Chatbot - JARVIS

ICT1002

Team 49

Date Prepared: 1-12-2021

Prepared By:

| **Matriculation No.** | **Name** | **Email** |
| --- | --- | --- |
| 2100771 | YEO JIE KAI | 2100771@sit.singaporetech.edu.sg |
| 2102909 | CHUA JIA ZHE | 2102909@sit.singaporetech.edu.sg |
| 2102090 | YEONG JING KANG | 2102090@sit.singaporetech.edu.sg |
| 2101738 | MARCUS OH HAN YAO | 2101738@sit.singaporetech.edu.sg |
| 2101290 | ZHENG SHITAN | 2101290@sit.singaporetech.edu.sg |

# 

[**Introduction**](#_7o59d7o9btb7) **2**

[**Implementation**](#_folmu8vy1ve5) **2**

[Main.c](#_mwhakd628ihi) 3

[Compare\_token](#_p99qwsa0p2jf) 3

[Prompt\_user](#_d1ihxat5mftd) 3

[Chatbot.c](#_c0wba05atnow) 3

[Smalltalk](#_7ln7yuky14fe) 4

[Load](#_otxqdpbsjat3) 4

[Question](#_9onczhe166of) 4

[Reset](#_fgjpml68mfqd) 5

[Save](#_6xkpmrd14n8k) 5

[Update](#_d37kt85aed8l) 5

[Knowledge.c](#_3cpih6sp1ayi) 5

[Knowledge\_node\_crud.c](#_klt1ime4bwkr) 6

[Data Structure](#_gu56t6xzw5s8) 6

[Knowledge CRUD](#_6l55yjbsx717) 7

[Save function](#_3qm38kfkb7bv) 7

[Read function](#_xpvighimq9cm) 7

[**Conclusion / Work Distribution**](#_6hhvrsnhky5s) **7**

# 

# **Introduction**

This project revolves around the implementation of a simple chatbot, named JARVIS, developed in the C programming language, that is able to respond to basic queries consisting of an *intent* (question word), an *entity* (Object). This is achieved by identifying the intent (such as *where or what*) and entity (like *SIT or Singapore*), followed by searching the answer that corresponds to them in a database (text file).

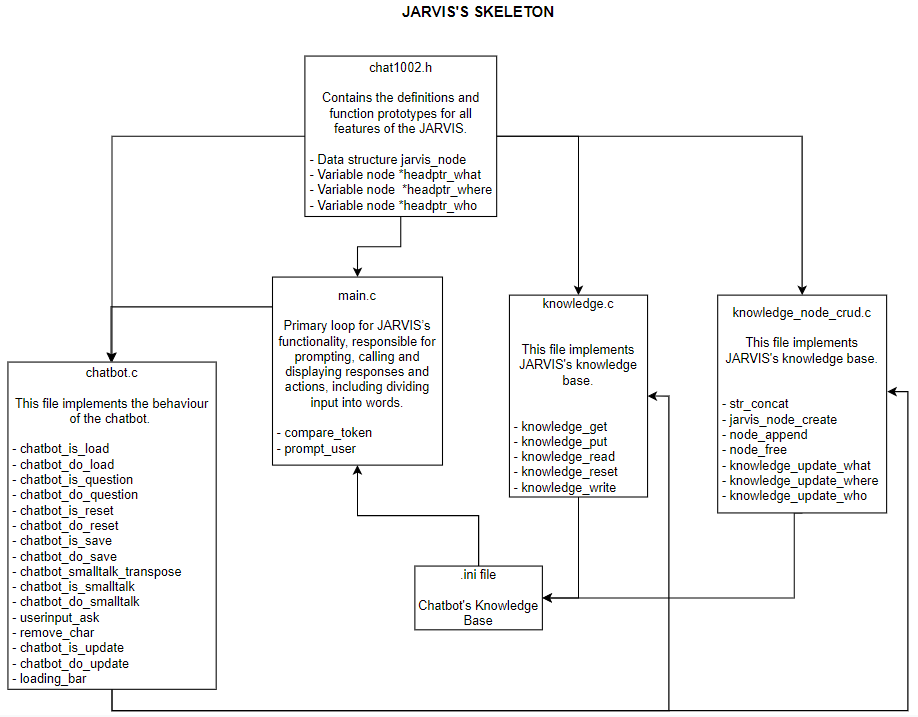
JARVIS is able to learn new answers to questions by querying users for the answers themselves. If the user asks a question for which an answer is not found in the database, the user will be prompted with an option to supply one that can be used to respond to the same question in the future.

JARVIS is also able to “chit-chat” with users, by recognising a limited number of words and or phrases and providing rote replies.

# Implementation

This section details the implementation of the various segments that make up the chatbot, such as the data structures used, and how the Smalltalk, Knowledge Base (KB), and File Operations were designed.

The program consists of the following components:

*Diagram of JARVIS’s components.*

## Main.c

This file is the primary loop for JARVIS’s functionality, responsible for prompting, calling and displaying responses and actions.

### Compare\_token

This acts as a utility function for comparing string case-insensitively, primarily for user inputs. This is done by converting two function variables, strings token1 and token2, to uppercase using toupper(). Afterwards the function compares each character’s ASCII values in the two strings, to return values -1, 0, or 1, similar to strcmp().

### Prompt\_user

This function performs prompting of users for input to questions or responses from JARVIS. It contains error-handling to ensure that the user input does not exceed buffer length and overflow into the next command, by replacing the very last character in the buffer with the terminating symbol.

## Chatbot.c

This file is responsible for processing user intentions, and calling the appropriate sub-functions to generate and return responses based on them. The functions operate based on the following assumptions.

The first word of the sentence indicates the intent. The chatbot\_main() function invokes all the relevant sub-function checkers to determine if the intent is recognized. If it is not, JARVIS will respond with "I don’t understand [intent]." or similar in the form of a question, and ignore the rest of the input.

However, if the second word of the sentence may be a part of speech that makes up in the intent, example:

* If intent is WHAT, WHERE and WHO, it could be "is" or "are".
* If intent is SAVE, it could be "as" or "to".
* If intent is LOAD, it could be "from".

Otherwise, the word is ignored and omitted. The remainder of the input (including the second word, if it is not one of the above) makes up the entity. JARVIS’s answer is stored in the output buffer, and no longer than 256 characters long. The contents of this buffer will be printed by the main loop.

### Smalltalk

To determine if the intent of a user’s input is chat related, the first word of the input is checked against an internal cache of known smalltalk related starting words.

This cache is constructed as a 2D array, to sort and categorize the different possible smalltalk keywords based on similarity. These categories are then referenced via their index, to perform the next level of processing to determine the appropriate response. By storing them this way, it provides an easier way to increase the size of the smalltalk keyword cache, by just adding the keywords to the corresponding category.

There are also additional lists consisting of template responses, to input such as smalltalk questions etc. These are randomized to give JARVIS a more “responsive” and “interactive” feel.

After the initial check for smalltalk, input is then processed through the perspective transposer function. The input entity is checked here for all words that are subject or object pronouns, or in other words, subject words that refer to a particular person. These include “I”, “I’m”, “Me” or “You” amongst others. If they satisfy these criteria, they are transposed / replaced with a word of the opposite meaning, like “You” into “I”.

Finally, the input is then assigned a response based on the initial intent keyword; Some categories such as “I” or “You” will be assigned a random response under the *random\_response* list, while “Hello” will elicit a response from the *random\_greeting* list. There is also a special intent keyword, “time”, where the system time is extracted and displayed as a response to the user.

**Exit**

This sub-function performs the exit procedure for the program, by doing a knowledge\_reset which frees up any allocated memory used by nodes and reset the *what*, *where* and *who* linked lists.

### **Load**

This sub-function is used to load the knowledge base from a file. The file path is obtained from the user input, specifically the 3rd word of the sentence. File contents are parsed through knowledge\_read, which stores them in the linked list data nodes for future questions.

### **Question**

This sub-functions is used to detects if the intent of the user is a

what,where or who and handle the response by verifying if the user input

is in the linked list and storing the data accordingly if it is not found

### Reset

This sub-function resets JARVIS’s knowledge base by calling knowledge\_reset, where all data nodes of the linked list are freed of their contents.

### Save

This sub-function creates a.ini file in which the linked list will be saved,

and if the same file name already exists, it prompts the user to overwrite

the file.

### Update

This sub-function overwrites a response for existing entities by finding the node of matching entity in the link list, the response will be overwritten by users input. If no entities match, it will create a node with the entity and user input as a response and append it into the link list.

## Knowledge.c

This file contains sub-functions to facilitate JARVIS’s knowledge base operations. The sub-functions will examine the intent to see if it matches where, what, or who. If the keyword was previously saved, it would be able to respond appropriately.

The cache is built as a linked list with three sections for each of the following keywords: who, what, and where. These lists are constructed and saved using entity and response, where entity is the keyword, for example, "where is sit?" sit is the keyword in this case. The response would be based on the users' input.

* + 1. **Node\_create & Node\_add**

If the entity and response are empty, they are saved in a node with the help of the create node function and will continue adding on to the linked list with node\_add function, which iterates through the respective intent linked list to find the tail node and append a new one.

* + 1. **Knowledge\_what & Knowledge\_where & Knowledge\_who**

From the nodes linked list It would then transmit the information to the relevant knowledge where, what, and who functions, which would update each linked list appropriately.

* + 1. **Knowledge\_get & Knowledge\_put**

Knowledge\_get will sort out according to which entity (what, where, and person) it loops over the link list and retrieves the answer from the query. If no answer is found, the query is passed down to knowledge\_put where it would add a new response data node to the linked list. If it already exists, it will be overwritten by making use of knowledge\_what, where and who functions.

* + 1. **Node\_free**

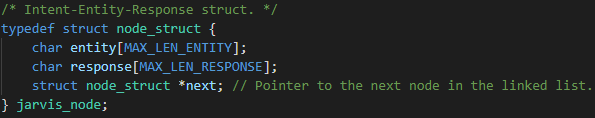
Node\_free is responsible for freeing the node memory from the linked lists, by utilizing the built-in default function free(), to systematically iterate through each linked list via the head node, and freeing each node. This function is implemented in the knowledge\_reset.

## Knowledge\_node\_crud.c

### Data Structure

The technical descriptions should explain what data structures (if any) were used and why these were chosen.

The primary data structures we utilized for this project were linked lists. These are used in the knowledge processing component of JARVIS, such as processing, saving, and updating pieces of knowledge in the database.



*Definition of data structure in program.*

Knowledge is categorized and sectioned based on the different question intent keywords, *Where*, *What* and *Who.* If the input matches these criteria, the input is processed and stripped of delimiters, before being broken down into word-sized and stored in linked lists, accessible via pointers.

These linked lists nodes consist of two variables, the entity and a response of type char, both extracted from the user input. The linked lists are categorized into the 3 intent words, *Where*, *What* and *Who* linked lists. Each list has its own CRUD functions defined in knowledge\_node\_crud.c for knowledge data.

**Intent**

The intent is assumed to be a line of the file that is no longer than the maximum length of 256 + 1 (accounting for the equal sign and new line character) characters, as defined in the header file .

**Entity**

Under each linked list, individual nodes have an entity element to further segment and store possible responses. The entity is referenced when adding or appending new data or responses to the linked lists, to ensure that there is no overlap.

**Response**

The response element in each node comprises the answers given by the user to their own question, assuming that the response or answer was previously not known.

## Knowledge CRUD

### Save function

Knowledge\_write will make use of the link list heads to locate the data to be saved. It will write the intent of the link list before writing from the head of the node which includes information such as entity and response to the last node in the link list. This is done for the link list of every intent.

### Read function

Knowledge\_read will read a given file line by line from top to bottom. The intent will be identified when it reads through the lines. Text after the identified intent follows a format of entity = response. Using the function knowledge\_put, new nodes will be created and added into the link list based on the intent, entity and response read from the file. Until every line of the ini file is read.

* 1. **Error Handling**

### userinput\_ask

This function will receive the user’s input when the program requires a Y/N response from the user. This function will check and reject any responses that are not “Y” or “N” (not case sensitive) and ask the user for the correct input. To prevent fget() buffer overflow, fseek is implemented after the fget() to clear the input buffer.

### remove\_char

This function checks a given string for a specified character. It is primarily used to check the file path strings given in the user input, as the fgets() function is used to retrieve it. The function checks for a trailing “\n” char, an escape sequence that denotes a newline character. This is to ensure that when the string is parsed to other functions in the future, execution errors will be prevented.

# Conclusion / Work Distribution

Smalltalk can be more robust, by implementing more detailed transposition, where the responses given by JARVIS can be constructed using a word key (database) and pieces of the user’s input. This would make the responses more random, life-like.

**Task Allocation:**

| **Task** | **Prepared by** |
| --- | --- |
| **chatbot\_is\_exit**  **chatbot\_do\_exit**  **chatbot\_is\_load**  **chatbot\_do\_load**  **chatbot\_is\_reset**  **chatbot\_do\_reset**  **knowledge\_reset()** | **Marcus** |
| **knowledge\_get()**  **knowledge\_read()**  **knowledge\_write()**  **chatbot\_is\_save**  **chatbot\_do\_save** | **Jia Zhe** |
| **str\_concat()**  **jarvis\_node\_create()**  **node\_append()**  **node\_free()**  **loading\_bar()**  **userinput\_ask()**  **remove\_char()** | **Jie Kai** |
| **chatbot\_is\_question**  **chatbot\_do\_question**  **chatbot\_is\_smalltalk**  **chatbot\_do\_smalltalk** | **Jing Kang** |
| **chatbot\_is\_update**  **chatbot\_do\_update**  **knowledge\_update\_what()**  **knowledge\_update\_where()**  **knowledge\_update\_who()**  **knowledge\_put()** | **Shitan** |
| **Report** | **Shitan**  **Marcus**  **Jie Kai**  **Jia Zhe**  **Jing Kang** |
| **Video Demo** | **Shitan**  **Marcus**  **Jie Kai**  **Jia Zhe**  **Jing Kang** |

**APPENDIX**

**#ifndef \_CHAT1002\_H**

**#define \_CHAT1002\_H**

**#include <stdio.h>**

**/\* the maximum number of characters we expect in a line of input (including the terminating null) \*/**

**#define MAX\_LEN\_INPUT 256**

**/\* the maximum number of characters allowed in the name of an entity (including the terminating null) \*/**

**#define MAX\_LEN\_ENTITY 64**

**/\* the maximum number of characters allowed in a response (including the terminating null) \*/**

**#define MAX\_LEN\_RESPONSE 256**

**/\* return codes for knowledge\_get() and knowledge\_put() \*/**

**#define KNOWLEDGE\_FOUND 200**

**#define KNOWLEDGE\_NOTFOUND 404**

**#define KNOWLEDGE\_INVALID 500**

**#define KNOWLEDGE\_MEMALLOC\_FAIL 501**

**/\* Intent-Entity-Response struct. \*/**

**typedef struct node\_struct {**

**char entity[MAX\_LEN\_ENTITY];**

**char response[MAX\_LEN\_RESPONSE];**

**struct node\_struct \*next; // Pointer to the next node in the linked list.**

**} jarvis\_node;**

**/\* Main.c defines the functions. \*/**

**int compare\_token(const char \*token1, const char \*token2);**

**void prompt\_user(char \*buffer, int n, const char \*format, ...);**

**/\* Chatbot.c defines the functions. \*/**

**const char \*chatbot\_botname();**

**const char \*chatbot\_username();**

**int chatbot\_main(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_exit(const char \*intent);**

**int chatbot\_do\_exit(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_load(const char \*intent);**

**int chatbot\_do\_load(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_question(const char \*intent);**

**int chatbot\_do\_question(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_reset(const char \*intent);**

**int chatbot\_do\_reset(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_save(const char \*intent);**

**int chatbot\_do\_save(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_smalltalk(const char \*intent);**

**int chatbot\_do\_smalltalk(int inc, char \*inv[], char \*response, int n);**

**int chatbot\_is\_update(const char \*intent);**

**int chatbot\_do\_update(int inc, char \*inv[], char \*response, int n);**

**/\* Knowledge.c defines the functions. \*/**

**int knowledge\_get(const char \*intent, const char \*entity, char \*response, int n);**

**int knowledge\_put(const char \*intent, const char \*entity, const char \*response);**

**void knowledge\_reset();**

**int knowledge\_read(FILE \*f);**

**void knowledge\_write(FILE \*f);**

**/\* Knowledge\_node\_crud.c defines the functions. \*/**

**void str\_concat(char \*ptr\_dest, char \*ptr\_src[], size\_t ptr\_src\_size, size\_t n, int offset);**

**jarvis\_node \* jarvis\_node\_create(const char \*entity, const char \*resp);**

**void node\_append(jarvis\_node \*head, jarvis\_node \*node);**

**void node\_free(jarvis\_node \*node);**

**int knowledge\_update\_what(jarvis\_node \*new\_node);**

**int knowledge\_update\_where(jarvis\_node \*new\_node);**

**int knowledge\_update\_who(jarvis\_node \*new\_node);**

**/\* Initialize loading bar for the animation, this is for saving function \*/**

**void loading\_bar();**

**/\* Chatbot Linked lists, our chatbot named Jarvis. \*/**

**jarvis\_node \*headptr\_what;**

**jarvis\_node \*headptr\_where;**

**jarvis\_node \*headptr\_who;**

**#endif**