SEP Project
Coding Standards

CodeCatalyst UG33

Team Documentation

Version 1.0

September 6, 2025

Contents

1 Quick Reference

Essential commands and standards for immediate productivity.

1.1 Pre-Commit Quality Checks

Mandatory Before Every Commit

Required quality checks:

Format code: `./scripts/format-code.sh`

All tests pass: `make test-all`

Cross-platform build: `make windows-package`

Clean build: `make clean && make test-all`

1.2 Code Quality Commands

Essential Quality Commands

```
# Format all source files (mandatory)
./scripts/format-code.sh

# Build and test everything
make test-all

# Format specific file
clang-format -i src/block.cpp

# Check formatting compliance
clang-format --dry-run --Werror src/block.cpp

# Generate compile commands for IDE
make compile-commands

# Cross-platform build verification
make windows-package
```

1.3 Naming Quick Reference

Element	Style	Example
Classes/Structs	`CamelCase`	`BlockModel`
Functions	`snake_case`	`read_specification()`
Variables	`snake_case`	`x_count`
Constants	`UPPER_CASE`	`MAX_DEPTH`
Private members	Suffix `_`	`data_`

Table 1: Naming Convention Quick Reference

2 Code Formatting Standards

Automatic code formatting requirements and configuration.

2.1 clang-format (Mandatory)

Mandatory Formatting Requirement

All code must be formatted with clang-format before submission.
This is enforced by:

- CI/CD pipeline checks
- Code review requirements
- Pre-commit hooks (when configured)
- Team development standards

Non-compliance will result in:

- Failed CI/CD builds
- Rejected pull requests
- Request for reformatting

2.2 Style Configuration

2.2.1 Project Style Rules

Formatting Configuration (Google-based with modifications)

Key formatting rules:

- **Line length**: 100 characters maximum
- **Indentation**: 4 spaces (no tabs)
- **Braces**: Attached style ({ on same line)
- **Pointer/Reference**: Left-aligned (`int* ptr`, `int& ref`)
- **Access modifiers**: Indented -2 spaces from class level

Spacing rules:

- No spaces inside parentheses: `function(arg)` not `function( arg )`
- Spaces around binary operators: `a + b` not `a+b`
- Space after control statements: `if (condition)` not `if(condition)`
- Two spaces before trailing comments

2.2.2 Example Code Formatting

Properly Formatted C++ Code

```cpp
class BlockModel {
  public:
    void read_specification();
    int get_block_count() const;

    // Constructor with member initialization list
    BlockModel(int x_count, int y_count, int z_count)
        : x_count_(x_count), y_count_(y_count), z_count_(z_count) {}

  private:
    int x_count_;
    int y_count_;
    int z_count_;
    std::vector<Block> blocks_;

    void process_internal_data();
};

// Function implementation with proper spacing
void BlockModel::read_specification() {
    std::string line;
    if (std::getline(std::cin, line)) {
        std::vector<int> values = split_csv_ints(line);
        if (values.size() >= 6) {
            x_count_ = values[0];
            y_count_ = values[1];
            z_count_ = values[2];
        }
    }
}
```

2.3 Formatting Commands

clang-format Usage

```bash
# Format all source files (recommended before commit)
cmake --build build --target format

# Format specific file
clang-format -i src/block.cpp
clang-format -i include/block.h

# Format all files manually
find src/ include/ tests/ -name "*.cpp" -o -name "*.h" | \
    xargs clang-format -i

# Check formatting without modifying files
clang-format --dry-run --Werror src/block.cpp

# Show formatting differences
clang-format --dry-run src/block.cpp | diff src/block.cpp -
```

2.4 IDE Integration

Automatic Formatting Setup

Visual Studio Code:

- Install C/C++ Extension Pack
- Enable "Format on Save" in settings
- Configure to use project `.clang-format` file
- Keyboard shortcut: Shift+Alt+F

CLion:

- Import `.clang-format` in Code Style settings
- Enable "Reformat code" on commit
- Keyboard shortcut: Ctrl+Alt+L

Vim/Neovim:

- Use vim-clang-format plugin
- Map to key binding: `nnoremap <Leader>f :ClangFormat<CR>`
- Auto-format on save with autocmd

3 Naming Conventions

Comprehensive naming standards for consistent code readability.

3.1 Detailed Naming Rules

Element	Style	Example	Notes
Classes	`CamelCase`	`BlockModel`, `Vec3`	
Structs	`CamelCase`	`Point3D`, `BlockData`	
Functions	`snake_case`	`read_specification()`, `get_count()`	
Methods	`snake_case`	`calculate_volume()`, `is_valid()`	
Variables	`snake_case`	`x_count`, `block_size`	
Parameters	`snake_case`	`input_file`, `max_depth`	
Local variables	`snake_case`	`temp_value`, `i`, `j`	
Member variables	`snake_case_`	`x_count_`, `blocks_`	Suffix
Constants	`UPPER_CASE`	`MAX_DEPTH`, `DEFAULT_SIZE`	
Enums	`CamelCase`	`BlockType`, `ErrorCode`	
Enum values	`UPPER_CASE`	`SOLID_BLOCK`, `SUCCESS`	
Namespaces	`snake_case`	`block_utils`, `math`	
Files	`snake_case`	`block_model.cpp`, `utils.h`	
Macros	`UPPER_CASE`	`DEBUG_PRINT`, `ASSERT`	Avoid

Table 2: Complete Naming Convention Reference

3.2 Naming Examples

3.2.1 Good Naming Examples

Proper Naming Practices

```cpp
// Class and member naming
class BlockModel {
  public:
    void read_specification();
    bool is_valid_block(const Block& block) const;
    std::size_t get_block_count() const { return blocks_.size(); }

  private:
    int x_count_;                     // Member variable with underscore
    int y_count_;
    std::vector<Block> blocks_;       // Container with clear name
    std::unordered_map<char, std::string> tag_table_;
};

// Function and variable naming
void process_model_data(const std::string& input_file) {
    int total_blocks = 0;            // Local variable
    bool processing_complete = false;

    for (const auto& current_block : blocks_) {
        if (current_block.is_valid()) {
            total_blocks++;
        }
    }
}

// Constants and enums
const int MAX_MODEL_DIMENSION = 1024;
const double COMPRESSION_THRESHOLD = 0.85;

enum class BlockType {
    SOLID_BLOCK,
    EMPTY_SPACE,
    BOUNDARY_MARKER
};
```

3.2.2 Poor Naming Examples

Naming Practices to Avoid

```
// Poor naming examples - DO NOT USE
class bm {                                 // Unclear abbreviation
  public:
    void rd();                             // Meaningless function name
    int gc() const;                        // Unclear abbreviation

  private:
    int x, y, z;                           // Too generic
    std::vector<Block> b;                  // Single character name
    std::map<char, std::string> tt;        // Unclear abbreviation
};

// Poor variable naming
void func(std::string f) {                 // Unclear parameter name
    int n = 0;                             // Meaningless variable name
    bool flag = false;                     // Generic flag name

    for (auto& item : collection) {        // 'item' is too generic
        if (item.flag2) {                  // Numbered variables indicate poor
           design
            n++;
        }
    }
}

// Poor constants
#define MAX 1000                           // Too generic
const int X = 50;                          // Meaningless name
```

4 C++ Coding Standards

Best practices for modern C++ development.

4.1 Language Standards

C++ Standard Requirements

Project uses C++17 standard:

- Compiler flag: `-std=c++17`
- Modern C++ features encouraged
- Standard library preferred over custom implementations
- Performance and safety focus

Compiler support:

- GCC 7.0+ (primary Linux compiler)
- Clang 7.0+ (alternative compiler)
- MSVC 2019+ (Windows compiler)
- MinGW-w64 (cross-compilation)

4.2 Namespace Usage

Namespace Rules (Strictly Enforced)

Prohibited practices:

- `using namespace std;` – Never use in any file
- `using namespace` directives in headers
- Importing entire namespaces globally

Required practices:

- Always use explicit `std::` prefixes
- Individual using declarations acceptable in limited scope
- Namespace aliases for long namespace names

4.2.1 Namespace Examples

Proper Namespace Usage

```cpp
// Good: Explicit std:: prefixes
#include <iostream>
#include <vector>
#include <string>

void process_data() {
    std::vector<int> numbers;
    std::string input_line;
    std::cout << "Processing data..." << std::endl;

    // Acceptable: Limited scope using declaration
    {
        using std::cout;
        using std::endl;
        cout << "Temporary scope usage" << endl;
    }
}

// Good: Namespace alias for long names
namespace bg = boost::geometry;
bg::point<double, 2> create_point(double x, double y);
```

Prohibited Namespace Usage

```cpp
// BAD: Never use global using namespace
using namespace std;

// BAD: Don't import namespaces in headers
// In header file:
using namespace std;        // Affects all files that include this header

// BAD: Don't use in global scope
using std::vector;          // Pollutes global namespace
using std::string;

void process_data() {
    vector<int> numbers;    // Unclear which vector this is
    string input_line;      // Could be std::string or another string class
}
```

4.3 Header Management

Header Include Practices

Prefer specific includes:

```
// Good: Individual standard library headers
#include <iostream>     // For std::cout, std::cin
#include <vector>       // For std::vector
#include <string>       // For std::string
#include <algorithm>    // For std::sort, std::find
#include <memory>       // For std::unique_ptr, std::shared_ptr

// Good: Project headers with quotes
#include "block.h"
#include "block_model.h"
```

Avoid convenience headers:

```
// BAD: Don't use convenience headers
#include <bits/stdc++.h>    // Non-standard, bloated
#include <iostream.h>       // Deprecated form
```

4.4 Modern C++ Features

Encouraged Modern C++ Practices

C++17 features to use:

```
// Structured bindings (C++17)
auto [x, y, z] = get_coordinates();

// if constexpr (C++17)
template<typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        // Handle integer types
    } else {
        // Handle other types
    }
}

// std::optional (C++17)
std::optional<Block> find_block(int id) {
    if (block_exists(id)) {
        return Block{id};
    }
    return std::nullopt;
}

// Range-based for loops (C++11, enhanced in C++17)
for (const auto& block : blocks_) {
    process_block(block);
}

// Auto type deduction
auto result = expensive_computation();
const auto& reference = get_large_object();
```

5 Static Analysis

Automated code quality checking with clang-tidy.

5.1 clang-tidy Configuration

Static Analysis Benefits

clang-tidy provides:

- Bug detection and prevention
- Performance optimization suggestions
- Modern C++ feature recommendations
- Code readability improvements
- Best practice enforcement

Available via tools:

- Manual clang-tidy execution
- CI/CD pipeline integration
- IDE integration available
- Customizable rule sets

5.2 Enabled Check Categories

Check Category	Purpose
`clang-diagnostic-*`	Compiler diagnostic messages
`clang-analyzer-*`	Static analysis checks
`cppcoreguidelines-`	C++ Core Guidelines compliance
`modernize-*`	Modern C++ feature suggestions
`performance-*`	Performance optimization opportunities
`readability-*`	Code readability improvements
`bugprone-*`	Bug-prone pattern detection

Table 3: Enabled clang-tidy Check Categories

5.3 Running Static Analysis

clang-tidy Usage

```
# Analyze specific file
clang-tidy src/block.cpp -- -Iinclude

# Run with specific checks only
clang-tidy -checks='readability-*,performance-*' src/block.cpp

# Generate compile commands for accurate analysis
make compile-commands

# Run clang-tidy with compile commands database
clang-tidy -p . src/block.cpp

# Format and check quality together
./scripts/format-code.sh && make test-all
```

5.4 Common Issues and Fixes

5.4.1 Readability Issues

Readability Problems

```
// Issue: Magic numbers
if (blocks.size() > 100) {           // What does 100 represent?
    compress_data();
}

// Issue: Long parameter lists
void process(int x, int y, int z, int w, int h, int d, char tag, bool flag);

// Issue: Unclear variable names
bool flag = true;
int n = calculate();
```

Fixes:

```
// Fix: Named constants
const std::size_t MAX_BLOCKS_BEFORE_COMPRESSION = 100;
if (blocks.size() > MAX_BLOCKS_BEFORE_COMPRESSION) {
    compress_data();
}

// Fix: Structure parameters
struct BlockParameters {
    int x, y, z, width, height, depth;
    char tag;
    bool is_compressed;
};
void process(const BlockParameters& params);

// Fix: Descriptive names
bool is_compression_needed = true;
int total_block_count = calculate();
```

5.4.2 Performance Issues

Performance Problems

```cpp
// Issue: Unnecessary copies
for (std::string item : large_collection) {     // Copies each string
    process(item);
}

// Issue: Repeated expensive operations
for (int i = 0; i < vec.size(); ++i) {          // size() called repeatedly
    if (expensive_function() > threshold) {     // expensive_function()
       called repeatedly
        process(vec[i]);
    }
}
```

Fixes:

```cpp
// Fix: Use const references
for (const std::string& item : large_collection) {
    process(item);
}

// Fix: Cache expensive operations
const auto vec_size = vec.size();
const auto expensive_result = expensive_function();
for (std::size_t i = 0; i < vec_size; ++i) {
    if (expensive_result > threshold) {
        process(vec[i]);
    }
}
```

6 Testing Standards

Comprehensive testing framework and quality assurance.

6.1 Test Architecture

Testing Framework Components

Test types implemented:

- **Unit Tests** – Test individual components in isolation
- **Integration Tests** – Test component interactions end-to-end
- **Validation Tests** – Verify output correctness and reconstruction
- **Cross-Platform Tests** – Ensure compatibility across systems

Test organization:

- Tests in `tests/` directory
- Test data in `tests/data/`
- CTest integration for automation
- CI/CD pipeline integration

6.2 Test Implementation Standards

6.2.1 Unit Test Guidelines

Unit Test Best Practices

```
// Good: Descriptive test function names
void test_block_creation_with_valid_parameters() {
    // Arrange
    const int x = 10, y = 20, z = 30;
    const int width = 5, height = 6, depth = 7;
    const char tag = 'A';

    // Act
    Block block(x, y, z, width, height, depth, tag);

    // Assert
    assert(block.x == x);
    assert(block.y == y);
    assert(block.z == z);
    assert(block.width == width);
    assert(block.height == height);
    assert(block.depth == depth);
    assert(block.tag == tag);
}

void test_block_model_compression_algorithm() {
    // Arrange
    BlockModel model;
    // ... setup test data

    // Act
    model.read_specification();
    model.read_tag_table();
    model.read_model();

    // Assert - verify expected behavior
    // Check output format, block count, etc.
}
```

6.2.2 Integration Test Guidelines

Integration Test Structure

```
// Integration test: Full pipeline verification
void test_compression_and_validation_pipeline() {
    // Test the complete workflow:
    // Input -> Compression -> Output -> Validation -> Reconstruction

    // 1. Setup test input
    std::string test_input = load_test_case("case1.txt");

    // 2. Run compression
    std::ostringstream compressed_output;
    BlockModel model;
    // ... run compression algorithm

    // 3. Run validation
    std::istringstream validation_input(compressed_output.str());
    bool validation_passed = run_validation_test(validation_input);

    // 4. Assert results
    assert(validation_passed);
    assert(output_format_is_correct(compressed_output.str()));
}
```

6.3 Test Execution

Running Tests

```
# Run all tests (recommended)
make test-all

# Run specific test types
make test-compression-unit
make test-integration

# Run with sample data
make run-case1
make run-case2

# Validate output
make validate-case1
make validate-case2

# Clean build and test
make clean && make test-all
```

6.4 Test Data Management

Test Data Organization

Test data structure:

- `tests/data/case1.txt` – Smaller test case (64×8×5)
- `tests/data/case2.txt` – Larger test case (64×16×5)
- Test data follows project input format specification
- Data files version controlled for consistency

Test data guidelines:

- Include edge cases and boundary conditions
- Test both successful and failure scenarios
- Use representative real-world data
- Keep test data files reasonably sized

7 CI/CD Quality Gates

Automated quality assurance in the development pipeline.

7.1 Pipeline Overview

Quality Pipeline Structure

Quality gates implemented:

- **Code formatting** – clang-format compliance check
- **Static analysis** – clang-tidy warning detection
- **Build verification** – Cross-platform compilation
- **Test execution** – All tests must pass
- **Documentation** – Updates and consistency checks

Optimized workflow:

- Quick validation on every PR (3 minutes)
- Comprehensive validation on schedule/manual trigger
- Smart change detection to avoid redundant runs
- 75% reduction in CI resource usage

7.2 Quality Requirements

Merge Requirements

Before merging to main branch:

- All tests pass on primary platform (Ubuntu)
- Code formatting compliance verified
- No clang-tidy warnings introduced
- Documentation updated as needed
- PR review approval from team member

Comprehensive validation (scheduled):

- Multi-platform builds (Linux, macOS, Windows)
- Multiple compiler compatibility (GCC, Clang, MSVC)
- Cross-compilation verification
- Performance regression detection

7.3 Local Quality Verification

Pre-Commit Verification Script

```bash
#!/bin/bash
# Save as scripts/verify-quality.sh

echo "     Running pre-commit quality checks..."

# 1. Format check
echo "     Checking code formatting..."
if ! ./scripts/format-code.sh; then
    echo "    Code formatting failed"
    exit 1
fi

# 2. Build and test check
echo "     Building and testing project..."
if ! make test-all; then
    echo "    Build or tests failed"
    exit 1
fi

# 3. Cross-platform check (if on Linux)
if command -v x86_64-w64-mingw32-g++ &> /dev/null; then
    echo "     Checking Windows cross-compilation..."
    if ! make windows-package; then
        echo "    Windows cross-compilation failed"
        exit 1
    fi
fi

# 4. IDE support check
echo "     Updating IDE support..."
if command -v bear &> /dev/null; then
    make compile-commands > /dev/null
fi

echo "    All quality checks passed!"
```

8 Best Practices Summary

8.1 Daily Development Checklist

Developer Daily Checklist

Before starting work:

Pull latest changes: `git pull origin main`

Create feature branch: `git checkout -b UG33-XX-feature-name`

Verify build: `make test-all`

During development:

Format code regularly: `./scripts/format-code.sh`

Run relevant tests: `make test-compression-unit`

Test with sample data: `make run-case1`

Before committing:

All tests pass: `make test-all`

Code formatted: `./scripts/format-code.sh`

Clean build works: `make clean && make test-all`

Cross-platform build: `make windows-package`

Documentation updated if needed

Meaningful commit message (conventional commits)

8.2 Code Review Guidelines

Code Review Focus Areas

Technical review points:

- Correctness and logic validation
- Performance implications
- Memory management and resource handling
- Error handling and edge cases
- Test coverage adequacy

Style and maintainability:

- Naming convention compliance
- Code organization and structure
- Documentation and comments
- Consistent formatting (automated)
- Modern C++ feature usage

9 Related Documentation

- **development-environment.tex** – Complete environment setup guide
- **README.md** – Quick reference and essential commands
- **Git & Github Workflow.pdf** – Version control and collaboration
- **.clang-format** – Formatting configuration file

- **.clang-tidy** – Static analysis configuration file
- **Makefile** – Build system configuration