1. What is the purpose of the next() function, and how does it contribute to the compilation process?

   The next() function serves as the lexical analyzer in the C4 compiler. Its primary role is to tokenize the input source code, meaning it reads the source code character by character and converts it into meaningful tokens such as keywords, identifiers, numbers, operators, and more. These tokens are then used by the parser to build the abstract syntax tree (AST) and generate intermediate code. The next() function handles various tasks, including recognizing and categorizing tokens, skipping whitespace and comments, populating the symbol table with identifiers, and detecting lexical errors. It also keeps track of the current line number and position in the source code, which is crucial for error reporting. By converting raw source code into a stream of tokens, next() enables the subsequent stages of the compilation process to work with structured and meaningful data.

2. How does C4 handle **symbol resolution** (e.g., variables, functions)?

   C4 handles symbol resolution using a symbol table, which is a simple array of identifiers. Each identifier in the symbol table stores information such as its name, type, class such as global, local, or function, and value such as memory address or constant value. The compiler uses a hashing mechanism to quickly look up identifiers in the symbol table. For example, when an identifier is encountered, its hash value is computed, and the symbol table is searched for a matching entry. If found, the identifier's attributes such as type and address are retrieved; if not, the identifier is added to the symbol table. C4 distinguishes between global variables, local variables, and functions by assigning them different classes (Glo, Loc, and Fun, respectively). Global variables are stored in the data section of memory, while local variables are assigned offsets relative to the stack frame of their function. This approach allows C4 to efficiently resolve symbols during compilation and ensure that no identifier is declared more than once in the same scope.

3. What are the limitations of C4 as a compiler? What features of C does it not support?

   C4 is a minimalistic compiler designed for simplicity and self-hosting, so it lacks many features found in standard C compilers like GCC. One major limitation is the absence of a

preprocessor, meaning it does not support directives like #include, #define, or #ifdef. Additionally, C4 only supports basic data types like int and char, and pointers, with no support for complex types such as float, double, struct, union, or enum. While it supports pointer arithmetic, it does not support multi-dimensional arrays. The compiler also lacks a standard library, offering only a few built-in functions like printf, malloc, and exit. Furthermore, C4 performs no optimizations, generating straightforward and unoptimized code. Error handling is minimal, with the compiler often exiting on the first error without detailed diagnostics. Scope rules are also limited, as C4 does not support block-level scoping, treating all variables declared inside blocks as function-level locals. Finally, C4 does not support function pointers or higher-order functions. These limitations make C4 unsuitable for real-world C programming but ideal for educational purposes and demonstrating core compiler concepts.

4. How does C4 achieve **self-hosting** (i.e., compiling itself)? What are the implications of this

   design?

   C4 achieves self-hosting by being written in a subset of C that it can compile. This means the C4 compiler can compile its own source code, producing a new executable that is also a C compiler. To achieve this, C4 is first compiled using an existing C compiler such as GCC, and the resulting executable is then used to compile its own source code. This process, known as bootstrapping, demonstrates the compiler's correctness and functionality. The self-hosting capability is made possible by C4's minimalistic design, which uses only the features of C that it supports. This simplicity ensures that the compiler can handle its own source code without requiring unsupported features. The implications of this design are significant. Self-hosting proves C4's correctness and design elegance but limits its real-world practicality due to missing features, though it remains an excellent educational tool for understanding compiler design and bootstrapping.