RefactoringAndHeaderFiles

January 18, 2026

1 Refactoring

1.1 External Resources

- Refactoring Guru
- YouTube Podcast: C++ Refactoring and Header Files
- YouTube Video: C++ Refactoring and Header Files
- Learning Materials on NotebookLM: Notebook LM Refactoring Resources

1.2 What is Refactoring?

- process of restructuring existing code without changing its external behavior
- improves nonfunctional attributes of the software
- intended to improve code readability and reduce complexity

1.3 Benefits of Refactoring

1.3.1 Improved Code Readability

- Cleaner function names
- Better variable names
- Simpler structure
- Makes the code easier to understand for you and others

1.3.2 Reduced Complexity

- Breaking down large functions into smaller ones
- Removing redundant code
- Simplifying complex logic
- Makes the code easier to maintain and modify

1.3.3 Reduced Code Duplication

- Identifying and eliminating duplicate code
- Promotes code reuse (e.g., using functions or classes)
- Makes the codebase smaller and more efficient

1.3.4 Easier Maintenance

- Well-structured code is easier to debug and fix
- Easier to add new features without breaking existing functionality

1.3.5 Enhanced Collaboration

- Clearer code is easier for team members to understand and work with
- Facilitates code reviews and knowledge sharing

1.3.6 Better Testing

- Refactored code is often easier to test
- Encourages writing unit tests for smaller, focused functions

1.3.7 Increased Performance

- Sometimes refactoring can lead to performance improvements
- Optimizing algorithms and data structures during refactoring

1.3.8 Fix Bad Structure

- Eliminate code smells (e.g., long functions, large classes)
- Improve overall architecture of the codebase

2 Header Files

- header files and their purpose
- standard library header files
- header file content rules
- using header files
- implementation files

2.1 Header files and their purposes

- **Header** files tell the compiler what exists.
- **Source** files tell the compiler how it works.
- we've used library header files such as `iostream`, `string`, etc.
- the purpose of this chapter is to learn how to create your own header files and why
- as program grows bigger, code need to be divided into many files
- using function prototype (forward declaration) and defining functions after main is not scalable
- breaking solution code into many files and organizing into logical folders has many advantages:
  - makes program easier to manage, read, update, debug
  - makes it easier to work in a team where each member works on a separate file
  - avoid conflicts while using version control such as git
- Generally, header files allow us to put declarations in one or more files and then include them wherever we need them
  - this can save a lot of typing in multi-file programs by avoiding repetitive declarations
  - allows us to separate interface (declarations) from implementation (definitions)
  - helps us create our own library of important functions
  - prevents circular dependencies in multi-file programs

2.2 Pitchfork Layout

- a common and highly recommended structure, often referred to as the **Pitchfork layout**, looks like this:

```
project_root/
  build/        # Generated build artifacts (object files, executables, etc.)
  docs/         # Project documentation
  include/      # Public header files (exposed to other projects/libraries)
  src/          # Source files (.c/.cpp) and private header files
  tests/        # Unit and integration tests
  external/     # External dependencies (third-party libraries)
  README.md     # Project description and build instructions
  Makefile      # Example build system file
  LICENSE       # Project license file
```

2.2.1 Key Organizational Practices

- **Separate Interface from Implementation**
  - a primary principle is to distinguish between public headers (your library's API) and implementation files.
  - `include/`:
    - contains all public header files (e.g., my_lib/my_class.h)
    - when your project is installed or used as a library by other projects, only the contents of this folder are exposed.
  - `src/`:
    - contains all source files (.c or .cpp) and any private headers that are only used internally within your project
- **Consistent Naming Conventions**
  - Use clear and consistent naming conventions for files and directories.
  - Example: Use lowercase with underscores for file names (e.g., my_class.h, my_class.cpp).
  - files and folders should not have spaces or special characters (except underscores) in their names
- **Logical Grouping**:
  - Group related files together in the directory structure.
- **Mirror Namespace/Module Structure**
  - Within `src/` and `include/`, it is good practice to use subdirectories to mirror your project's logical modules or namespaces.
    - Example: For a package named Game with a Player class, you would have `include/Game/Player.h` and `src/Game/Player.cpp`.
    - This allows you to use clear include directives like `#include "Game/Player.h"`, making dependencies explicit and preventing name collisions with other libraries
- **Out-of-Source Builds**
  - Always use a dedicated `build/` directory for generated files. This keeps your source tree clean and allows you to easily delete all build artifacts without affecting your source code.
- **Organize by Functionality**:
  - For larger projects, break your code into logical modules (e.g., `src/core`, `src/graphics`,

`src/physics`). This promotes separation of concerns.

- **Keep Tests Separate**:
  - Placing tests in their own `tests/` directory keeps them isolated from the main source code.
- **Small Projects**:
  - For very small projects with only a few files, a single folder for everything might be sufficient to start.
  - The structure should scale with the project; don't add complexity (extra folders) until you need it.
- By following these guidelines, your C/C++ project will be more maintainable and easier for other developers to understand and work with.

2.3 Creating header files

- header files typically contain declarations of functions, classes, structs, constants, macros, etc.
- header files do NOT contain function definitions or implementations
- header files have `.h` (C language) or `.hpp` (C++ language) extension
- header files contain only declarations and no executable code
- header files include preprocessor directives to avoid multiple inclusions
- syntax for include guard:

```
#ifndef HEADER_FILE_NAME_H
#define HEADER_FILE_NAME_H

// Declarations go here

#endif // HEADER_FILE_NAME_H
```

- `HEADER_FILE_NAME_H` is a unique identifier, typically the header file name in uppercase
- better syntax using `#pragma once` (non-standard but widely supported):

```
#pragma once

// Declarations go here
```

- header best practices:
  - must contain header include guards (to avoid being included multiple times in the final binary)
  - typically contains struct and class definitions only
  - contains function prototypes
  - avoid function definitions
  - may include other header files
  - do NOT declare global variables
  - each header file should be as independent as possible with specific purpose

2.4 Implementing header files

- typically a header file is implmented in a separate corresponding .cpp file
- functions and classes are implmented or defined in implementation files
- implementation file are regular .cpp files that must include the header being implemented

- must also include any library header files required to implment the functions
- syntax:

```
#include "header_file_name.h"
```

- note the double quotes `"header_file_name.h"` instead of `<>` `<header_file_name.h>` used for built-in headers

2.5 Using header files

- include user-defined header files similar to library header file
- include only the header files that are required
- syntax:

```
#include "header_file_name.h"
```

- use the functions and user-defined types, etc. defined in the included header file

2.6 Compiling multiple cpp files

- header files are not compiled; ONLY the CPP files
- compiling multiple file is similar to compiling single file using g++ compiler
  - simply list all the .cpp files separated by a space

```
g++ <switches: -std=c++17, -Wall, etc.> -o program_name input_file1.cpp input_file2.cpp ..
```

- or use the Makefile as shown in the following demo programs

2.6.1 Demo 1

- see folder `demos/header_files/header/`
- a simple demo program with one header file

2.6.2 Demo 2

- see folder `demos/header_files/triangle/`
- contains several header and cpp files
- a single-file `triangle.cpp` program provided in File IO chapter - `demos/file_io/triangle/` is separated into several `header` and `cpp` files
- test functions are separated into `test.h` and `test.cpp` files
- utility functions are separated into `utility.h` and `utility.cpp` files
- main entry/driver function is in `main.cpp` file
- triangle definition and related functions are separated into `triangle.h` and `triangle.cpp` files

[ ]: