SEP Project
Project Structure

CodeCatalyst UG33

Team Documentation

Version 1.1

September 6, 2025

Contents

1 Quick Reference

This section provides essential commands for project build and testing workflows.

1.1 Essential Make Commands

Build and Test Commands

```
# Development workflow
make test-all               # Build and test everything
make run-case1              # Run with test data
make windows-package        # Create submission package

# Build targets
make                        # Build main executable
make test                   # Build test programs
make clean                  # Clean build artifacts

# Testing targets
make test-compression-unit  # Run algorithm unit tests
make test-integration      # Run end-to-end tests
make validate-case1        # Validate specific test case

# Help and information
make help                   # Show all available targets
```

1.2 Quick Build Checklist

Before Submission

Essential checks before submitting code:

`make test-all` passes completely

Code follows project structure (`src/`, `include/`, `tests/`)

No `using namespace std` in code

Build artifacts only in `build/` directory

Windows package builds successfully (`make windows-package`)

All unit tests pass (compression algorithm)

All integration tests pass (end-to-end pipeline)

2 Project Structure and Build System

This section covers the standardized C++ project structure and build system using Make.

2.1 Project Directory Structure

The project follows standard C++ conventions with organized directories:

Directory	Purpose
src/	Source files (.cpp) including main.cpp
include/	Header files (.h)
tests/	Test files and test data
build/	Build output directory (auto-generated)
docs/	Documentation files
Makefile	Build configuration and automation

Table 1: Project Directory Structure

2.2 Detailed Directory Contents

2.2.1 Source Files (src/)

File	Purpose
main.cpp	Program entry point
block.cpp	Block class implementation
block_growth.cpp	Block growth algorithm
block_model.cpp	Model reading and processing

Table 2: Source Files Organization

2.2.2 Header Files (include/)

File	Purpose
block.h	Block class definition
block_growth.h	Growth algorithm interface
block_model.h	Model processing interface

Table 3: Header Files Organization

2.2.3 Test Files (tests/)

File/Directory	Purpose
compression_test.cp	Unit tests for compression algorithm
validate_test.cpp	Integration tests for output validation
data/case1.txt	Test case data (64x8x5)
data/case2.txt	Test case data (64x16x5)

Table 4: Test Files Organization

2.3 Build System with Make

The project uses a comprehensive Makefile for all build operations:

2.3.1 Primary Build Targets

Main Build Commands

```
# Build the main executable
make

# Build everything and run comprehensive tests
make test-all

# Cross-compile for Windows and package
make windows-package

# Clean build artifacts
make clean
```

2.3.2 All Available Make Targets

Target	Description
`all`	Build the main executable (default)
`test`	Build both test executables
`test-all`	Run all tests (unit + integration)
`test-compression-unit`	Run compression algorithm unit tests
`test-integration`	Run integration tests (compress + validate)
`run-case1`	Run main program with case1.txt data
`run-case2`	Run main program with case2.txt data
`validate-case1`	Validate main program output with case1.txt
`validate-case2`	Validate main program output with case2.txt
`windows`	Cross-compile for Windows
`windows-package`	Complete Windows build and packaging
`clean`	Clean build artifacts
`help`	Show all available targets

Table 5: Complete Make Targets Reference

2.3.3 Automated Windows Cross-Compilation

The Makefile handles Windows compilation automatically:

Windows Build Process

```
# One command handles everything:
make windows-package

# This automatically:
# 1. Installs MinGW-w64 if needed
# 2. Cross-compiles with proper flags
# 3. Creates block_model.exe.zip
# 4. Ready for submission
```

Compilation flags explained:

- `-std=c++17` – Use C++17 standard
- `-O2` – Optimization level 2 for performance
- `-static` – Statically link libraries
- `-static-libstdc++ -static-libgcc` – Static linking for portability
- `-Iinclude` – Include directory for headers

3 Testing Framework

The project includes a comprehensive testing framework with two distinct test programs:

3.1 Test Programs

Test Program	Purpose
`compression_test.cp`	Unit tests for compression algorithm
`validate_test.cpp`	Integration tests for output validation

Table 6: Test Programs and Their Purposes

3.1.1 Compression Test Program

The compression test program validates the algorithm directly:

Compression Test Features

What it tests:

- Basic compression functionality
- Algorithm with case1.txt (generates expected block count)
- Algorithm with case2.txt (validates output length)
- BlockModel class instantiation and operations

Code quality standards:

- No `using namespace std`
- Proper `std::` prefixes throughout
- Individual header includes (not `<bits/stdc++.h>`)
- Links with library objects (excludes main.o)

3.1.2 Validation Test Program

The validation test program verifies output correctness:

Validation Test Features

What it does:

- Takes compressed block output (format: `x,y,z,width,height,depth,label`)
- Reconstructs 3D model from compressed blocks
- Outputs visual representation to verify correctness
- Validates reconstruction matches original data

Usage patterns:

- Pipeline testing: `block_model < case1.txt | validate_test`
- Interactive validation with manual input
- Automated integration testing

3.2 Test Commands

Comprehensive Testing

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

# Run specific test types
make test-compression-unit   # Algorithm unit tests
make test-integration       # End-to-end pipeline tests

# Individual test programs
make run-compression-test   # Run compression tests directly
make run-validate-test      # Interactive validation test

# Test with specific case data
make validate-case1         # Test with case1.txt
make validate-case2         # Test with case2.txt
```

3.3 Test Data Organization

Test data is organized within the `tests/` directory:

- `tests/data/case1.txt` – Smaller test case (64x8x5)
- `tests/data/case2.txt` – Larger test case (64x16x5)
- Test programs validate both compression and reconstruction
- Data files follow the project input format specification

3.4 Expected Test Output

3.4.1 Compression Unit Tests

Example Output

```
$ make test-compression-unit
=== Compression Test Suite ===
Running compression tests...
Testing basic compression...
    Basic compression test passed
Testing case1 compression...
    Case1 compression test passed - generated 86 blocks
Testing case2 compression...
    Case2 compression test passed - output length: 2134 chars
All compression tests passed!
```

3.4.2 Integration Tests

Example Output

```
$ make test-integration
Running integration tests (compression + validation)...
Testing case1.txt...
    Case 1 integration passed
Testing case2.txt...
    Case 2 integration passed
All integration tests completed!
```

4 Code Quality Standards

This section defines the coding standards and practices for the project.

4.1 C++ Coding Standards

Required Coding Practices

Namespace usage:

- Never use `using namespace std`
- Always use explicit `std::` prefixes
- Use individual header includes (`#include <iostream>`)
- Avoid `#include <bits/stdc++.h>`

File organization:

- Headers in `include/` directory
- Sources in `src/` directory
- Include paths relative to repository root
- Use `-Iinclude` compiler flag

4.2 Code Formatting Standards

4.2.1 Required: clang-format Usage

All code must be formatted using clang-format before submission. This ensures consistent formatting across the team and reduces review overhead.

clang-format Requirement

Mandatory for all submissions:

- All C++ source files must be formatted with clang-format
- Configure your development environment for automatic formatting
- Run clang-format before committing code
- PR reviews will reject improperly formatted code

Benefits of consistent formatting:

- Eliminates formatting discussions in code reviews
- Reduces diff noise in pull requests
- Improves code review focus on logic vs style
- Maintains consistent project appearance

clang-format Usage

```
# Install clang-format (Ubuntu/Debian)
sudo apt install clang-format

# Format a single file (required before commit)
clang-format -i src/main.cpp

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

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

Note: Project-specific clang-format configuration file will be provided. Configure your development environment to use clang-format automatically on save or before commit.

4.3 Build Artifact Management

Build Cleanliness Rules

Required practices:

- All build outputs go to `build/` directory
- No compiled files in source directories
- `.gitignore` excludes build artifacts
- `make clean` removes all generated files

Prohibited in version control:

- `⋆.exe`, `⋆.o`, `⋆.zip` files
- IDE-specific files (`.vscode/`, `.DS_Store`)
- Temporary or cache files
- Build directories with compiled code

5 Pre-Submission Verification

Before creating a pull request, run the complete verification process:

5.1 Automated Verification

Complete Verification Workflow

```
# 1. Clean and rebuild everything
make clean
make test-all

# 2. Verify Windows compilation
make windows-package

# 3. Test with both case files
make validate-case1
make validate-case2

# 4. Verify help documentation
make help
```

5.2 Manual Verification Checklist

Pre-Submission Checklist

Build Verification:

`make test-all` passes completely

Code compiles without errors or warnings

Windows package builds successfully (`make windows-package`)

All unit tests pass (compression algorithm)

All integration tests pass (end-to-end pipeline)

Code Quality:

No use of `using namespace std` in any files

Proper include paths (relative to repository root)

Build artifacts only in `build/` directory

Test data organized in `tests/data/`

Code follows project structure conventions

All code formatted with clang-format (mandatory)

Documentation:

README.md reflects current project structure

Make targets documented and working

Test procedures clearly explained

Build instructions are accurate

6 Troubleshooting

Common issues and their solutions:

6.1 Build Issues

Common Build Problems

Problem: Compilation errors about missing headers

- Check include paths use `-Iinclude` flag
- Verify headers are in `include/` directory
- Ensure `#include "header.h"` not `#include "../include/header.h"`

Problem: Multiple definition of main

- Test programs should link with library objects only
- Exclude `main.o` when building test executables
- Use `LIB_OBJECTS` variable in Makefile

6.2 Test Issues

Common Test Problems

Problem: Tests fail to find data files

- Ensure test data is in `tests/data/` directory
- Check Makefile `DATA_DIR` variable points correctly
- Run tests from repository root directory

Problem: Integration tests fail

- Verify main program builds successfully
- Check validation test accepts piped input
- Ensure output format matches expected input format

7 Related Documentation

This guide focuses on technical implementation and code quality. For other project aspects:

- **Git Workflow** – See `docs/Git & Github Workflow.pdf` for branching, rebasing, PR process
- **Project Planning** – See `docs/Jira Workflow.pdf` for ticket management and sprint planning
- **Quick Daily Reference** – See `README.md` for essential commands

7.1 Document Maintenance

- Update this guide when project structure or build system changes
- Ensure all team members understand the build and test processes
- Keep Make targets documentation synchronized with actual Makefile
- Update code quality standards as team practices evolve
- Use `make help` for live reference of available build targets