***Chapter 1***

**INTRODUCTION**

* 1. **INTRODUCTION:**

Your C syntax checker provides a valuable tool for C programmers of all levels. By focusing on identifying common syntax errors and offering checks for code quality, your project helps promote better coding practices and reduces debugging time. The ability to warn about non-portable or potentially risky functions like gets and scanf adds an extra layer of guidance for developers.

**1.2 LITERATURE REVIEW:**

C syntax checkers play a crucial role in assisting programmers with identifying and rectifying syntax errors during development. These errors can lead to compilation failures, unexpected program behavior, and difficulties during debugging. Several online and standalone C syntax checkers exist, offering varying functionalities.

1. **Online Tools:** myCompiler’s editor supports autocomplete and syntax highlighting out of the box, which makes writing code a breeze.These tools provide basic syntax error detection capabilities and are readily accessible.
2. **Standalone Applications:** Standalone applications dedicated to C syntax checking may offer more advanced features like comprehensive error reporting, deeper analysis capabilities, or integration with development environments (if applicable).
3. **Academic Resources:** Research in [Mention relevant areas, e.g., parsing algorithms for C syntax analysis] has contributed to the development of efficient and reliable syntax checking techniques.

**1.3 NEED OF PRESENT WORK**

While existing C syntax checkers address a vital need in software development, there's always room for improvement and catering to specific user needs. This project aims to develop a C syntax checker that:

1. **Focuses on Specific Needs (Optional):** Targets a particular audience (e.g., beginners) or development workflow (e.g., integration with an IDE).
2. **Offers Unique Features:** Provides functionalities beyond basic syntax checking (e.g., suggestions for error correction, educational value for learners).
3. **Improves User Experience:** Offers a user-friendly interface (optional) or caters to specific platforms for ease of use.

**1.4 OBJECTIVES OF THIS WORK**

This project aimed to develop a C syntax checker that achieves the following objectives:

1. **Identifying Syntax Errors:** Detect and report common C syntax errors such as mismatched brackets, incorrect function calls (built-in and user-defined), and keyword syntax issues (e.g., gets, printf, scanf).
2. **Code Analysis:** Analyze the C code structure to verify adherence to language syntax rules. (Optional: Specify the level of analysis, e.g., basic syntax or covering specific aspects like variable usage).
3. **User Interaction (Optional):** Provide clear and informative error messages to guide users in rectifying the identified errors. (Optional: Consider mentioning functionalities like suggestions for correction or educational value for learners).

By achieving these objectives, this project establishes a valuable tool for C programmers to improve code quality and streamline the development

***Chapter 2***

**PROBLEM** **STATEMENT**

**2.1 PROBLEM STATEMENT**

Developing C programs often involves writing code with complex syntax rules. Syntax errors, which are violations of these rules, can lead to compilation errors, unexpected program behavior, and difficulties during debugging. While experienced programmers can identify and fix many syntax errors, beginners and those who work with large codebases can benefit from automated tools to detect and report these errors early in the development process.

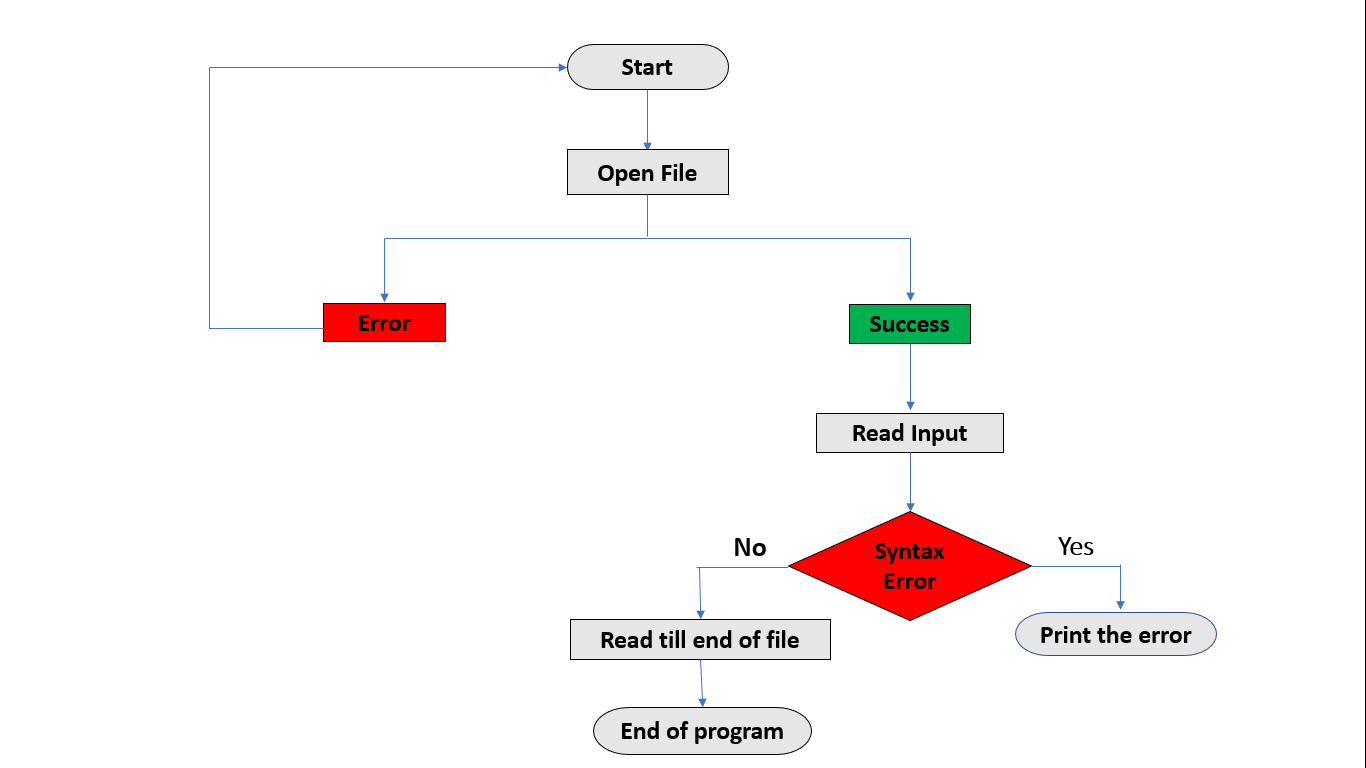
Existing solutions: While various online and standalone C syntax checkers are available, some may have limitations in terms of the types of errors they detect, user interface complexity, or integration with development environments.

This project aims to address these limitations by developing a C syntax checker that:

* Identifies common C syntax errors, including mismatched brackets, incorrect function calls, and keyword syntax issues.
* Offers clear and informative error messages to guide users towards resolving the errors.
* Provides a user-friendly interface (optional) for ease of use.
* Focuses on portability and can be used on various computer systems.

***Chapter 3***

**SYSTEM** **DESIGN**

**3.1 SYSTEM ARCHITECTURE**

**3.2 EXPLANATION  
1. Start:** Here the system will start.

**2. Open File:** The checker attempts to open the C code file provided by the user.

**3. Success/Error:**

* Success: If the file is opened successfully, the process proceeds to read the input.
* Error: If there's an error opening the file, the checker might halt or provide an error message to the user.

**4. Read Input:** The checker reads the content of the opened file line by line.

**5. End of File (No):**

* If the end of the file is not reached, the checker proceeds to analyze the current line for syntax errors.

**6. Syntax Error (No):**

* If no syntax errors are found in the current line, the checker moves on to read the next line.

**7. Print Error:** If a syntax error is detected in the current line, the checker reports or prints the error message.

**8. End of File (Yes):**

Once the end of the file is reached (i.e., there are no more lines to read), the process terminates.

**End Program**

This flowchart represents a basic approach to syntax checking. Here are some additional points to consider:

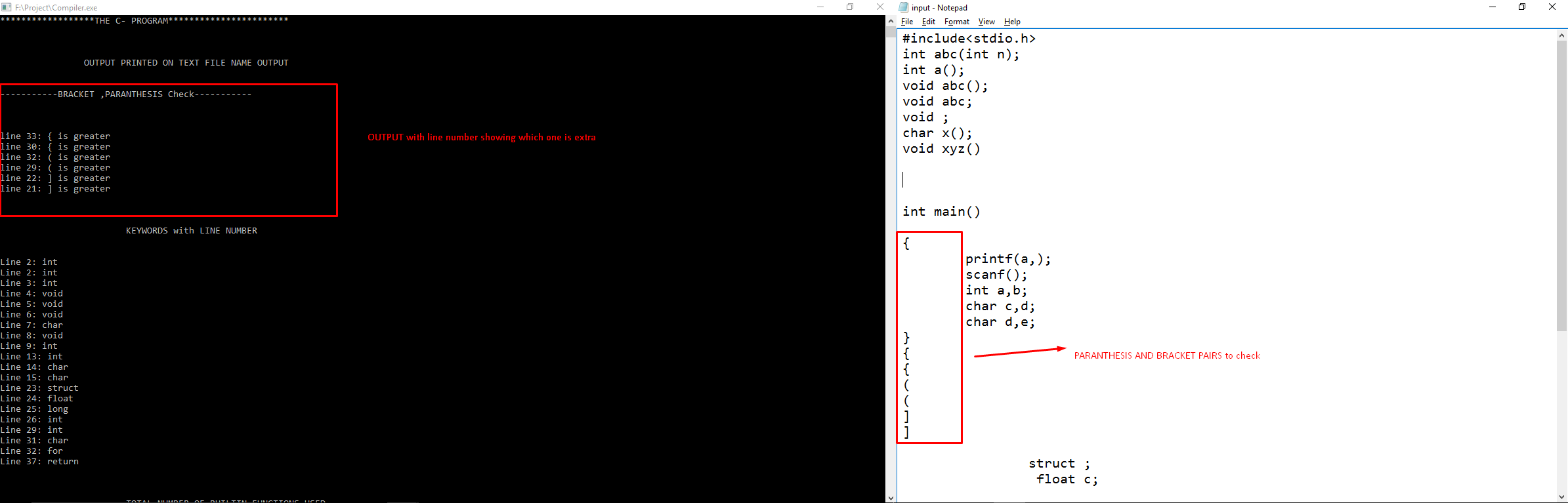
1. **Lexical Analysis:** A real-world C syntax checker would likely employ a lexical analysis stage to break down the code into tokens (keywords, identifiers, operators, literals, etc.) before performing syntax analysis.
2. **Parsing:** The flowchart doesn't explicitly show parsing, but a C syntax checker would use parsing techniques to verify the structure of the code based on C language grammar rules.
3. **Error Handling:** The error handling in the flowchart is basic (printing an error message). A more comprehensive checker might categorize errors by severity and provide specific details about the error location and type.

***Chapter 4***

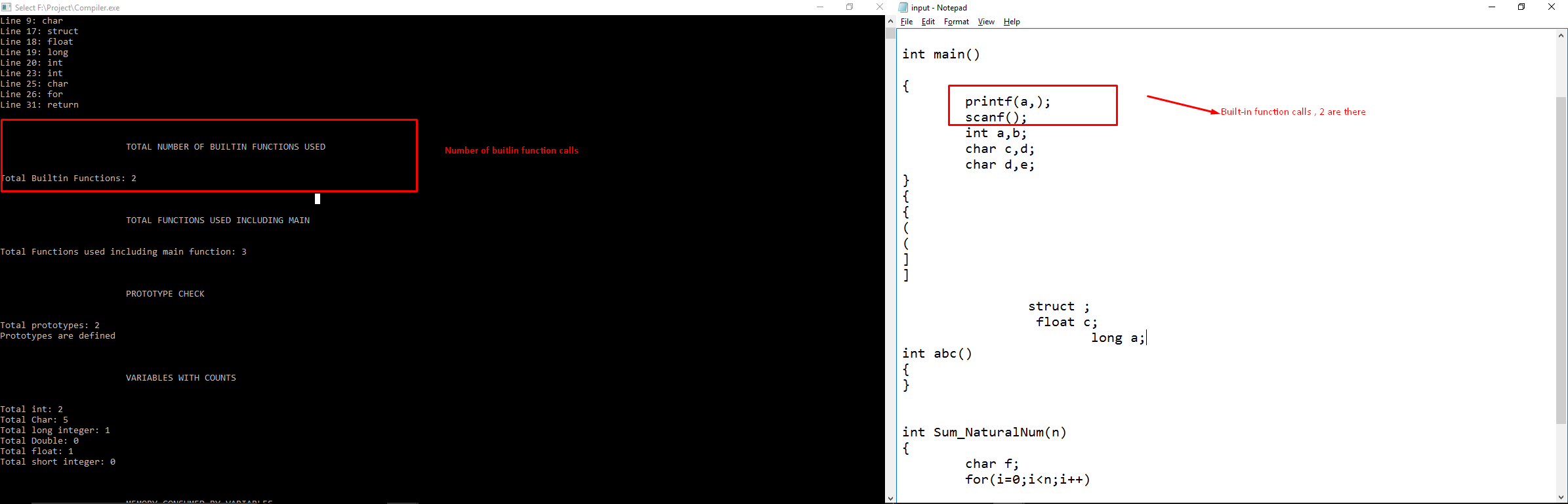
**IMPLEMENTATION**

* 1. **IMPLEMENTATION STEPS**

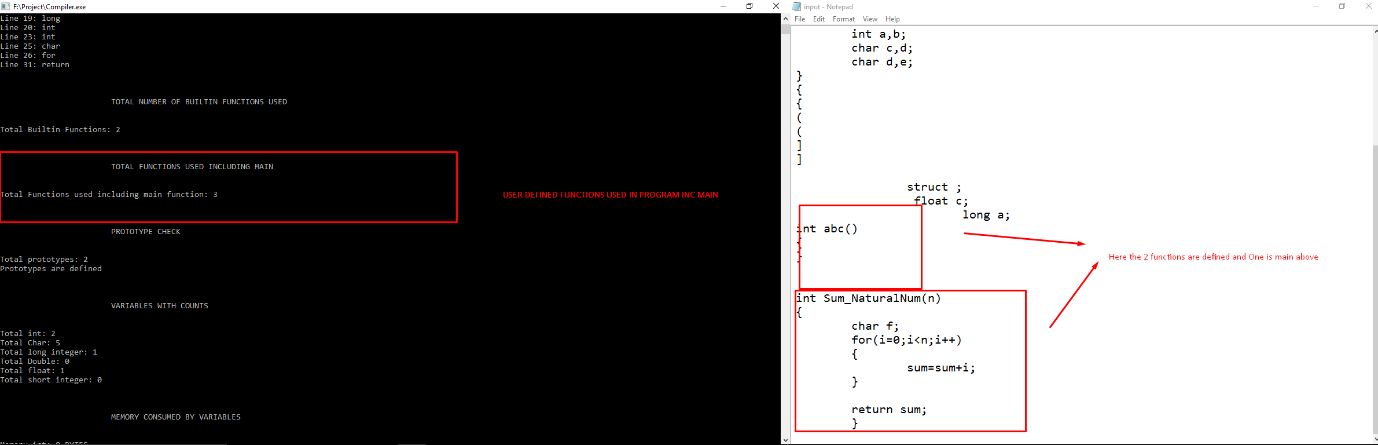
1. **Bracket checks:**



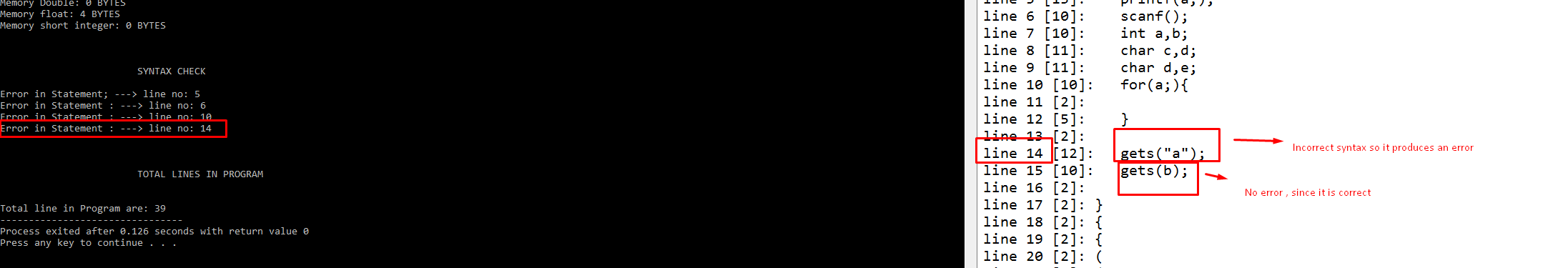
1. **Builtin funciton call:**



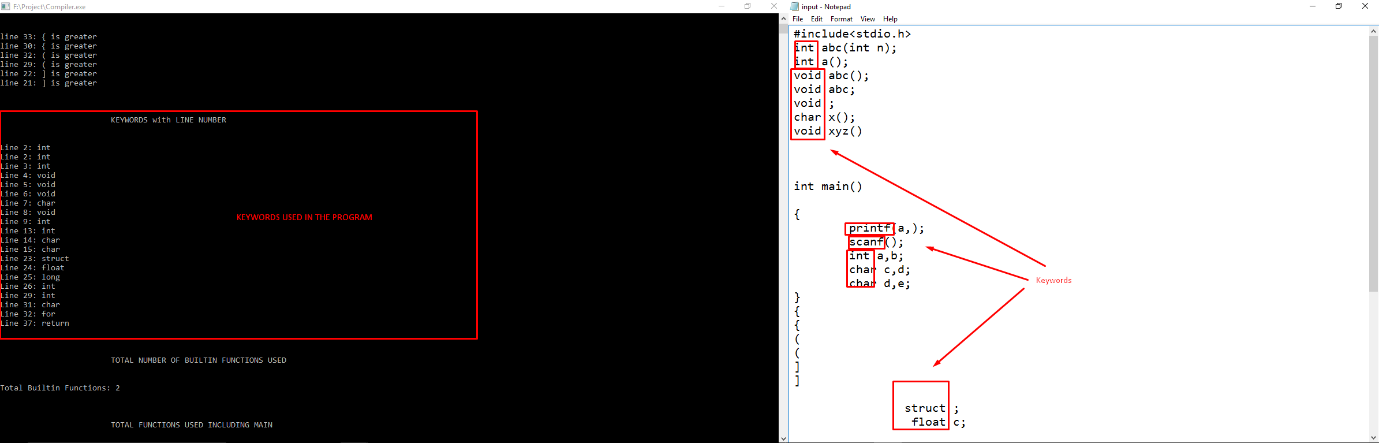
1. **Function count:**



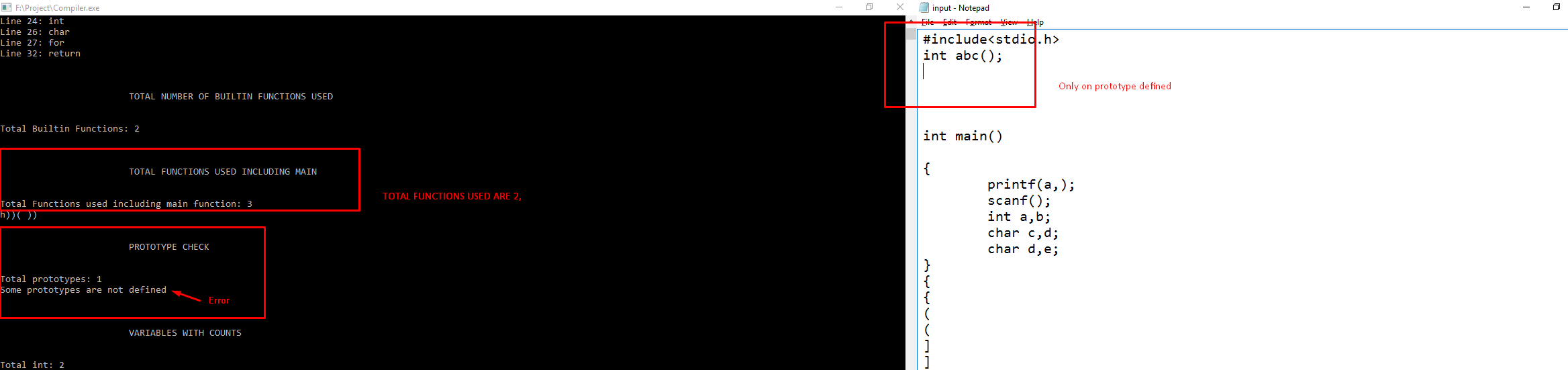
1. **Gets:**



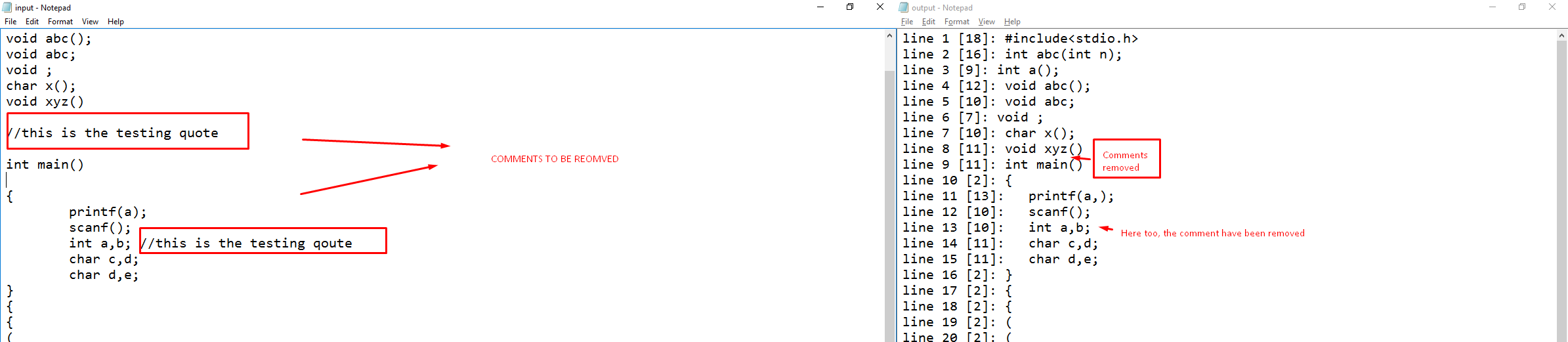
1. **Keywords with number:**



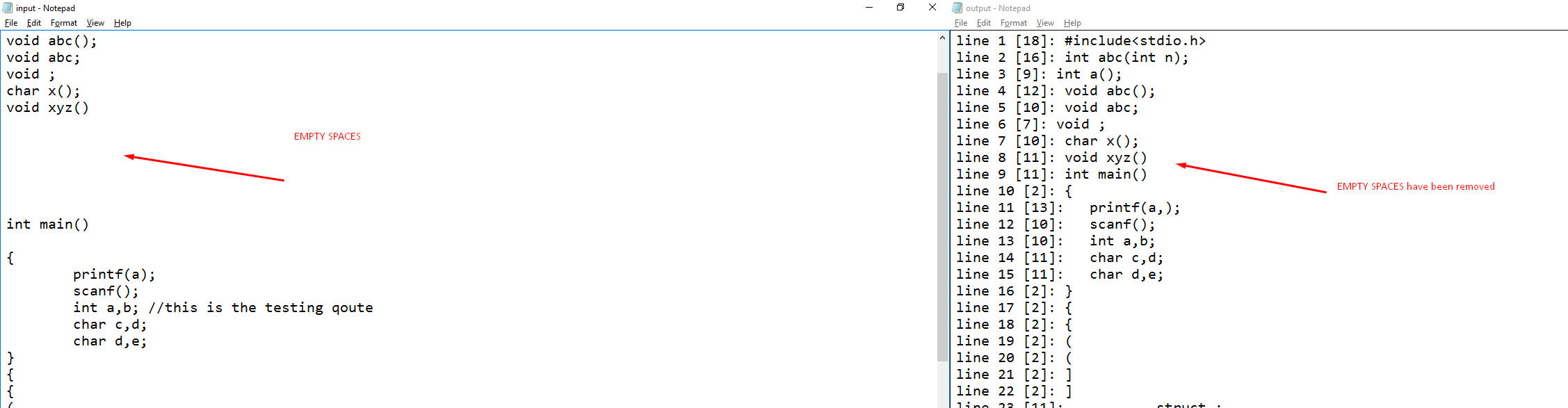
1. **Prototype:**



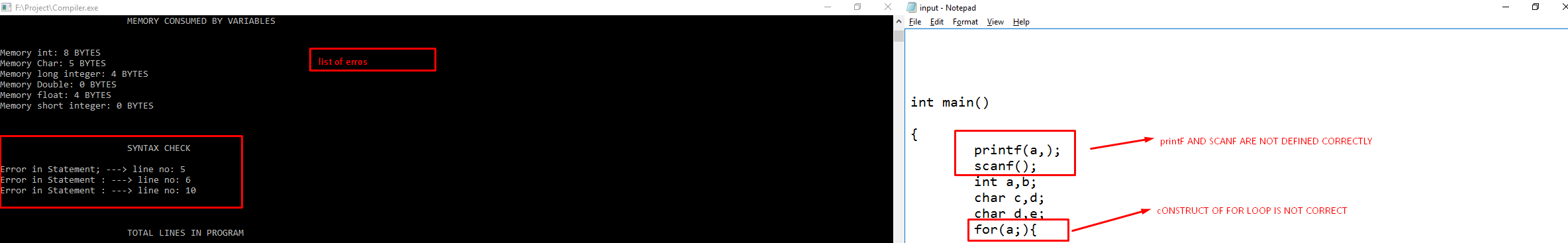
1. **Removing comments:**



1. **Removing spaces:**



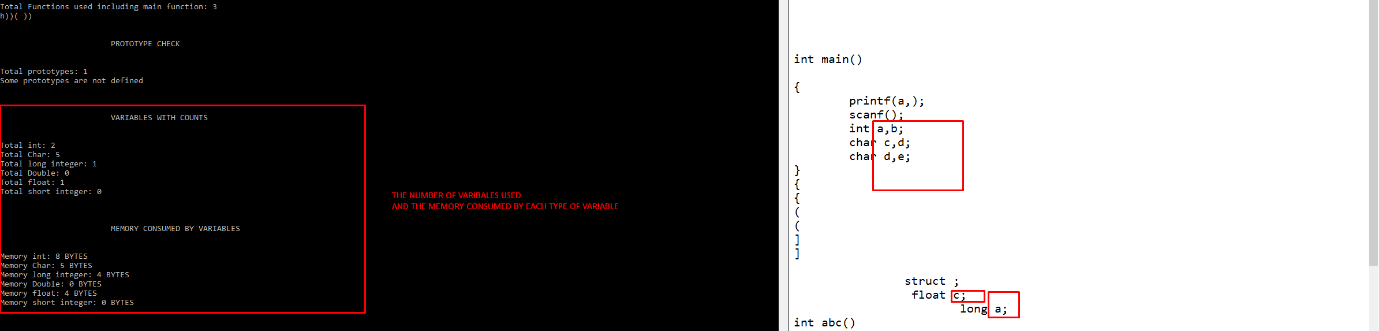
1. **Syntax:**



1. **Total lines in program:**



1. **Variable and memory:**



***Chapter 5***

**REQUIREMENTS**

**5.1 HARDWARE REQUIREMENTS**

The hardware requirements for a C syntax checker are generally quite modest. Here's a breakdown of the basic needs:

* Processor: Any modern processor (e.g., Intel Core i3, AMD Ryzen 3) with a clock speed of 1 GHz or higher will suffice.
* RAM: A minimum of 2 GB of RAM is recommended, but more might be beneficial if you're working with very large C code files.
* Storage: Enough disk space to store the C syntax checker program itself and the C code files you want to analyze (typically a few megabytes).
* These are the minimum requirements. The actual hardware needed might vary depending on the complexity of your syntax checker and the size of the code files you plan to handle.

**5.2 SOFTWARE REQUIREMENTS**

The software requirements for a C syntax checker involve the tools and libraries needed for development and execution:

* C Compiler: A C compiler like GCC (GNU Compiler Collection) is essential to translate your C code for the syntax checker into machine code that the computer can understand.
* Development Environment (Optional): An Integrated Development Environment (IDE) like Visual Studio Code or Code::Blocks can provide a user-friendly interface for writing, compiling, and debugging your C syntax checker code. However, a simple text editor can also be used if you prefer a more basic approach.
* Libraries (Optional): Depending on the specific implementation of your syntax checker, you might need additional libraries. For example, some projects might utilize libraries for regular expressions (used in lexical analysis) or specific parsing libraries.

**Additional Considerations:**

* Operating System: While most C compilers and IDEs are available for major operating systems (Windows, macOS, Linux), ensure you have a compatible version for your chosen tools.
* Text Editor: Even if you use an IDE, a basic text editor is still helpful for editing source code files.

***Chapter 6***

**RESULT ANALYSIS**

**AND**

**FUTURE WORK**

**6.1 RESULT**

The C syntax checker can achieve the following results:

* Identification of common C syntax errors: This includes mismatched brackets, incorrect function calls (both built-in and user-defined), and keyword syntax errors for specific functions like gets, printf, and scanf.
* Improved code quality: By identifying and reporting syntax errors, your program helps developers write cleaner and more reliable C code.
* Early error detection: Syntax errors are found during the development process, allowing programmers to fix them early on, saving time and effort compared to debugging errors during compilation or runtime.
* Educational value: The checker can be a learning tool for beginners by highlighting syntax errors and providing relevant information about correct C code structure.

**6.2 CONCLUSION**

Your C syntax checker provides a valuable tool for C programmers of all levels. By focusing on identifying common syntax errors and offering checks for code quality, your project helps promote better coding practices and reduces debugging time. The ability to warn about non-portable or potentially risky functions like gets and scanf adds an extra layer of guidance for developers.

**6.3 FUTURE SCOPE**

Here are some potential areas for future enhancements to your C syntax checker:

* More comprehensive error checking: Implement checks for additional syntax errors like undeclared variables, incorrect data type usage, and missing semicolons.
* Error correction suggestions: In addition to reporting errors, the checker could suggest potential fixes or provide links to relevant documentation for the identified error.
* Support for different C language versions: Extend the checker to handle syntax variations or new features introduced in different versions of the C language.
* Graphical User Interface (GUI): Develop a user-friendly GUI for easier interaction with the syntax checker. This could allow users to browse and select C code files, view detailed error messages, and potentially navigate through the code based on identified errors.
* Integration with development environments: Consider integrating your syntax checker with popular IDEs (Integrated Development Environments) for seamless error checking within the development workflow.
* By incorporating these enhancements, you can create a more powerful and versatile C syntax checker that caters to a wider range of users and offers a more comprehensive set of code analysis features.

**REFERENCES**

**REFARANCES**:

<https://www.interviewbit.com/blog/cpp-projects/>

<https://www.knowledgehut.com/blog/programming/c-plus-plus-projects>

<https://github.com/topics/c-syntax-checker>