_.saveToOop(outputFile);
        }
    }
};
```

Pattern 4: Versioning and Rollback

```
class VersionedConfiguration {
private:
    ioc_config::VersionedOopParser parser_;

public:
    bool initialize(const std::string& filepath) {
        if (!parser_.loadFromOop(filepath)) {
            return false;
        }
        return parser_.enableVersioning();
    }

    void updateAndVersion(const std::string& section,
                          const std::string& key,
                          const std::string& value,
                          const std::string& description) {
        parser_.setParameter(section, key, value);
        parser_.createVersion(description);
    }

    void showHistory() {
        auto history = parser_.getHistory();
        for (const auto& entry : history) {
            std::cout << "Version " << entry.number << " - "
                      << entry.timestamp << ": " <<
        entry.description << std::endl;
        }
    }

    void revertToVersion(size_t versionNumber) {
        parser_.rollback(versionNumber);
    }
};
```

Pattern 5: Concurrent Access

For multi-threaded applications:

```
class ThreadSafeConfigManager {
private:
    std::shared_ptr<ioc_config::OopParser> parser_;
    mutable std::shared_mutex configMutex_;

public:
    bool loadConfiguration(const std::string& filepath) {
        std::unique_lock<std::shared_mutex> lock(configMutex_);
        auto temp = std::make_shared<ioc_config::OopParser>();

        if (!temp->loadFromOop(filepath)) {
            return false;
        }
        parser_ = temp;
        return true;
    }

    std::string getParameter(const std::string& section,
                             const std::string& key,
                             const std::string& defaultVal = "") {
        std::shared_lock<std::shared_mutex> lock(configMutex_);
        if (!parser_) return defaultVal;
        return parser_->getParameter(section, key, defaultVal);
    }

    void updateParameter(const std::string& section,
                        const std::string& key,
                        const std::string& value) {
        std::unique_lock<std::shared_mutex> lock(configMutex_);
        if (parser_) {
            parser_->setParameter(section, key, value);
        }
    }
};
```

Common Scenarios

Scenario 1: Loading Orbital Data Configuration

```
struct OrbitalData {
    int objectId;
    std::string objectName;
    double magnitude;
    std::string discoveryDate;
};

class OrbitalDataLoader {
public:
    OrbitalData loadFromConfig(const std::string& configFile) {
        ioc_config::OopParser parser;
        parser.loadFromOop(configFile);

        OrbitalData data;
        data.objectId = std::stoi(parser.getParameter("object",
         "id", "0"));
        data.objectName = parser.getParameter("object", "name",
         "Unknown");
        data.magnitude = std::stod(parser.getParameter("object",
         "magnitude", "0.0"));
        data.discoveryDate = parser.getParameter("object",
         "discovery_date", "");

        return data;
    }
};
```

Scenario 2: Batch Configuration Conversion

```
void convertConfigurationDirectory(const std::string& sourceDir,
                                   const std::string& targetDir,
                                   const std::string&
        targetFormat) {
    std::vector<std::string> files;

    // Collect all .oop files
    for (const auto& entry :
        std::filesystem::directory_iterator(sourceDir)) {
        if (entry.path().extension() == ".oop") {
            files.push_back(entry.path().string());
        }
    }

    // Batch convert
    auto stats = ioc_config::BatchProcessor::convertAll(
        files, "oop", targetFormat, targetDir
    );

    std::cout << "Converted " << stats.successful_operations
              << " files successfully" << std::endl;

    if (stats.failed_operations > 0) {
        for (const auto& error : stats.error_messages) {
            std::cerr << "Error: " << error << std::endl;
        }
    }
}
```

Scenario 3: Configuration Merging

```
void mergeConfigurations(const std::vector<std::string>&
        configFiles,
                        const std::string& outputFile) {
    ioc_config::OopParser baseConfig;
    baseConfig.loadFromOop(configFiles[0]);

    // Merge all additional configs
    for (size_t i = 1; i < configFiles.size(); ++i) {
        ioc_config::OopParser otherConfig;
        otherConfig.loadFromOop(configFiles[i]);
        baseConfig.merge(otherConfig,
        ioc_config::MergeStrategy::APPEND);
    }

    baseConfig.saveToOop(outputFile);
}
```

Scenario 4: Configuration Diff Analysis

```
void analyzeChanges(const std::string& oldFile,
                   const std::string& newFile) {
    ioc_config::OopParser oldConfig, newConfig;
    oldConfig.loadFromOop(oldFile);
    newConfig.loadFromOop(newFile);

    auto differences = oldConfig.diff(newConfig);

    std::cout << "Changes found: " << differences.size() <<
        std::endl;
    for (const auto& diff : differences) {
        std::cout << "  [" << diff.section << "] " << diff.key
        << ": "
```

```
                      << diff.oldValue << " → " << diff.newValue <<
          std::endl;
    }
}
```

Scenario 5: Path-Based Access

```
void updateUsingPaths(ioc_config::OopParser& parser) {
    // Using RFC 6901 JSON Pointer syntax

    // Get value by path
    std::string value = parser.getValueByPath("/object/name");

    // Set value by path (creates sections if needed)
    parser.setValueByPath("/search/magnitude", "16.5");

    // Delete by path
    parser.deleteByPath("/object/deprecated_field");
}
```

Performance Optimization

1. Caching Configuration

For frequently read configurations:

```
class CachedConfiguration {
private:
    std::map<std::string, std::string> cache_;
    ioc_config::OopParser parser_;

public:
    void loadAndCache(const std::string& filepath) {
        parser_.loadFromOop(filepath);
        rebuildCache();
    }

private:
    void rebuildCache() {
        cache_.clear();
        for (const auto& section : parser_.getSectionNames()) {
            for (const auto& key :
        parser_.getParameterKeys(section)) {
                std::string cacheKey = section + "/" + key;
                cache_[cacheKey] = parser_.getParameter(section,
        key);
            }
```

```cpp
        }
    }

public:
    std::string get(const std::string& section,
                    const std::string& key,
                    const std::string& defaultVal = "") const {
        std::string cacheKey = section + "/" + key;
        auto it = cache_.find(cacheKey);
        if (it != cache_.end()) {
            return it->second;
        }
        return defaultVal;
    }
};
```

2. Lazy Loading

Load configuration only when needed:

```cpp
class LazyConfiguration {
private:
    std::string filepath_;
    ioc_config::OopParser parser_;
    bool loaded_ = false;

public:
    void setPath(const std::string& filepath) {
        filepath_ = filepath;
        loaded_ = false;
    }

private:
    void ensureLoaded() {
        if (!loaded_) {
            parser_.loadFromOop(filepath_);
            loaded_ = true;
        }
    }

public:
    std::string get(const std::string& section,
                    const std::string& key,
                    const std::string& defaultVal = "") {
        ensureLoaded();
        return parser_.getParameter(section, key, defaultVal);
    }
};
```

3. Incremental Updates

Update only changed values:

```
void incrementalUpdate(ioc_config::OopParser& parser,
                      const std::map<std::string,
        std::map<std::string, std::string>>& updates) {
    for (const auto& [section, params] : updates) {
        for (const auto& [key, value] : params) {
            std::string current = parser.getParameter(section,
        key);
            if (current != value) {  // Only update if changed
                parser.setParameter(section, key, value);
            }
        }
    }
}
```

Troubleshooting

Issue: File Not Found

Problem: `loadFromOop()` returns false with error “File not found”

Solutions: 1. Verify absolute path: Use `std::filesystem::absolute(path)` 2. Check working directory: Print `std::filesystem::current_path()` 3. Use error message: `parser.getLastError()`

```
#include <filesystem>

std::string absolutePath =
        std::filesystem::absolute("config.oop").string();
if (!parser.loadFromOop(absolutePath)) {
    std::cerr << "Failed: " << parser.getLastError() <<
        std::endl;
}
```

Issue: Invalid Format

Problem: Parser fails with “Invalid format” or incorrect parsing

Solutions: 1. Verify file format matches parser type 2. Check for BOM (Byte Order Mark) 3. Ensure proper line endings (use `\n` not `\r\n`) 4. Test with validator

```
// Explicit format loading
std::string format = "oop";  // or "json", "xml", etc.
if (format == "json") {
    parser.loadFromJson(filepath);
} else if (format == "oop") {
    parser.loadFromOop(filepath);
}
```

Issue: Type Detection Errors

Problem: `detectType()` returns unexpected type

Solutions: 1. Check actual value format 2. For JSON values, ensure they're valid 3. Use explicit type conversion

```
auto param = parser.getParameter("section", "key");
std::string detectedType = parser.detectType(param);

// Manual type check
if (param.find('{') != std::string::npos) {
    // JSON value
} else if (param == "true" || param == "false") {
    // Boolean
}
```

Issue: Thread Safety Problems

Problem: Crashes or corruption in multi-threaded environment

Solutions: 1. Use ThreadSafeConfigManager wrapper 2. Ensure only one write at a time 3. Use shared_lock for reads

```
class SafeConfig {
private:
    std::shared_ptr<ioc_config::OopParser> parser_;
    mutable std::shared_mutex mutex_;

public:
    // All access through this class
    std::string get(const std::string& section, const
         std::string& key) {
        std::shared_lock<std::shared_mutex> lock(mutex_);
        return parser_->getParameter(section, key);
    }
};
```

Issue: Memory Leaks

Problem: Memory usage grows continuously

Solutions: 1. Check you're not holding pointers indefinitely 2. Clear old versions with `deleteOldVersions()` 3. Use smart pointers consistently

```
// Bad - leaks memory in loop
for (int i = 0; i < 1000; ++i) {
    OopParser* p = new OopParser();
    p->loadFromOop("config.oop");
    // Never deleted
}

// Good - automatic cleanup
for (int i = 0; i < 1000; ++i) {
    auto p = std::make_unique<OopParser>();
    p->loadFromOop("config.oop");
    // Automatically deleted at end of scope
}
```

Issue: Performance Degradation

Problem: Load/save operations become slow

Solutions: 1. Check file size (> 100MB may be slow) 2. Enable caching for repeated access 3. Use streaming for very large files 4. Batch operations instead of individual operations

```
// Slow - individual operations
for (const auto& file : files) {
    OopParser parser;
    parser.loadFromOop(file);
    // Process...
}

// Fast - batch operation
auto stats = ioc_config::BatchProcessor::validateAll(files);
```

Summary

IOC_Config provides flexible integration options for various project requirements. Start with basic integration and gradually adopt advanced patterns as needs grow.

Key Takeaways: - ✅ Use wrapper classes for domain-specific logic - ✅ Implement thread-safety wrapper for multi-threaded apps - ✅ Cache frequently accessed configurations - ✅ Use batch operations for multiple files - ✅ Monitor error messages for debugging

Next Steps: 1. Review API_REFERENCE.md for detailed method documentation 2. Check examples/ directory for sample code 3. Run tests to verify integration: `ctest` 4. Profile performance for your specific use case