

INF1002 P9\_2

**Project: Chatbot in C**

Team Members:

| **Name** | **Admission Number** |
| --- | --- |
| Cheng Yi Xing | 2201626 |
| Declan Fong Yi Ren | 2201461 |
| Lan Yiling Elizabeth | 2201982 |
| Lee Ying Zhen | 2201785 |
| Cheryl Toh Huiyi | 2202334 |
| Wong Yu Fei | 2201589 |

**I.** **PROJECT DETAILS**

Websites and other interactive applications are increasingly including chatbots, computer programs that can react to questions input in natural language.

This project aims to construct a straightforward chatbot that can react to straightforward questions composed of a single query word (referred to as the intent) and an object (called the entity). By determining the purpose (where, what, who), entity (printer or C), and answer (matching to this combination), the chatbot will be able to respond to queries like "where is SIT?" and "who is Frank Guan?"

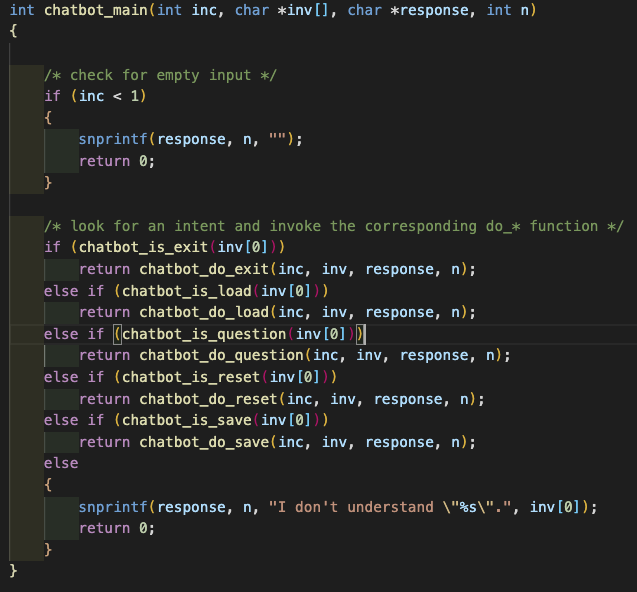
By posing queries to the user, the chatbot will also be able to discover new answers to questions. The opportunity to provide a response that may be utilized to address the same topic in the future will be shown to the user if they ask a question for which the database does not already have an answer. (This mimics having the chatbot forward the unanswered query to a different person who is aware of the solution.)

**II.** **IMPLEMENTATION OF CHATBOT.C FUNCTIONS**

*A. Chatbot\_main function*

*Chatbot\_main* first checks for empty input by the user and continues to prompt the user until the user inputs something to the chatbot.

The first word of the user’s input indicates the intent. If the intent is valid, being “exit”, “load”, “reset”, “save”, “what”, “who”, “where”, it is then passed to its corresponding functions, the bot continues to run accordingly. (eg. *chatbot\_is\_load* -> *chatbot\_do\_load*). If the intent is not recognised, *chatbot\_main* responds with “I don’t understand [intent]”.



*Figure 1.1: Chatbot\_main function codes*

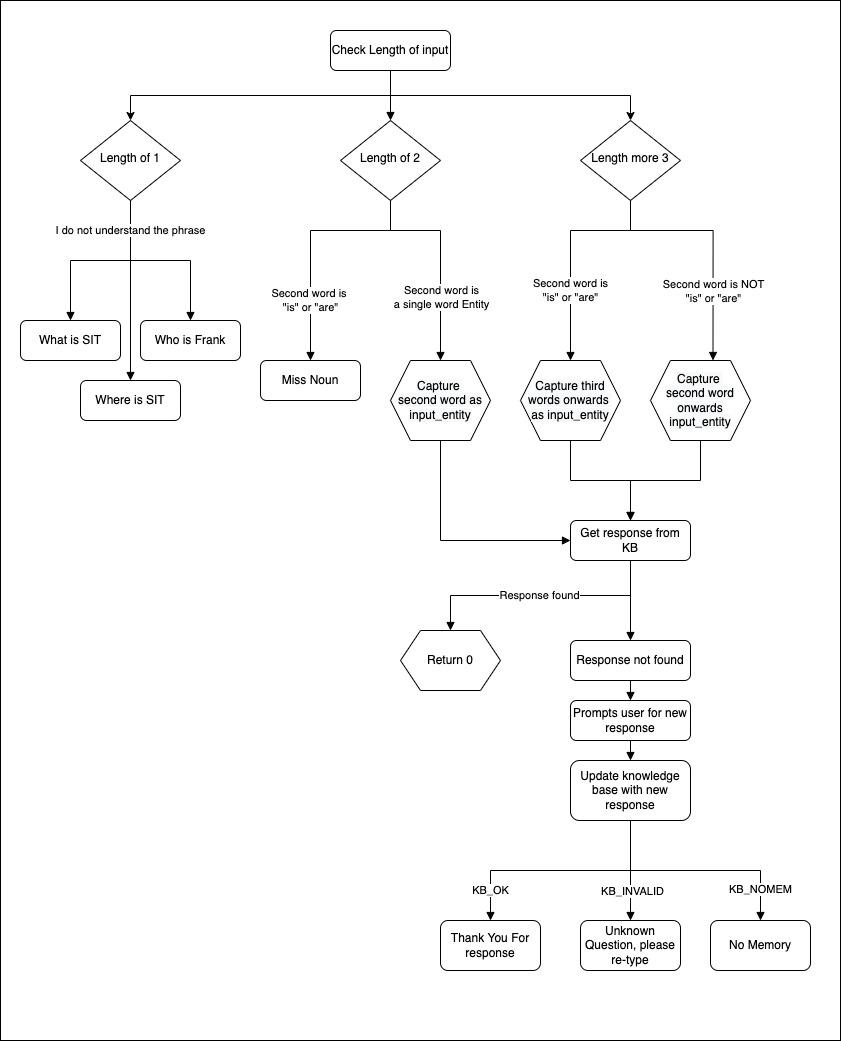
*B1. Chatbot\_is\_question function*

*Chatbot\_is\_question* is called to verify if the intent of the user is to ask a question. *Chatbot\_is\_question* calls the *compare\_token* function to loop through the intent string against “where”, “what” and “who” string individually to compare each character, and return 1 (True) if and only if the intent string matches any of the question string “where”, “what” or “who” exactly.



*Figure 1.2: Chatbot\_is\_question function codes*

*B2. Chatbot\_do\_question function*

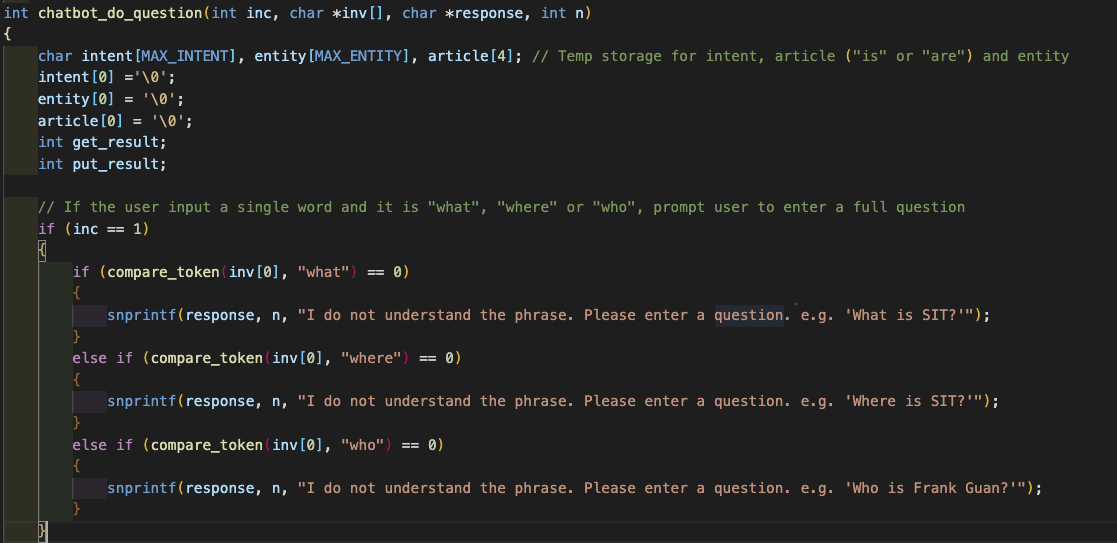
****

*Figure 1.3: Flowchart of Chatbot\_do\_question codes*

*Chatbot\_do\_question* is a function that checks the validity of a user’s input. If input is valid, the function would search the knowledge for the response and if the response is not found, the program will prompt the user for a new response. It first checks for the length of the question the user has input.

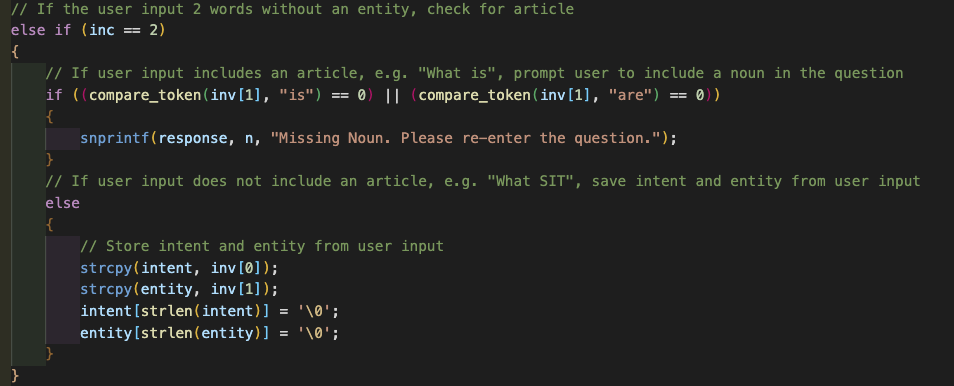
*Case 1: User input has only 1 word*

The user has entered a question that contains only the intent (e.g. “Who”). The chatbot will prompt the user to re-enter the questions providing a possible question that contains the intent.



*Figure 1.4: Chatbot\_do\_question codes, if user input 1 word*

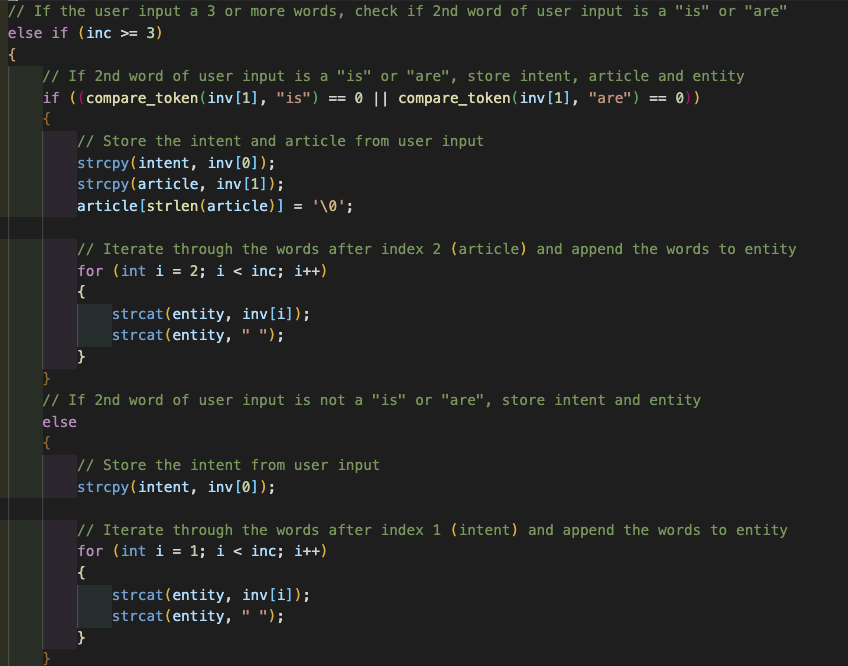
*Case 2: User input has 2 words (2 Scenarios)*

If the second value contains article words like “is” or “are” (e.g. “Who is”), it will prompt the user to enter a noun upon re-typing the question. However, If the second value is not an article word (e.g. “Where SIT”), it will store the second value as the entity.

*Figure 1.5: Chatbot\_do\_question codes, if user input 2 words*

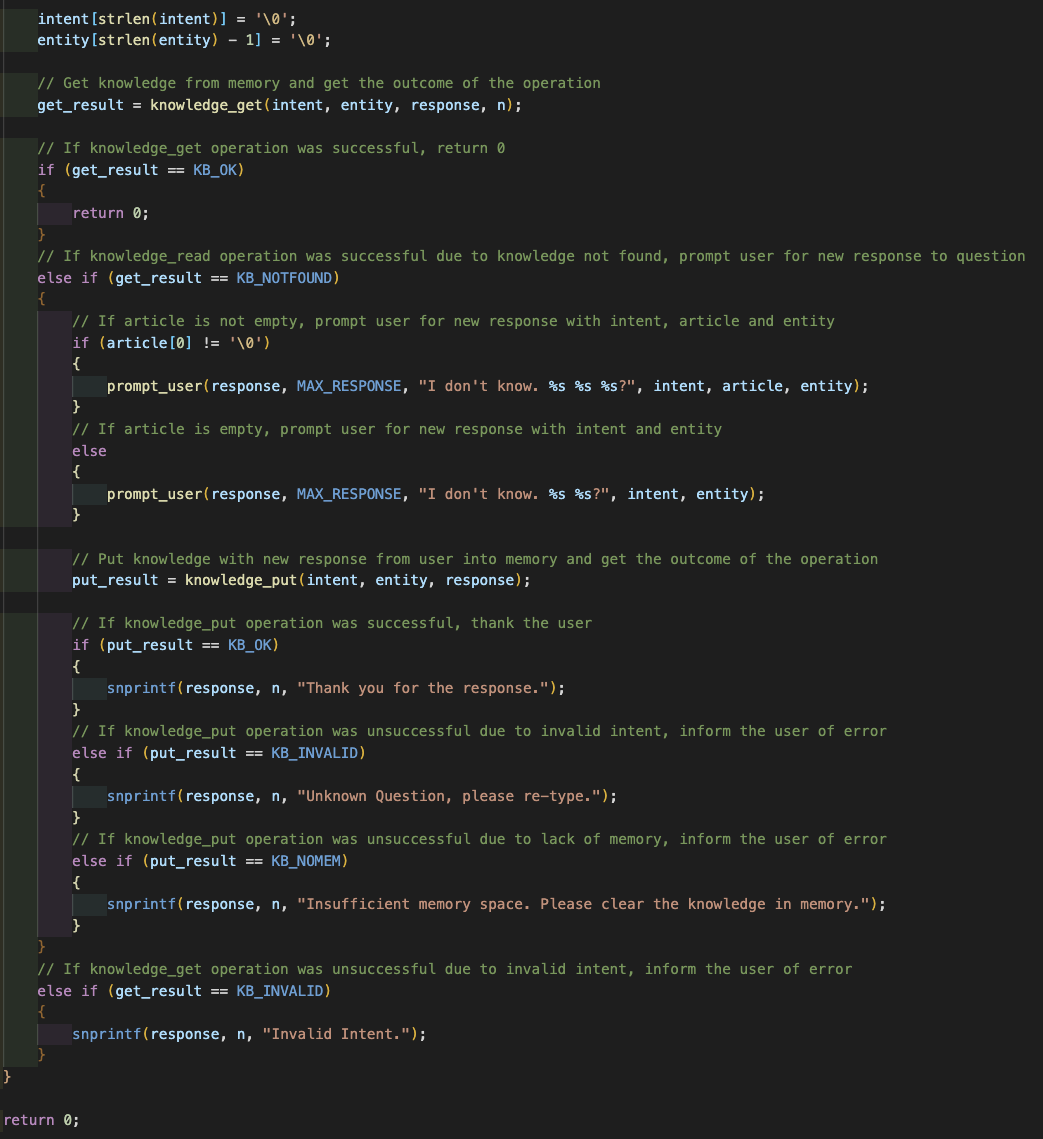
*Case 3: User input has more than 3 words (2 Scenarios)*

If the second value contains an article word (e.g. “Where is SIT”) , it will store the first value as the intent, the second value will be stored as the article and store the third value onwards as the entity. However, if the second value does not contain an article word (e.g. “Where Frank Guan go), it will store the second value onwards as the entity. To store the entity, the program will loop through the input array and concatenate it to get a full string.



*Figure 1.6: Chatbot\_do\_question codes, if user input more than 3 words*

Upon validation, the program will call *knowledge\_get* to check if a question exists. If the function returns *KB\_OK*, it means that a response has been found for the question. If the function returns *KB\_NOTFOUND*, it will prompt the user for a response to the question. When the user provides the response, the program will store the response in a response buffer which will be parsed into the *knowledge\_put* function to update the knowledge\_base. *Knowledge\_put* will return either *KB\_OK*, *KB\_INVALID* and *KB\_NOMEM* and will return the appropriate response to the user for each output.



*Figure 1.7: Chatbot\_do\_question codes, after validation*

*C1. Chatbot\_is\_load function*

*Chatbot\_is\_load* is called to verify if the intent of the user is to “load” knowledge from a certain .ini file into memory. *Chatbot\_is\_load* calls the *compare\_token* function to loop through the intent string and “load” string to compare each character, and return 1 (True) if and only if the intent string matches the “load” string exactly.

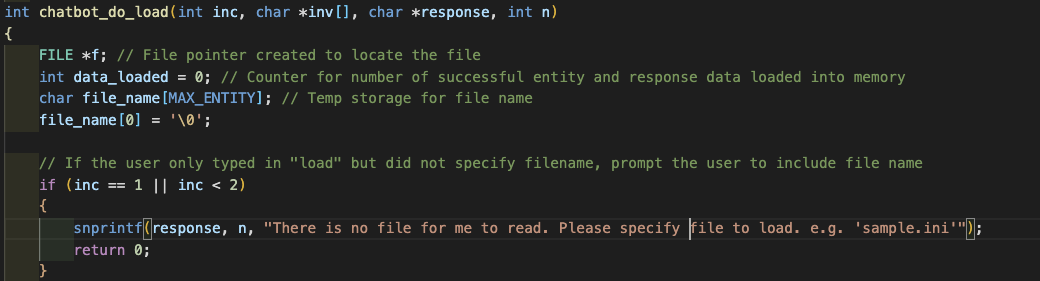


*Figure 1.8: Chatbot\_is\_load codes*

*C2. Chatbot\_do\_load function*

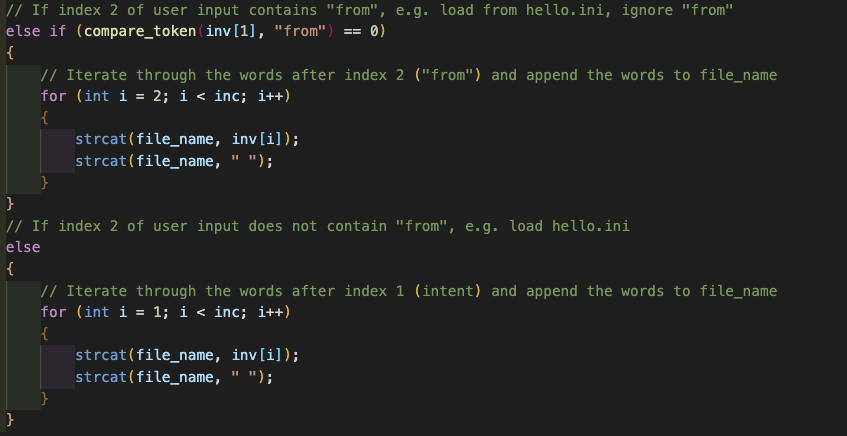
*Chatbot\_do\_load* allows the user to upload an .ini file, pass it to *knowledge\_read* then display the number of data that has been successfully read and loaded.

Before doing so, *chatbot\_do\_load* validates the user’s input. The function first checks the number of words input to the chatbot. If the user only enters “load” without an entity or filename, the number of words will be one. The function then returns an error message prompting the user to specify the exact filename to load.



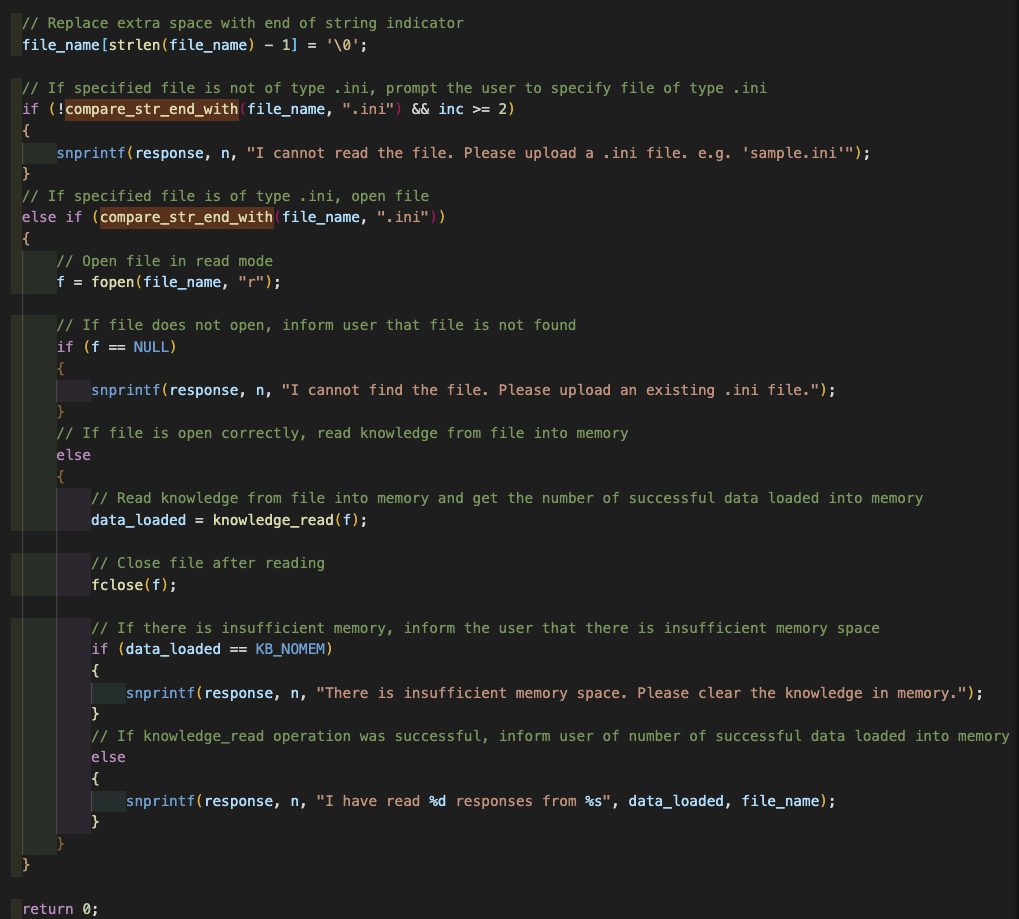
*Figure 1.9: Chatbot\_do\_load codes, validating user input*

Next, the entity is determined via a for loop, starting from the second word. If the user’s input to the chatbot contains “from” (eg. load from hello world.ini) the program ignores that, the for loop then starts from the third word instead.



*Figure 1.10: Chatbot\_do\_load codes, extracting entity*

With the entity determined, the function uses the *compare\_str\_end\_with* function to search the entity for “.ini” to validate the file type. If the file inputted is not of .ini file type, the program returns an error message. Else, the function calls *fopen* which opens the file, and returns a file pointer. If the file pointer points to *NULL*, the file does not exist and an error message is returned. Else, the existing file is passed to *knowledge\_read*, and *chatbot\_do\_load* returns the number of data that has been successfully read and loaded. Finally, an error message is also returned if there is not enough space in the current memory to load the new data completely.



*Figure 1.11: Chatbot\_do\_load codes, loading data into memory*

*D1. Chatbot\_is\_reset function*

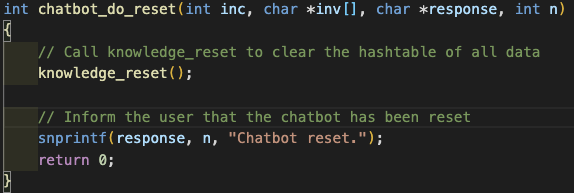
*Chatbot\_is\_reset* is called to verify if the intent of the user is to “reset” the knowledge within the memory of the chatbot. *Chatbot\_is\_reset* calls the compare\_token function to loop through the intent string and “reset” string to compare each character, and return 1 (True) if and only if the intent string matches the “reset” string exactly.



*Figure 1.12: Chatbot\_is\_reset codes*

*D2. Chatbot\_do\_reset function*

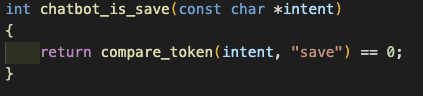
*Chatbot\_do\_reset* resets by calling *knowledge\_reset*. The chatbot then responds with “Chatbot reset.” and returns 0 which means the chatbot can continue chatting.



*Figure 1.13: Chatbot\_do\_reset codes*

*E1. Chatbot\_is\_save function*

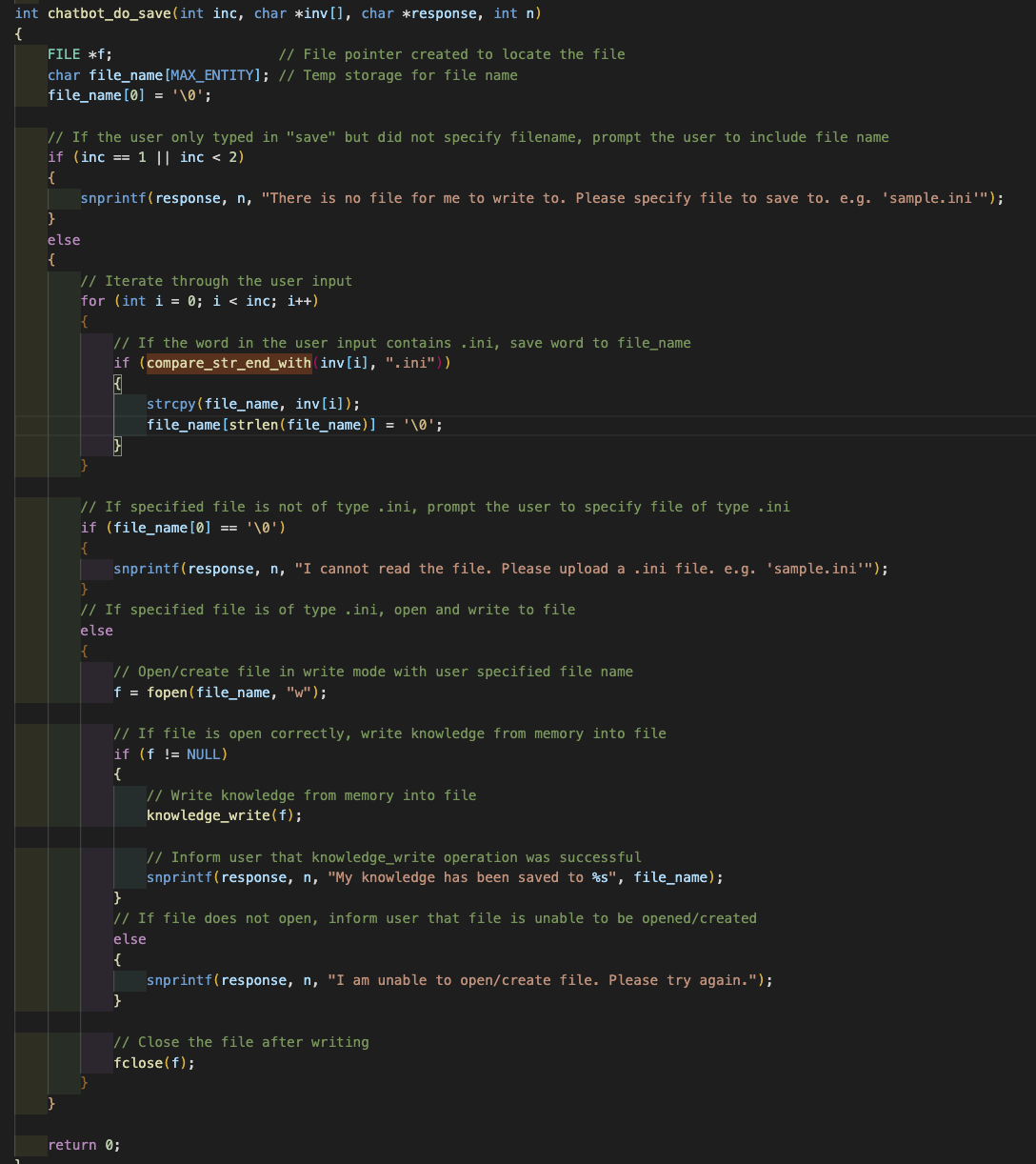
*Chatbot\_is\_save* is called to verify if the intent of the user is to “save” the knowledge in memory to a .ini file. *Chatbot\_is\_save* calls the *compare\_token* function to loop through the intent string and “save” string to compare each character, and return 1 (True) if and only if the intent string matches the “save” string exactly



*Figure 1.14: Chatbot\_is\_save codes*

*E2. Chatbot\_do\_save function*

*Chatbot\_do\_save* is called when the intent is “save”. From the user input, it will extract the word that contains “.ini” using the compare\_str\_end\_with function and save it as a file name. The program will attempt to open a new file with the filename. If the process is successful, the function will call *knowledeg\_write* function to write the data from memory into the file. Chatbot will display “My knowledge has been saved to <file name>” if successful, else, it will display “Unable to save successfully”.

*Figure 1.14: Chatbot\_do\_save codes*

*F1. Chatbot\_is\_exit function*

*Chatbot\_is\_exit* is called to verify if the intent of the user is to “exit” the program. *Chatbot\_is\_exit* calls the *compare\_token* function to loop through the intent string and “exit” or “quit” string to compare each character, and return 1 (True) if and only if the intent string matches the “exit” string exactly.



*Figure 1.15: Chatbot\_is\_exit codes*

*F2. Chatbot\_do\_exit function*

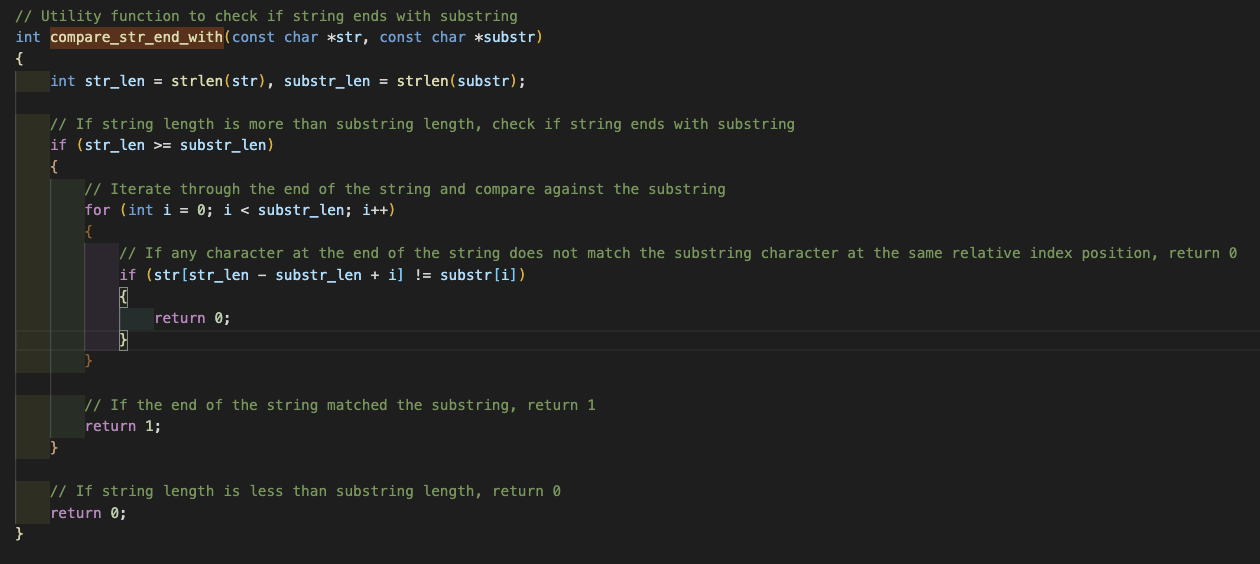
*Chatbot\_do\_exit* is called if Chatbot\_is\_exit determines the intent to be “exit”. *Chatbot\_do\_exit* responds with “Goodbye!” and returns 1 which exits the chatbot.



*Figure 1.16: Chatbot\_do\_exit codes*

*G. Compare\_str\_end\_with function*

*Compare\_str\_end\_with* function checks if a string ends with a substring. It is used in *chatbot\_do\_load* and *chatbot\_do\_save* to check if the filename ends with “.ini”. The function first checks if the string length is more than the substring length. Then, it will iterate through the string to check if any of the characters at the end of the string do not match with the substring character at the same relative position, if it does not match, the function returns 0 to show that the string does not match with the substring. It returns 1 if it does.



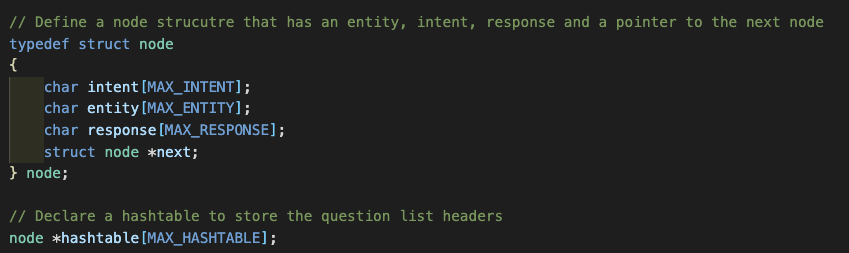
*Figure 1.17: Compare\_str\_end\_with codes*

**II.** **IMPLEMENTATION OF KNOWLEDGE BASE**

*A. Hash Table*

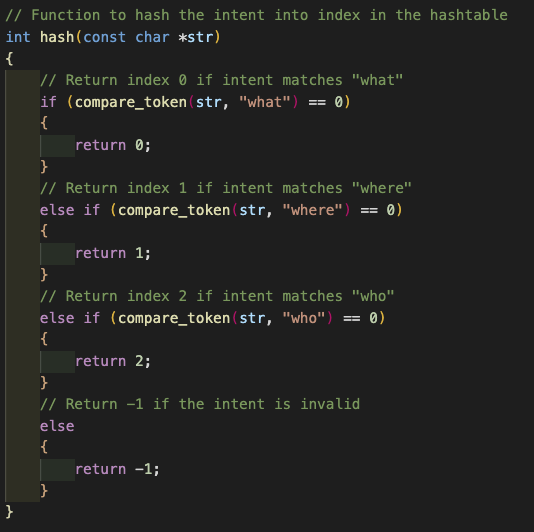
Hash Tables will be used to store the knowledge in the memory. Hash table combines the random access ability of an array with the dynamism of a linked list. This means that insertion and lookup of data can tend towards runtime of O(1).

A node struct is defined for us to create the hash table and the nodes that will exist within the hash table. The node consists of the intent (String), entity (String), response (String) and a next pointer that points to the address of the next node. A hash table of size 3 is declared in order to categorize data into their intent (Who, where, what)



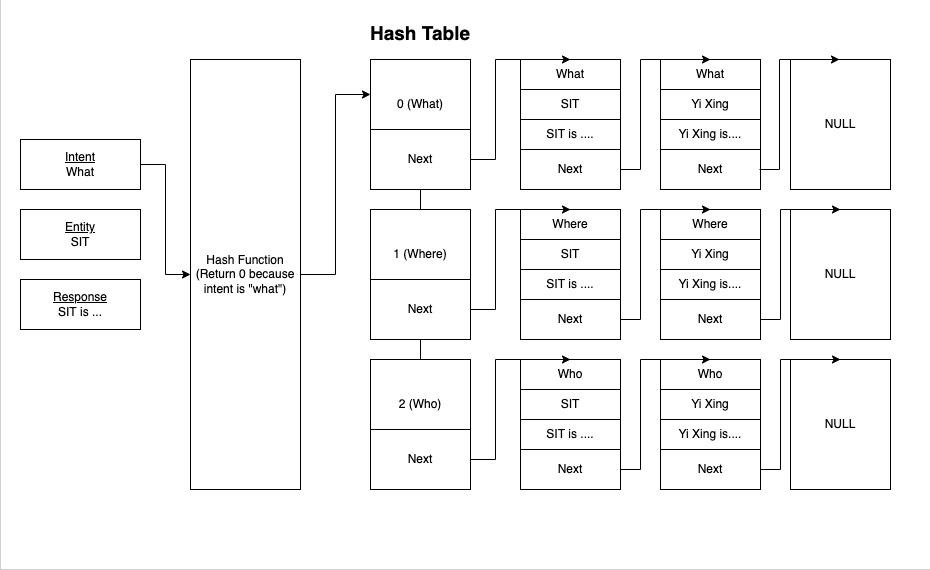
*Figure 2.1: Node Struct and Hash Table Initialization*

The hash table includes a *hash* function which returns a nonnegative integer value called hash code. The idea is that we run our data through the *hash* function, and then store the data in the element of the array represented by the returned hash code.



*Figure 2.2: Hash Function*

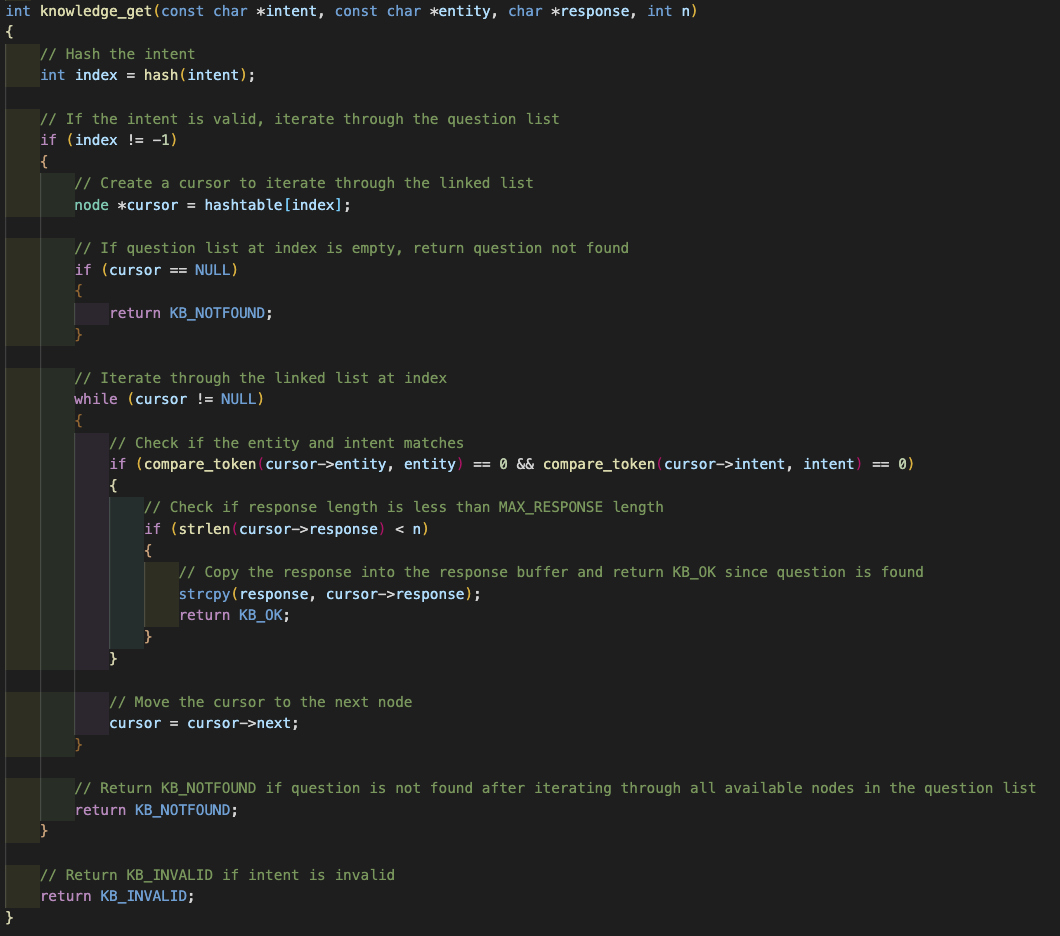
The following chart shows how our data is being run through the hash table and eventually stored within the hash table.



*Figure 2.3: Hash Table Illustration*

*B. Knowledge\_get function*

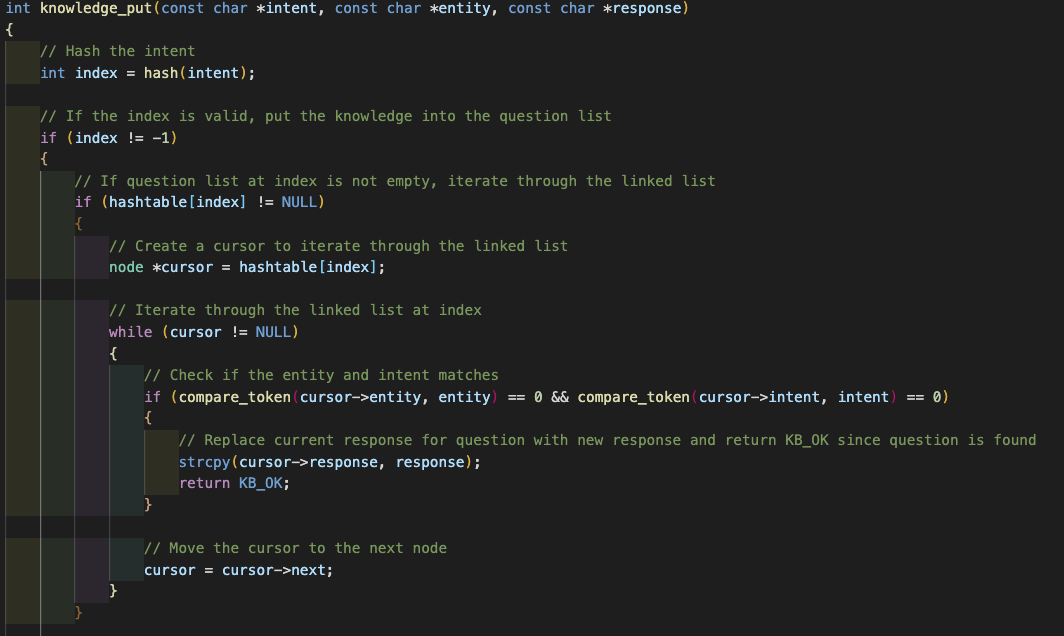
*Knowledge\_get* is a function called by *chatbot\_do\_question* when the program has verified that the user asked a question with the intent “where”, “what” or “who”. It retrieves the response from the hash table based on the intent and entity. The function first gets the index from the hash function in order to determine which index of the hash table array to search the data from. Once the index is returned, the function iterates through the linked list from the hash table index and compares each node to see if the intent and entity matches. If both variables match, it will copy the response variable from the node to a buffer. If a response is not found, the function will return *KB\_NOTFOUND* which means there is no response to the question



*Figure 2.4: knowledge\_get function codes*

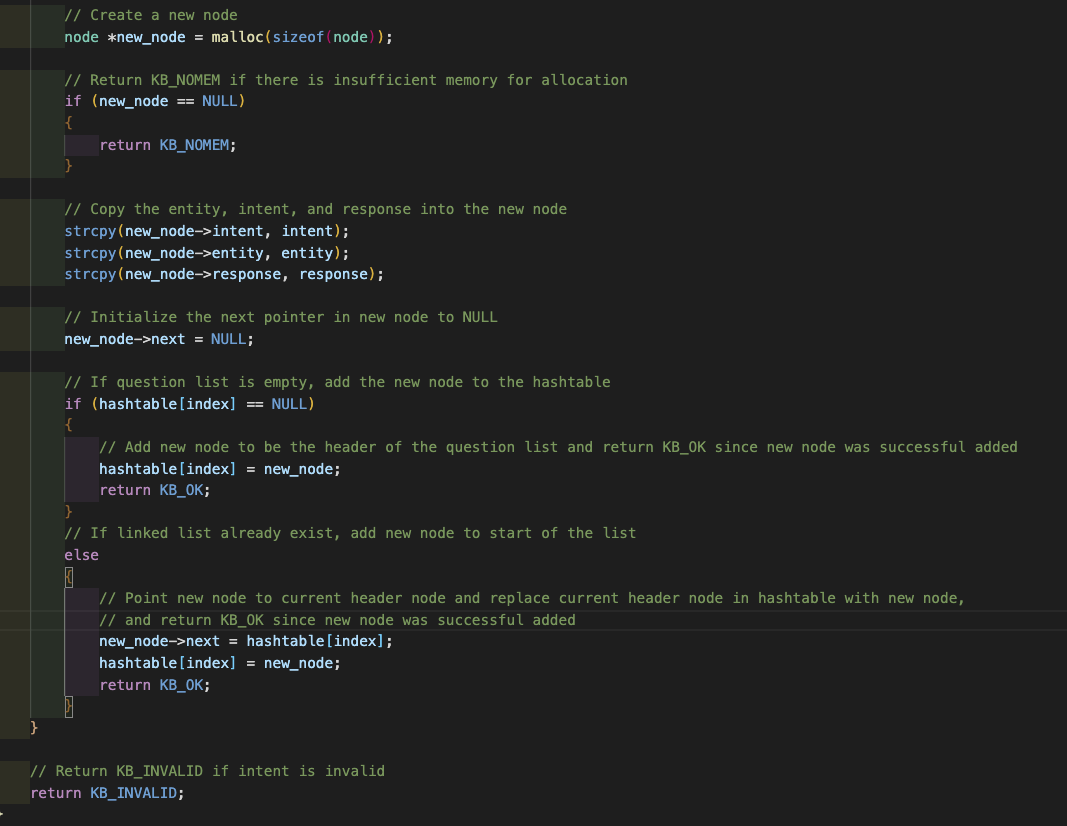
*C. Knowledge\_put function*

*Knowledge\_put* is a function called by *chatbot\_do\_question* when the program is unable to find a response to the question and has asked the user for a response. It inserts a new node with the new response into the hash table. If a response already exists for the given intent and entity, it will be overwritten. The function first gets the index from the hash function in order to determine which index of the hash table array to search the data from. Similarly to *Knowledge\_get*, it traverses the linked list of the return index of the hash table and searches if there exists a matching intent and entity, if it exists, it will change the response variable of the node.



*Figure 2.5: knowledge\_put function codes, replacing response if question exists*

If the function determines that there is no node that matches the intent and entity, it will create a new node with the intent, entity and new response and add the node at the front of the hash table.

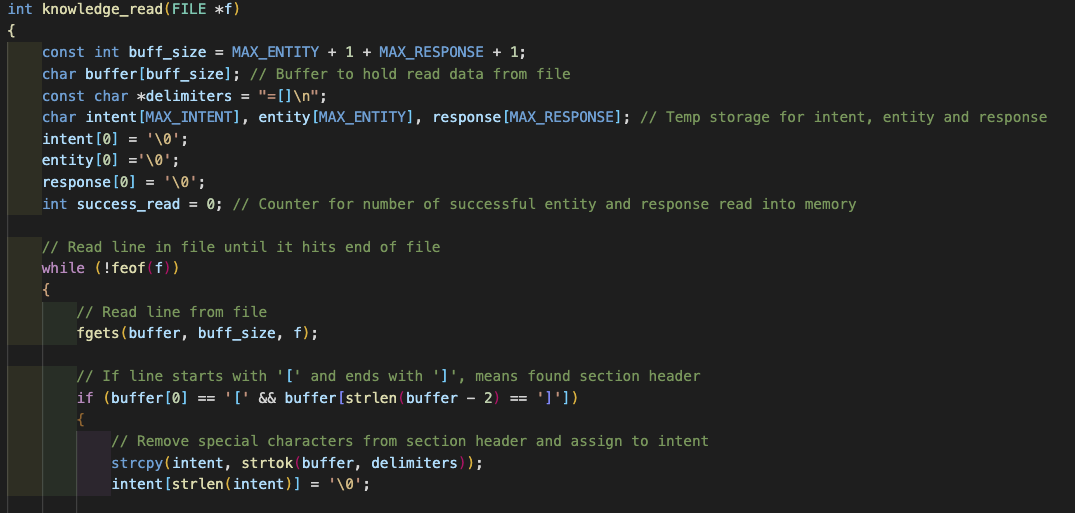


*Figure 2.6: knowledge\_put function codes, adding new node*

*D. Knowledge\_read function*

*Knowledge\_read* is a function called by *chatbot\_do\_load* when the program verifies that the user wants to load data from a file. It reads the given file line by line and stores the information from the lines into the hash table data structure. The function first enters into a while loop that would read the file until it reaches the end of the file. It reads the file line by line and evaluates if the line contains the characters “[“ and “]” to determine if the line is a section header.

If the line is a section header, it would extract the intent from the header by removing special characters and assigning it the variable “intent”, else it would move on to the next line.



*Figure 2.7: knowledge\_read function codes, identifying section headers*

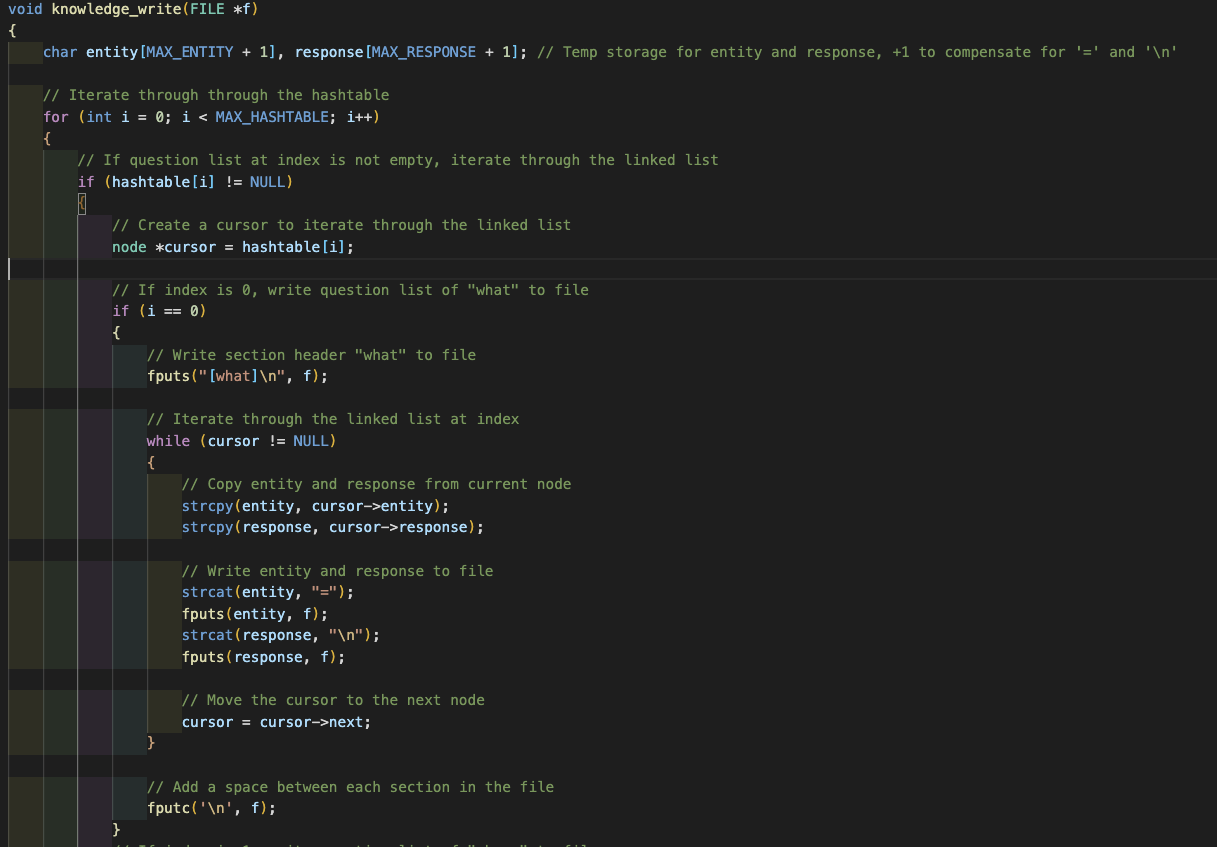
Then, an index is returned by inputting the intent into the hash function, in order to determine which index of the hash table the data should be stored. The program will now look at the subsequent lines below the section header, until it finds a line that is empty or it reaches another section header. The “=” character is checked for each line to determine if there exists an entity and response pair. If the line is valid, the entity and response are extracted and stored in the “entity” and “response” variable. The variables are then used as parameters for the *knowledge\_put* function which will store those variables into the hash table. Once the function returns a successful output, the *success\_read* variable will increment by one. When the function reaches the end of the file, it returns the *success\_read* variable that tells the program how many lines of the file have been successfully stored in the hash table.



*Figure 2.8: knowledge\_read function codes, extracting data line by line*

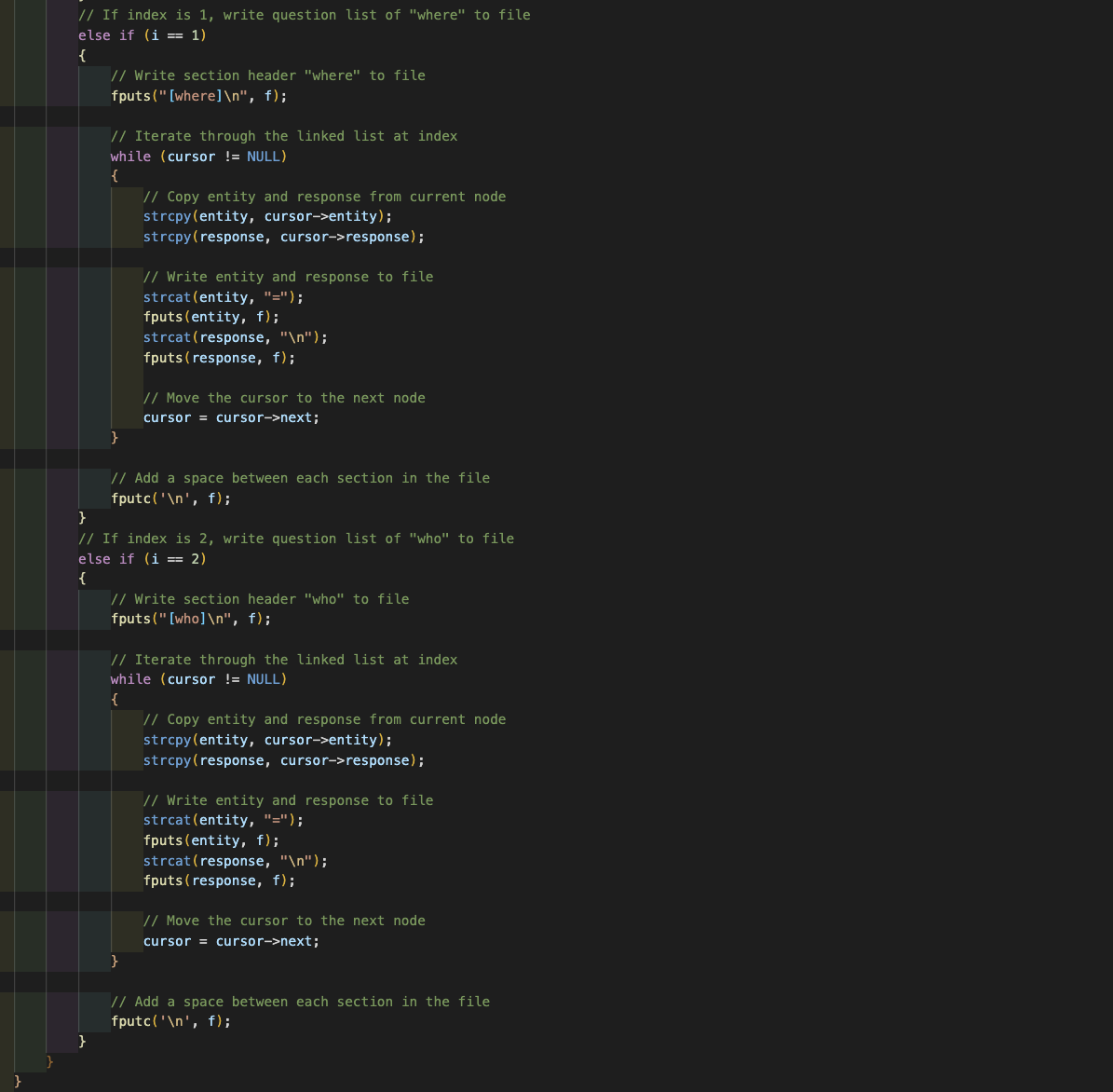
*E. Knowledge\_write function*

*Knowledge\_write* is a function called by *chatbot\_do\_save* when the user wants to save the hash table data into the given file. It reads data from the hash table and writes them into the file line by line. The function first loops through the hash table array and based on the index, writes on the file differently. If the index is “0”, it first writes the line “[what”] into the file. Then, it iterates through the nodes of the linked list at index 0 of the hash table and extracts the entity and response and writes them in the format “entity=response” into the file. Once the function reaches the end of the linked list, the main for loop will move to the next index.



*Figure 2.9: knowledge\_write function codes, writing “what” data*

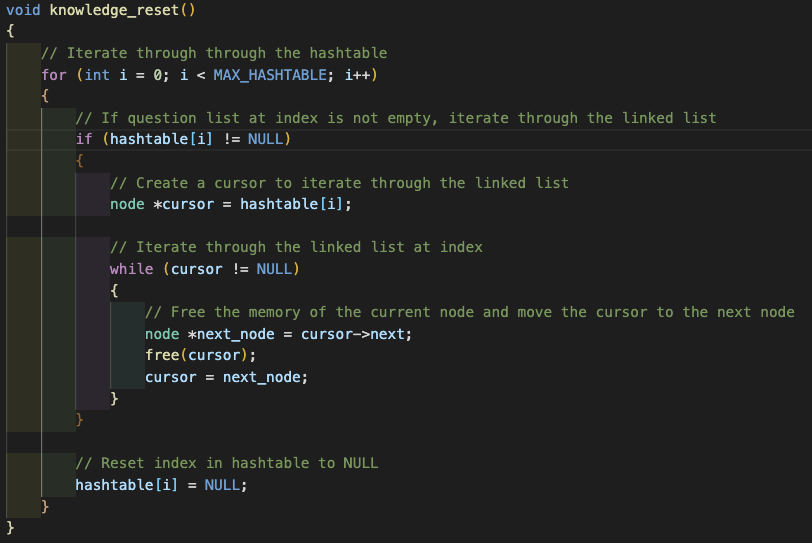
At index “1”, it writes the line “[where]” into the file and traverses the linked list to write the data into the file. At index “2”, it writes the line “[who]” and iterates through the last linked list to write data into the file.



*Figure 2.10: knowledge\_write function codes, writing “where” and “who” data*

*F. Knowledge\_reset function*

*Knowledge\_reset* is a function called by *chatbot\_do\_reset* to reset the knowledge base and remove all known entities from all intents. The function first loops through each index of the hash table in order to free the nodes of each linked list within the index. A cursor variable is initialized to act as the head of the linked list. Using a while loop, the cursor will traverse the linked list. A temp variable is initialized to point to the current cursor node and the cursor is redefined to the next node after the current one. The temp variable is then freed. This method of freeing the data prevents memory leaks. The method of freeing the nodes is applied on index 1 and 2 of the hash table, freeing all the data from the hash table. Lastly, the heads of the hash table are initialized to be *NULL*.



*Figure 2.11: knowledge\_reset function codes*

**IV. CONTRIBUTION**

| **Name** | **Contribution** |
| --- | --- |
| Cheng Yi Xing | Hash table, knowledge\_get(), knowledge\_put(), compiling of report  16.67% of the project |
| Declan Fong Yi Ren | chatbot\_is\_reset(), chatbot\_do\_reset(), knowledge\_reset(), Cleaning of codes  16.67% of the project |
| Lan Yiling Elizabeth | Hash table, chatbot\_is\_load(), chatbot\_do\_load(), knowledge\_get(), Slides  16.67% of the project |
| Lee Ying Zhen | chatbot\_is\_save(), chatbot\_do\_save(), knowledge\_write(), Slides  16.67% of the project |
| Cheryl Toh Huiyi | chatbot\_is\_question(), chatbot\_do\_question(), knowledge\_put(), Slides  16.67% of the project |
| Wong Yu Fei | Hash table, knowledge\_read(), knowledge\_put(), Video, Compiling of codes  16.67% of the project |
| Total: 6 Students | Total: 100% |