

μC++ Extension
Software Requirements Specification

Group 27

Fei Lin

Kevin Song

Abdur Javaid

Kevin Pierce

University of Waterloo

SE 463

November 2024

Table of Contents

Table of Contents.....	1
1. Introduction.....	3
1.1 Purpose.....	3
1.2 Scope.....	3
1.3 Terms.....	4
1.4 Abbreviations.....	5
1.5 References.....	5
2. Overall Description.....	6
2.1 Product Perspectives.....	6
2.2 Product Functions.....	7
2.3 User Characteristics.....	8
2.4 Constraints.....	8
2.5 Assumptions and Dependencies.....	9
2.6 Use Case Diagram.....	10
3. Specific Requirements.....	11
3.1 Highlight μ C++ Syntax.....	11
3.1.1 Scenarios.....	11
3.1.2 Alternative and Exceptional Scenarios.....	11
3.1.3 User Interface.....	11
3.2 Code Completion.....	12
3.2.1 Functional Requirements.....	12
3.2.2 Alternative and Exceptional Scenarios.....	12
3.2.3 User Interface.....	12
3.3 Show μ C++ Symbol Signatures and Definitions on Hover.....	12
3.3.1 Functional Requirements.....	12
3.3.2 Alternative and Exceptional Scenarios.....	13
3.3.3 User Interface.....	13
3.4 Semantically Analyze Files.....	13
3.4.1 Scenarios.....	13
3.4.2 Alternative and Exceptional Scenarios.....	14
3.4.3 User Interface.....	14
3.5 Alt-Click on a Function / Class To Navigate To Its Definition.....	15
3.5.1 Scenarios.....	15
3.5.2 Alternative and Exceptional Scenarios.....	15
3.5.3 User Interface.....	16
3.6 Alt-Click on a Variable Gets all References to the Variable.....	16
3.6.1 Scenarios.....	16

3.6.2 Alternative and Exceptional Scenarios.....	17
3.6.3 User Interface.....	17
3.7 Apply Formatting / Autoformatting on the currently open file.....	18
3.7.1 Scenarios.....	18
3.7.2 Alternative and Exceptional Scenarios.....	18
3.7.3 User Interface.....	19
3.8 General Alternative and Exceptional Scenarios.....	19
3.8.1 Incompatible VSCode Version.....	20
3.8.2 Unstable VSCode Extension Host API.....	20
3.8.3 Bad Extension Install and Activation.....	20
3.8.4 Insufficient System Resources.....	20

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) is to define the functional and non-functional requirements of our Extension for Visual Studio Code (VSCode) project, detailing the expected behavior, system constraints, and interfaces necessary for a successful implementation. This document aims to provide a complete, unambiguous, and verifiable reference that will guide all phases of the project's development, testing, and deployment. By clearly articulating these requirements, this SRS ensures alignment between all project stakeholders and serves as the foundation for evaluating the completed system against the defined specifications.

This document is primarily intended for:

- Current SE490 Development Team:
 - To ensure the team has a detailed understanding of the requirements and system behavior for implementing the Extension.
- Future Maintainers:
 - To provide an authoritative guide that can be referenced for system maintenance, troubleshooting, or future enhancements.

This SRS can be used as:

- A Detailed Reference Guide:
 - Used by developers to inform design and implementation and to design test cases that verify system functionality and performance against the defined specifications.
- A Communication Framework:
 - Ensuring consistent understanding across all project members and users of the end deliverable about the features, constraints, and intended user experience of the Language Extension. This alignment minimizes ambiguities and reduces the risk of scope changes during development.

1.2 Scope

The Extension provides a development environment for the μ C++ programming language within VSCode. This extension allows developers to perform essential language-based tasks, including

code autocompletion, inline error diagnostics, and code navigation ("Go to Definition"), making μ C++ development more accessible and efficient within a popular IDE.

The extension will run a fork of the clangd language server in the background, which will utilize our fork of clang to process μ C++ code and handle language-specific features. It supports μ C++ syntax and semantics, enabling users to code without encountering errors on valid μ C++ constructs. The extension identifies issues in real-time by flagging compilation errors and suggests valid code completions, streamlining the development process for μ C++ programmers. Lastly, almost all existing C++ features of the clangd extension are still expected as before even after our changes.

Designed for use within VSCode, the extension is packaged to be easily found, installed, and launched by developers working with μ C++. It requires no additional setup beyond VSCode's standard extension installation process. The scope of this project is to provide essential language server features, without extending to advanced debugging, profiling, or refactoring tools, and does not include support for IDEs beyond Visual Studio Code.

1.3 Terms

μ C++ Extension - The name of our project and deliverable. Named this way since it is a VSCode extension that will act as the interface for all language server features for C++ and μ C++.

VSCode - Shorthand for the Visual Studio Code IDE.

user - The developer who is using the μ C++ Extension to develop in VSCode.

Language Server - a program that provides programming language-specific features like autocompletion and error checking to development tools via an LSP.

Language Server Protocol (LSP) - a standardized protocol that facilitates communication between a language server and development tools to provide language-specific features like code completion and diagnostics.

Clang - an open-source compiler for the C, C++, and Objective-C programming languages

Clangd - a language server that leverages Clang's C and C++ parsing capabilities to provide smart code editing features like autocompletion, code navigation, and diagnostics via LSP

1.4 Abbreviations

SRS - Software **R**equirements **S**pecification

LSP - Language **S**erver **P**rotocol

1.5 References

- [1] P. Buhr and C. Lin, “ μ C++: Concurrency in the Object-Oriented Language C++,” [Online]. Available: [https://plg.uwaterloo.ca/~usystem/pub/uSystem/ \$\mu\$ C++.pdf](https://plg.uwaterloo.ca/~usystem/pub/uSystem/μC++.pdf). [Accessed: Sep. 14, 2024].
- [2] P. Buhr, “GitHub Repository,” [Online]. Available: <https://github.com/pabuhr/uCPP>. [Accessed: Sep. 14, 2024].

2. Overall Description

2.1 Product Perspectives

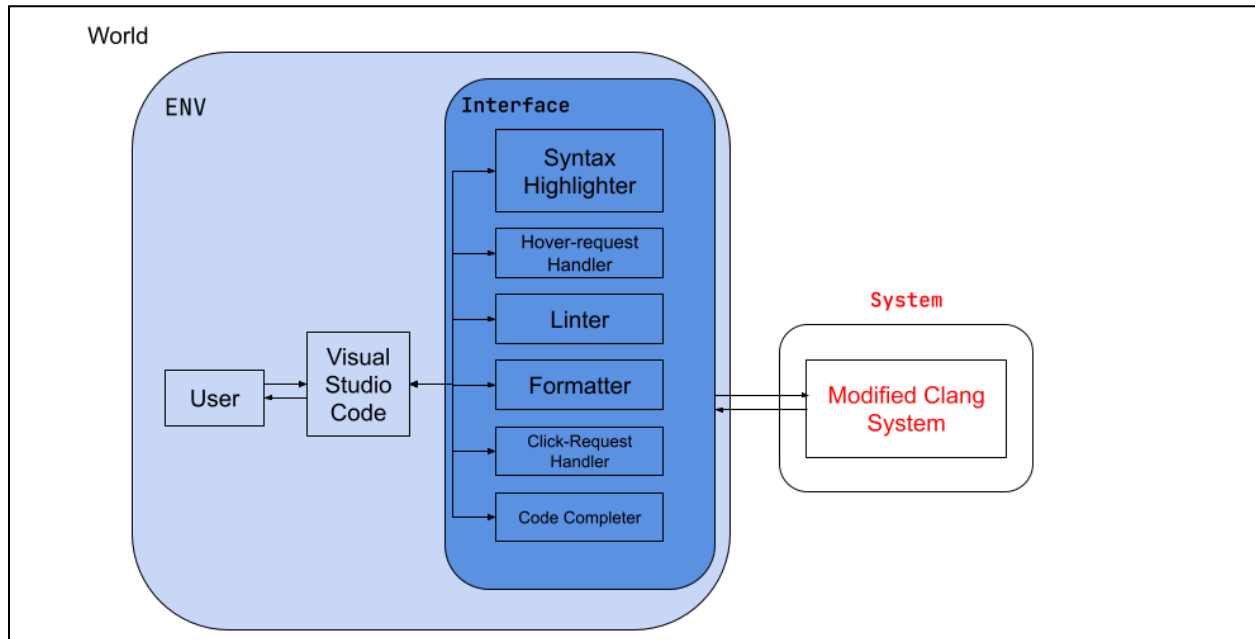


Figure 2.1: Context Diagram

The Extension is designed to function within the Visual Studio Code (VSCode) environment, leveraging a combination of the extension interface, a customized Clang language server (clangd), and a modified Clang compiler. Together, these components provide comprehensive support for $\mu\text{C++}$ language features, including syntax parsing, autocompletion, diagnostics, and navigation.

The VSCode extension serves as the front-facing component installed directly in VSCode, providing the user interface and interactions necessary to support $\mu\text{C++}$ development. It interfaces with VSCode's Language Server Protocol (LSP), allowing $\mu\text{C++}$ functionality to be integrated into the IDE, where users can access autocompletion, error diagnostics, and navigation features.

A customized version of Clangd, the language server component of Clang, will be part of the extension and run in the background. Clangd processes requests from the Extension, such as parsing code, providing symbol information, and managing code navigation functions. Through

the LSP, Clangd communicates analysis and diagnostic data back to VSCode, allowing the extension to display relevant feedback directly in the editor.

A customized Clang compiler component is responsible for parsing μ C++ code according to μ C++ grammar. It provides the core functionality needed to identify errors, generate accurate autocompletion suggestions, and perform syntax checking. By modifying Clang's parsing capabilities to support μ C++, the compiler ensures that all μ C++-specific features are correctly recognized and processed by the language server.

2.2 Product Functions

Highlight μ C++ Syntax

The extension applies syntax-based color coding to μ C++ code in the editor, improving readability by distinguishing keywords, variables, and other elements.

Provide Code/Auto Completion Suggestions

The extension offers context-aware code suggestions as the user types, enabling quick autocompletion of statements and expressions relevant to the current file and project.

Show μ C++ Function Signatures on Hover

Upon hovering over a function name, the extension displays the function signature, including parameter types and return type, for quick reference.

Lint μ C++ Code in the Currently Open File

The extension performs linting, analyzing the code for errors and stylistic issues, and provides warnings or suggestions on hover to assist with code quality.

Navigate to Function/Class Definition (Alt-Click)

Users can alt-click on a function, class, or variable name to navigate directly to its definition within the codebase.

Find All References of a Symbol (Alt-Click)

The extension enables users to alt-click on a symbol (e.g., a variable or function) to locate all references across the codebase.

Apply Formatting/Autoformatting to the Current File

The extension allows for automatic code formatting to the code when the user saves the file or initiates a formatting command, depending on the user's extension configuration.

2.3 User Characteristics

UC-1: Users are expected to have a working knowledge of C++ programming. Familiarity with standard C++ syntax, control structures, and class-based object-oriented programming is assumed.

UC-2: Users have some exposure to or experience with Visual Studio Code, including basic navigation, extension usage, and familiarity with its integrated development environment (IDE) features.

UC-3: Users are assumed to be developers or students using to explore concurrency and advanced control flow constructs, such as coroutines, monitors, actors, and other explicit concurrent constructs not typically found within standard C++.

UC-4: Users may have varying levels of expertise with syntax but should have a basic understanding of its additional keywords and constructs.

UC-5: Users are assumed to have basic technical proficiency, including installing and configuring VSCode extensions and navigating extension settings within VSCode.

UC-6: Basic troubleshooting skills are assumed for common development environment issues, e.g., extension conflicts, version compatibility

UC-7: There should be no training needed, as the extension's purpose is to streamline and enhance development by providing support for -specific keywords and structures.

UC-8: The user interface is designed with intermediate-to-advanced users in mind, minimizing onboarding steps while providing quick access to -specific syntax support.

2.4 Constraints

CO-1: The extension should be compatible with other extensions commonly used in C++ development environments (e.g., C++ Intellisense, GitLens). Parallel operation with other VSCode extensions should not hinder the performance or functionality of any unrelated extensions.

CO-2: The extension should not collect any user data or telemetry.

CO-3: This extension should be reliable enough such that usage of the extension would not disrupt development workflows.

CO-4: The extension must avoid unauthorized access or code execution, ensuring that only files within the active project are affected.

CO-5: The extension should follow the VSCode extension API standards and language specifications, ensuring accurate syntax highlighting and code parsing.

CO-6: The extension should have comparable performance to other similar existing VSCode language server extensions.

2.5 Assumptions and Dependencies

AS-1: The computer hardware running VSCode and the extension should rarely fail or crash.

AS-2: Users are expected to run an updated version of VSCode that supports the API features used by the extension

AS-3: Users are expected to have the necessary permissions to install extensions and run development tools on their systems.

DEP-1: Compatibility relies on stable VSCode and OS versions. Any updates to VSCode API may require adjustments to the extension.

DEP-2: All of the extension's functions depend on the language of .

2.6 Use Case Diagram

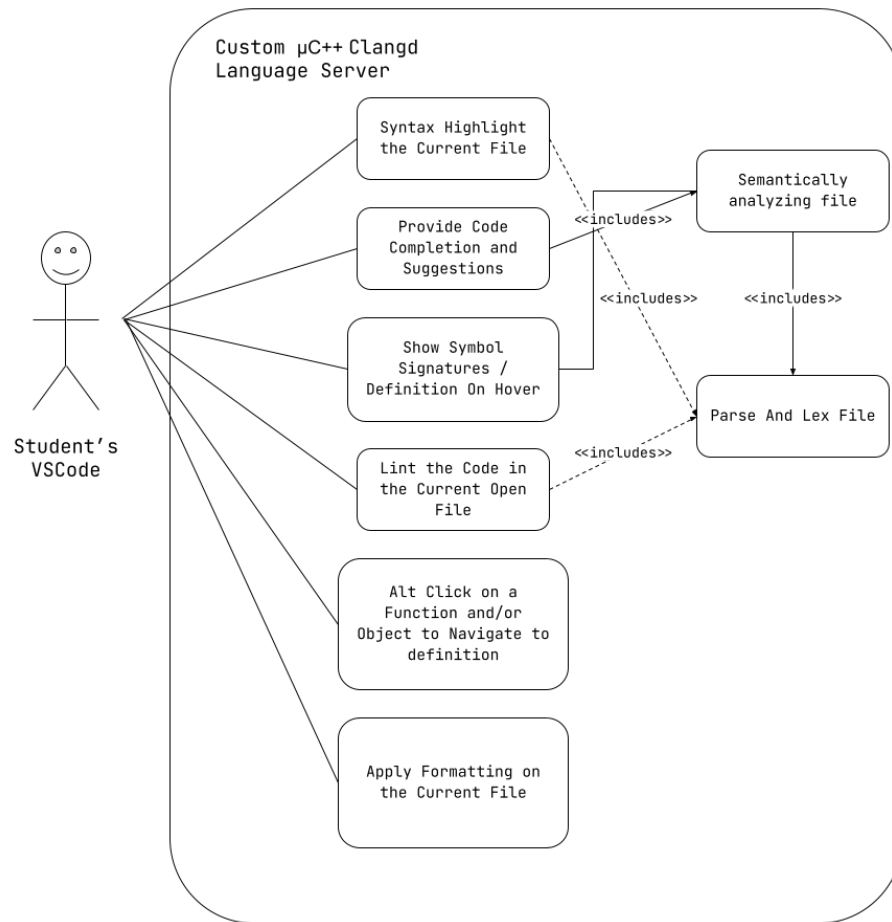


Figure 2.2: Final Use Case Diagram

3. Specific Requirements

For the following use cases, we define typical, alternative, and exceptional scenarios.

3.1 Highlight μ C++ Syntax

3.1.1 Scenarios

1. For the current open μ C++ or C++ file, the extension shall properly parse, lex, and recognize keywords from both C++ and μ C++ in the currently open file
2. The extension shall properly distinguish the different newly added μ C++ keywords, identifiers, literals, and operators according to the μ C++ syntax
3. The extension shall apply color coding in the editor for μ C++ and C++ keywords, including control structures, type declarations, and access specifiers

3.1.2 Alternative and Exceptional Scenarios

1. If the current file contains syntax errors, everything up to the syntax error will still be highlighted. Depending on the error, lines and words following the syntax error may still be highlighted, but will be defaulted to the text highlight colour if they are unable to be properly parsed
2. If the file is not a C++, the extension does nothing.
3. If the file is labeled as a C++ file when it isn't, the extension will still try to parse and highlight what keywords it can, treating everything else as a syntax error and following alternative and exceptional scenario 1.

3.1.3 User Interface



```
float Point::getX() const {  
    return this->x;  
}
```

Figure 3.1: UI for VSCode Syntax Highlighting

To obtain the syntax highlighting shown in Figure 3.1, open a C/C++/ μ C++ file.

3.2 Code Completion

3.2.1 Functional Requirements

1. The extension shall detect the typing activity and identifies potential completion triggers for keywords in realtime.
2. The extension shall parse the surrounding code context to identify relevant suggestions, including μ C++ keywords, functions, variables, and project-specific symbols.
3. The extension shall show a list of suggested completions is displayed in a popup near the cursor for the currently typed word, allowing the user to navigate through options.
4. The extension shall complete the suggested keyword if the user selects the option either from their cursor or by pressing tab.

3.2.2 Alternative and Exceptional Scenarios

1. If the file is not a or C/C++ file, the extension does nothing.
2. If the code uses μ C++ constructs or syntax that the extension does not fully support, the extension will not provide any additional automatic code completion or prediction.

3.2.3 User Interface



Figure 3.2: UI for Code Completion in Action

To obtain the auto-complete pop-up shown in Figure 3.2, start typing a substring of a defined function / class / variable into the opened text file.

3.3 Show μ C++ Symbol Signatures and Definitions on Hover

3.3.1 Functional Requirements

1. The extension shall detect when the user hovers over a function name in μ C++ code and identify the function's signature.
2. The extension shall display the function signature in a tooltip, including the function's parameter types and return type, directly next to the cursor.

3. The extension shall retrieve and display additional details, such as default parameter values if applicable, to provide a complete function signature view.
4. The extension shall allow the tooltip to remain visible as long as the cursor hovers over the function name and hide the tooltip when the cursor moves away.

3.3.2 Alternative and Exceptional Scenarios

1. If the file is not a C/C++ file, the extension does nothing.
2. If the symbol is a linter / syntax error, then no signature and definition can or will be shown
3. If the code uses μ C++ constructs or syntax that the extension does not fully support, the extension informs the user of this information about the symbol.

3.3.3 User Interface

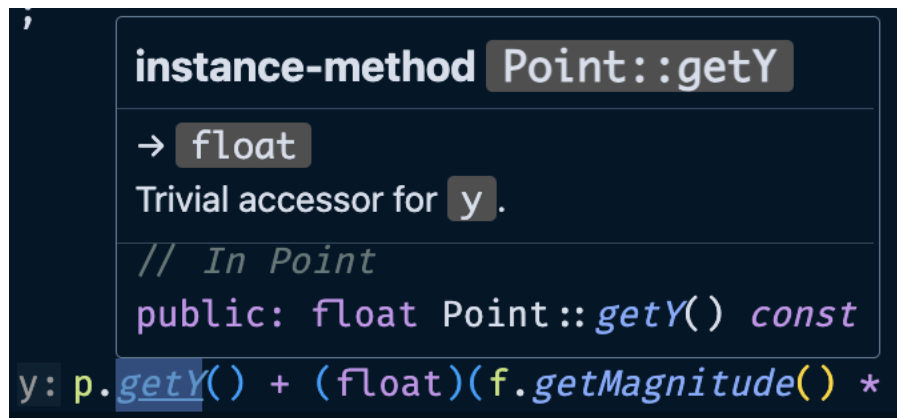


Figure 3.3: UI for Symbol Definition on Hover

To obtain the pop-up shown in Figure 3.3 to view μ C++ symbol definitions/method signatures, hover over the specified symbol with the mouse.

3.4 Highlights Erroneous Syntax

3.4.1 Scenarios

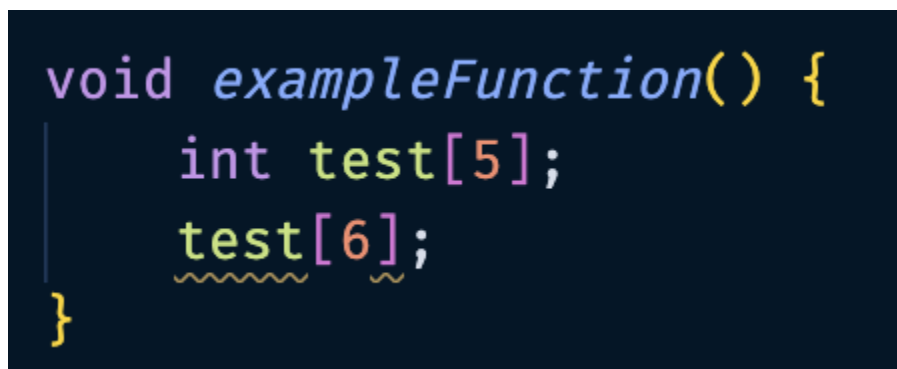
1. The extension shall analyze the currently open file, in real time, for μ C++ syntax errors and unsupported patterns.
2. The extension shall detect errors and warnings specific to μ C++ constructs.
3. The extension shall highlight all detected syntax errors, warnings, and unsupported patterns related to μ C++ constructs and syntax.

4. The user will see this highlighted information and correct the code based on the presented information.

3.4.2 Alternative and Exceptional Scenarios

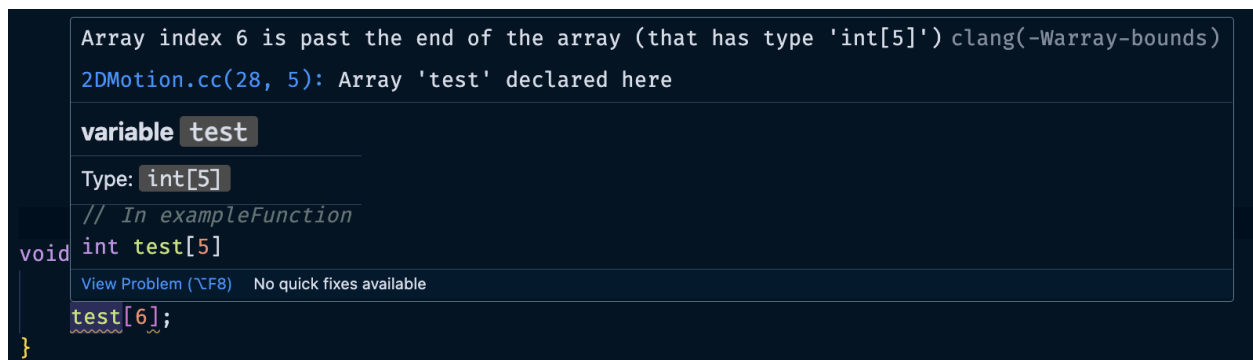
1. If another extension or tool interferes with the μ C++ semantic analyzer, then the extension shall detect this conflict and display a warning, suggesting the user disable the conflicting extension.
2. If the opened file is not labeled as a C++ file, the extension does nothing.

3.4.3 User Interface



```
void exampleFunction() {  
    int test[5];  
    test[6];  
}
```

Figure 3.4.1: UI for Linter Error



```
Array index 6 is past the end of the array (that has type 'int[5]') clang(-Warray-bounds)  
2DMotion.cc(28, 5): Array 'test' declared here  
  
variable test  
Type: int[5]  
// In exampleFunction  
void exampleFunction()  
{  
    int test[5];  
    test[6];  
}
```

Figure 3.4.2: UI for Linter Error on-hover action

To obtain the linter error underline (in yellow) shown in Figure 3.4.1, a static code error / bug must exist in the C/C++/ μ C++ file. To obtain the pop-up shown in Figure 3.4.2 to view linter errors in detail, hover over the specified error with the mouse.

3.5 Alt-Click on a Function / Class To Navigate To Its Definition

3.5.1 Scenarios

1. The extension shall identify the location of a function or class definition within the current project.
2. The user should be able to look up definitions of defined classes and functions within the open program file, using hold alt + click as the shortcut.
3. If the user requests, the extension shall locate definitions across multiple files within the project if the target function or class is defined in a different file than where it's referenced.
4. The extension shall instantly navigate to the identified function or class definition upon look-up, opening the relevant file, and scrolling to the definition's exact line.

3.5.2 Alternative and Exceptional Scenarios

1. If the function or object has multiple definitions due to ambiguity or overloading, the server will instead provide a list of possible definitions, allowing the user to choose the intended one. Once one is selected, the extension will navigate to that definition.
2. If the symbol being alt-clicked on is a syntax error, then the extension will do nothing.
3. If the clicked function or class has multiple definitions, then the extension displays a list of all definitions, allowing the user to select the desired one (e.g., due to overloading)
4. If instead of Alt-Click, the user has configured a different keybinding for this action (e.g., Ctrl-Click or Shift-Click), then the extension respects the user's custom keybinding configuration for navigation.
5. If the Clang binary was manually modified and changed, the extension will still assume that the binary is correct and will respect the errors it reports
6. If the code uses μ C++ functions or classes that the extension does not fully support, the extension informs the user of this limitation and advises manual checks for these components.

3.5.3 User Interface

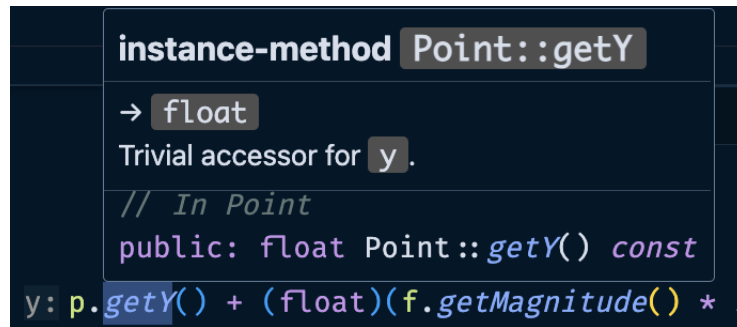


Figure 3.5.1: UI for definition of a highlighted symbol, capable of being navigated to

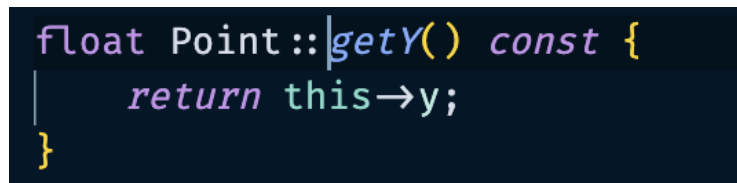


Figure 3.5.2: UI for redirect that occurs when Figure 3.5.1 is when alt-clicked

To redirect from a function / class reference, shown in Figure 3.5.1, to the definition shown in Figure 3.5.2, alt-click (CMD/CTRL-click) on the function / class name.

3.6 Alt-Click on a Variable Gets all References to the Variable

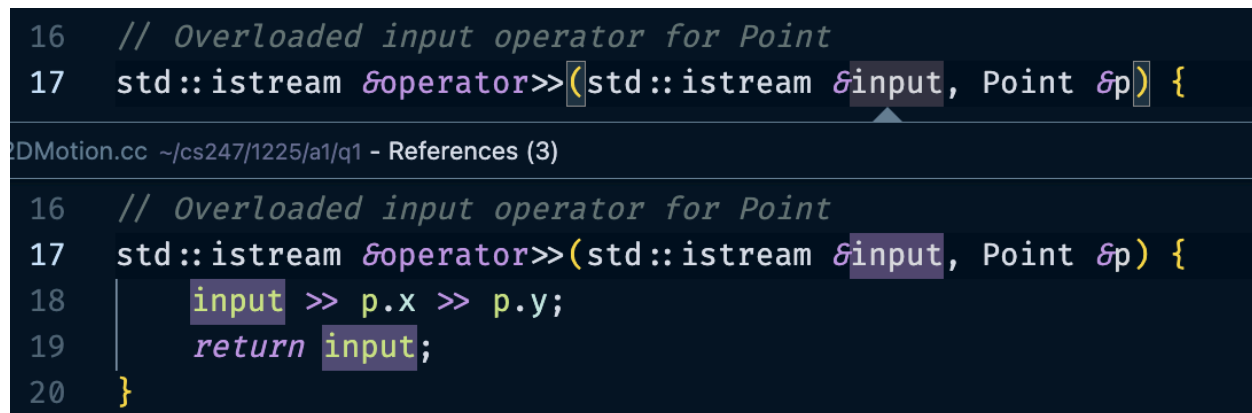
3.6.1 Scenarios

1. The extension shall identify all references to the selected variable within the current file and across other project files, if applicable.
2. The user should be able to search for all references of the variable, using holding alt and left clicking it as the shortcut.
3. The extension shall search for references across multiple files within the project and provide a consolidated list of all occurrences of the variable.
4. The extension shall display the list of variable references in a dedicated panel or overlay, showing file names and line numbers for each occurrence.
5. The extension shall highlight all instances of the variable within the currently open file upon Alt-Click, providing immediate visual feedback in the editor.
6. The extension shall allow users to navigate directly to each reference location by clicking on items in the references list, automatically opening the corresponding file if needed.
7. The extension shall differentiate between variables with similar or overlapping names within different scopes to avoid incorrect references.

3.6.2 Alternative and Exceptional Scenarios

1. If no references are found or the variable cannot be located, the extension shall display a message indicating that no references were found.
2. If instead of Alt-Click, the user has customized the keybinding (e.g., Ctrl-Click or Shift-Click) to find references, then the extension respects this configuration.
3. If there are multiple variables with the same name that exist within different scopes (e.g., within different functions or namespaces), then the extension prompts the user to specify which instance the user wants to search for, or groups references by scope.
4. If the code uses μ C++ constructs or syntax that the extension does not fully support, the extension informs the user of this limitation and advises manual checks for variable references.

3.6.3 User Interface



```
16 // Overloaded input operator for Point
17 std::istream &operator>>(std::istream &input, Point &p) {

DMotion.cc ~/cs247/1225/a1/q1 - References (3)

16 // Overloaded input operator for Point
17 std::istream &operator>>(std::istream &input, Point &p) {
18     input >> p.x >> p.y;
19     return input;
20 }
```

Figure 3.6: UI showing and highlighting all references on a variable that was alt-clicked

To obtain the pop-up shown in Figure 3.6 to view all variable references in the file, alt-click (CMD/CTRL-click) on the variable name.

3.7 Apply Formatting on the Currently Open File

3.7.1 Scenarios

1. The extension shall provide a set of default μ C++ formatting rules (e.g., indentation, spacing, line breaks) aligned with common coding conventions or configurable by the user.
2. The extension shall provide a command in the command palette for running formatting on the current file.

3.7.2 Alternative and Exceptional Scenarios

1. If the user applies formatting by pressing a custom keybinding configured for the action, then the extension formats the file on this keypress.
2. If the user has customized the formatting rules (e.g., indentation size, brace style) in settings, then the extension formats the file according to these personalized settings rather than default μ C++ conventions
3. If the file is exceptionally large, and formatting it may impact performance, the extension warns the user and either prompts for confirmation to continue or suggests limiting formatting to specific sections.
4. If after applying formatting, the user decides to revert to the previous version, the extension provides an option to undo the formatting changes.

3.7.3 User Interface

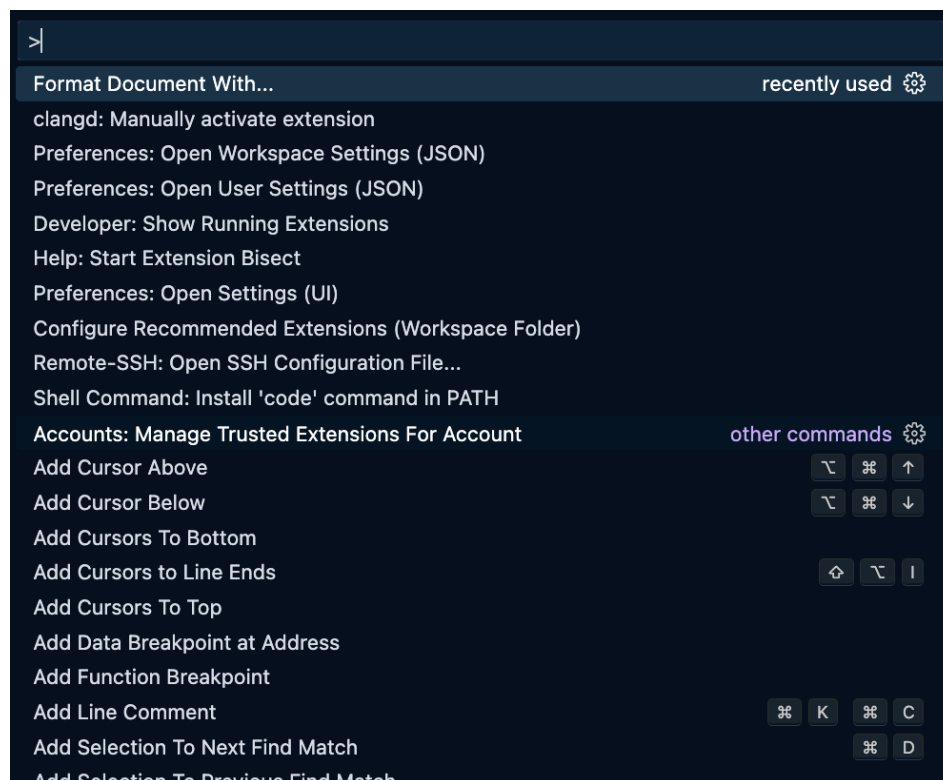


Figure 3.7.1: UI after navigating to VSCode Command Palette
(accessed via CTRL/CMD+SHIFT+P)

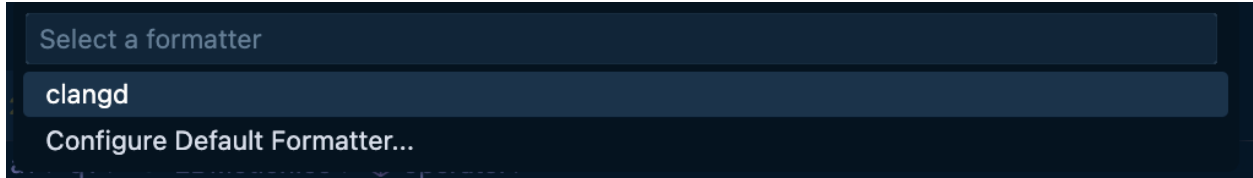


Figure 3.7.2: UI after selecting “Format Document With...” option in Command Palette

To obtain the VSCode Command Palette shown in Figure 3.7.1, press CTRL/CMD+SHIFT+P on the keyboard at the same time. To obtain the menu shown in Figure 3.7.2, click on the “Format Document With...” option from the VSCode Command Palette shown in Figure 3.7.1.

3.8 General Alternative and Exceptional Scenarios

The following are a list of alternative and exceptional scenarios that can occur for all use cases.

3.8.1 Incompatible VSCode Version

In this case, the behaviour of the extension is undefined. If the extension is still able to run, some features may be available while others may be unavailable or erroneous.

3.8.2 Unstable VSCode Extension Host API

If there is an unstable connection between the user’s IDE and the extension’s language server, diagnostics and features will be delayed or unavailable. In the latter case, an error dialog will popup in the bottom right with Clangd’s default timeout message.

3.8.3 Bad Extension Install and Activation

If a bad install or activation results in modified and/or missing files, the extension won’t take any corrective actions and will behave under the assumption that all files are correct.

3.8.4 Insufficient System Resources

If there aren’t enough system resources to run or install the extension, then no action will be taken on the extension’s side and the extension will do nothing.