



ICT1002 Programming Fundamentals

Team Project Specification

Objective

Chatbots – computer programs that can respond to queries typed in natural language – have recently become a popular addition to web sites and other interactive applications.

The aim of this project is to implement a simple chatbot that can respond to simple queries made up of a single question word (called the *intent*) and an object (called the *entity*). The chatbot will be able to answer questions such as *Where is the printer?* or *What is C?* by identifying the intent (*where* or *what*) and entity (*printer* or *C*) then looking up the answer corresponding to this pair in a simple database.

The chatbot will also be able to learn new answers by asking questions of the user. If a user asks a question for which the database does not contain an answer, the user will be given the option to supply an answer that can be used to respond to the same question in future. (This simulates having the chatbot refer the unknown question to a third party who *does* know the answer.)

Detailed Requirements

Your team is required to implement, in C, a chatbot of the kind described in the introduction. The chatbot will converse in the console using a command-line-like interface; **no graphical UI is required.**

On each iteration of the main loop of the program (implemented for you in the *main* function in the skeleton code), the program will display a prompt and wait for the user to type a question or instruction.

Firstly, your chatbot should recognise some stock words and phrases to which it can give rote replies. These phrases will be referred to as *smalltalk* in the remainder of this document. For example, the chatbot may reply “hello” whenever the user types “hello” or “hi”. Your chatbot should have at least five items of smalltalk.

Apart from smalltalk, the chatbot will understand only a very simplified form of English in which all sentences are composed of only two parts, called the *intent* and the *entity*, in that order.

- The *intent* represents what the user wants to do or to know. It may be a verb instructing the chatbot to do something, or a question word like *what*, *where*, etc.
- The *entity* is a noun phrase representing the object to which the instruction or question refers.

The table below describes the intents and entities that must be understood by your chatbot.

Instructions	Entity	Description
RESET	-	Reset the chatbot to its initial state.
LOAD	<i>filename</i>	Load entities and responses from <i>filename</i> .
SAVE	<i>filename</i>	Save the known entities and responses to <i>filename</i> .
EXIT	-	Exit the program.

Questions	Entity	Description
WHERE [IS]	any noun phrase	Give the location of the entity.
WHAT [IS]	any noun phrase	Give a definition of the term.
WHO [IS]	<i>name</i>	Describe the person of this name.

For example, the user might type in *Where is SIT?*, which consists of an intent “WHERE” and an entity “SIT”. The chatbot might answer *SIT has campuses at each of Singapore’s polytechnics, and a dedicated campus at Dover.*

The chatbot will start without any knowledge of any entities, but can learn about them by asking the user. If someone asks it a question for which it does not know the answer, the chatbot will print *I don’t know* followed by the question. The user may then type in answer. If the user types an answer, the chatbot will thereafter use this answer. If the user simply presses ‘enter’ without typing anything, the chatbot will ignore the response. In either case, the chatbot will then go back to waiting for more questions.

A conversation might therefore proceed as follows:

Chatbot: Hello.

Me: What is the ICT Cluster?

Chatbot: I don’t know. What is the ICT Cluster?

Me: The ICT Cluster teaches information and communications technology.

Chatbot: Thank you.

Me: What is the ICT Cluster?

Chatbot: The ICT Cluster teaches information and communications technology.

Some more detailed sample output is given at the end of this specification sheet.

Once the chatbot has learned some entities, they can be saved to disk using the SAVE intent, and recalled using the LOAD intent. The RESET intent will erase all of the chatbot’s knowledge, leaving only the smalltalk in memory (but it will not erase any of the files created by SAVE, so that they can be re-loaded later).

The LOAD intent should append all of the entities and responses in the file to whatever entities and responses already exist in the chatbot’s memory. If an entity in the file is the same as one that is already in memory, the corresponding response from the file should replace the response currently in memory.

Specifications

Skeleton

The skeleton program provided to you gives some guidance on how to structure a program of this kind and some hints on how it can be implemented. It consists of a header file, `chat1002.h`, that contains

all of the type definitions and function prototypes for the program together with three modules as follows:

- `main.c` contains the main loop and some utility functions. It prompts the user to type in a question, gathers the response, and divides it into words. You should not need to modify this file.
- `chatbot.c` implements the **behaviour of the chatbot**. It is required to:
 1. Recognise smalltalk and provide an appropriate response.
 2. Otherwise, identify the intent and entity in each line of input from the user.
 3. If the intent is an instruction, carry it out.
 4. If the intent is a question, search the knowledge base (see below) for an answer.
 5. If the knowledge base does not contain an answer, ask the user for an answer, and add the new answer to the knowledge base.
- `knowledge.c` implements the chatbot's knowledge base, as described in the next section. For each question intent understood by the chatbot, the knowledge base should support
 1. Searching the knowledge base for an answer corresponding to the entity in the question.
 2. Adding a new entity with a corresponding answer to the question.
 3. Erasing the existing entities and answers.

A detailed explanation of what each function should do is given in a comment at the start of the function. Refer also to the comment at the top of `chatbot.c` for how that module works.

You may alter the skeleton code if you wish but do be sure to update any comments in the code so that they reflect what the modified code does.

Knowledge Base

Conceptually, the knowledge base consists of a **single list for every question intent understood** by the chatbot. Each element in the **list consists of an entity together with the answer to the question** for that entity. Each list may be of arbitrary length; in particular it may grow indefinitely as the chatbot learns more answers to more questions. It is up to you to decide how these lists should be implemented, and your **implementation should be described in the report that accompanies your submission**.

You may assume that no entity is longer than `MAX_ENTITY` characters and that no answer is longer than `MAX_RESPONSE` characters, as defined in `chat1002.h`.

Different questions may refer to the same entity with different answers. *Where is SIT?*, for example, might have the answer already described, while *What is SIT?* might explain that *SIT is the Singapore Institute of Technology, an autonomous university in Singapore*. An entity need not be associated with an answer to all question intents; *Who is SIT?*, for example, need not be understood even if *Where is SIT?* and *What is SIT?* are.

Entities may be matched using **simple a case-insensitive string matching algorithm**, such as the one implemented by `compare_token()` in `main.c`. You **do not** need to perform expansion of acronyms, stemming, etc. or any other sophisticated types of matching (these require the use of third party libraries in C).

File Format

A chatbot's knowledge base will be stored on disk using a file format similar to that used by "INI" files. **An INI file is a text file divided into sections, with each section containing a series of key-value pairs written one per line.** For our chatbot, **each section corresponds to an intent, and each key-value pair specifies an entity (the key) and the answer for the intent with regard to that entity (the value).**

Each section begins with a line containing the name of that section (i.e. the intent) in square brackets. **Each line within the section consists of the key, followed by a single equal sign (=), followed by the**

corresponding value. You may assume that no line of the file is longer than `MAX_ENTITY + 1 + MAX_RESPONSE + 1` characters, as defined in `chat1002.h` (the +1's account for the equal sign and the new line character).

When reading a file, other lines should be handled as follows:

- blank lines should be ignored
- if the file contains a section heading that does not correspond to an intent understood by the chatbot, the whole section should be ignored
- any entity/response pairs that occur prior to the first section heading should be ignored
- lines that do not contain either an equal sign or square brackets should be ignored
- any characters that appear on the line after the closing square bracket in a section heading should be ignored

You may download a sample file (“ICT1002_Group Project Assignment.ini”) from xSiTe that describes a chatbot able to answer some questions about SIT. Your program should be able to read this file, and to write new files in the same format.

Hints

It is recommended that you complete this project in stages and test each stage works before moving on to the next stage. Working in stages means that you only need to solve one relatively simple problem at a time, and you can complete each stage as soon as you have mastered the necessary materials from class.

First, make sure that you understand the skeleton code. Pay careful attention to the comments as these give you many instructions on what you need to do.

You might then proceed as follows:

1. Implement `chatbot_is_smalltalk()` and `chatbot_do_smalltalk()` so that your chatbot understands some smalltalk.
2. Implement a preliminary version of `knowledge_get()` that simply returns a few hard-coded responses to questions, and implement `chatbot_is_question()` and `chatbot_do_question()` to use it.
3. Implement `knowledge_put()` so that you can add new entities to the knowledge base and re-implement `knowledge_get()` to retrieve these entities
4. Modify `chatbot_do_question()` so that it asks the user for new knowledge whenever the user asks a question that is not in the knowledge base.
5. Implement `chatbot_do_reset()` so that it erases all of the knowledge created by `knowledge_put()`. The main function resets the chatbot at the beginning of the program, and again whenever the user issues the RESET intent.
6. Implement `chatbot_do_save()` and `chatbot_do_load()`.

This sequence follows the order in which we will cover the necessary concepts and techniques in class.

Output

The main loop expects the `chatbot_do_*` functions to fill a buffer with the response from the chatbot. To do this, you can use the `snprintf()` function defined in `stdio.h`. This does the same thing as `printf()`, but puts the output in a buffer instead of on the screen. The following line copies a string “Result = *some number*” into the buffer called `output`, making sure that no more than `MAX_OUTPUT` characters (the length of the buffer) are written:

```
snprintf(output, MAX_OUTPUT, "Result = %f", result);
```

Since modern programs often make output to graphical displays, web pages, or other things, rather than the console, I think it is good practice to return messages to the part of the program responsible for making the display, rather than to print them directly.

The chatbot functions can also call the `prompt_user()` function defined in `main.c` in order to get a string from the user. This function will prompt the user in the same format as the main loop, read a whole line of input, strip the new line character at the end, and put the response in a buffer.

Error Handling

As you add new features to your program, think carefully about the kinds of things that might go wrong. Some of these are discussed in the specification above. Others, such as file errors or memory allocation failures, might be reported by C library functions.

Make sure that your program detects all of the errors that might occur, and outputs brief but meaningful feedback to users when something goes wrong. Also ensure that there are no buffer overflows in your program. Aside from producing professional programs that do not crash and are pleasant to use, I find that handling errors carefully helps me find identify problems in the code.

Report

In addition to your code, you must also submit a brief technical report that describes how the program works and how you went about constructing it. In particular, it should contain

- a technical description of how smalltalk is implemented;
- a technical description of how the knowledge base is implemented; and
- a breakdown of the contribution made by each team member to the project.

The technical descriptions should explain what data structures (if any) were used and why these were chosen. They should be of sufficient detail that someone else who has also studied ICT1002 would be able to implement a similar program without needing to refer to your code.

The report **does not** need to contain a user guide for the program.

Timeline and Deliverables

By 11:30pm, 2 Dec 2021, Thursday	Final Submission	<p>Submit a zip file to your group's xSiTe Dropbox, including:</p> <ul style="list-style-type: none">• All C source code• Report• Project presentation and demonstration video (maximum 10 minutes) <p>Instructions on the "Project presentation and demonstration video" will be shared at a later time.</p> <p>The zip file should be named in format of: ICT1002_C_GP_Team X.zip. For example: ICT1002_C_GP_Team 10.zip.</p>
By 11:30pm, 3 Dec 2021, Friday	Peer Evaluation	<p>Complete your peer evaluation on TEAMMATES. Otherwise, the contribution credits will be evenly distributed among the team members.</p>

Late Submission

A penalty of 20% per day for each deliverable will be imposed for late submission unless extension has been granted **prior** to the submission date. Request for extension will be granted on a case-by-case basis. Any work submitted more than 4 days after the submission date will not be accepted and no mark will be awarded.

Plagiarism

SIT's policy on copying does not allow you to copy software as well as your assessment solutions from another person. It is not acceptable to copy other person's work. It is the students' responsibility to guarantee that their assessment solutions are their own work. Meanwhile, you must also ensure that others don't obtain access to your work. Where such plagiarism is detected, both of the assessments involved will receive **ZERO** mark.

Assessment criteria

Your assignment will be assessed according to the criteria listed in the mark scheme in Table 1.

Table 1 Assessment Criteria

Each requirement/function will be evaluated based on the following criteria:	Weight
Code <ul style="list-style-type: none">• Code completeness and correctness: implement all required features correctly and efficiently.• Code reliability: good data validation, no bugs or errors• Code clarity: easy to read and understand with comments and good modularity	40% 20% 20%
Report <ul style="list-style-type: none">• Your report should include all required information• Your report should be well-organised and easy to read, without errors	15%
Project presentation and demonstration video <ul style="list-style-type: none">• Your video should clearly present all the functions and features implemented following the requirements	5%

The grade for each individual member will be weighted based on the contribution and peer evaluation.

Appendix A: Sample Output

The following shows a sample run of the completed program. In this dialogue, the chatbot is called “Zeus” and the user is called “Prometheus”. Black text indicates text output by the interpreter, blue text represents text input by the user, and the orange text on the right is commentary that does not appear on the screen.

Zeus: Hello, I'm Zeus.

Prometheus: Hello

Zeus: Hello.

Prometheus: It's hot today.

Zeus: Indeed it is.

Prometheus: What is the temperature?

Zeus: I don't know. What is the temperature?

Prometheus: It is about 32C.

Zeus: Thank you.

Prometheus: What is the temperature?

Zeus: It is about 32C.

Prometheus: That's hot

Zeus: I don't understand "That's".

Prometheus: reset

Zeus: Chatbot reset.

Prometheus: What is the temperature?

Zeus: I don't know. What is the temperature?

Prometheus:

Zeus: :-(

Prometheus: reset

Zeus: Chatbot reset.

Prometheus: load sample.ini

Zeus: Read 12 responses from sample.ini.

Prometheus: What is SIT?

Zeus: SIT is an autonomous university in Singapore.

Prometheus: Where is SIT?

Zeus: SIT has a main campus at Dover plus a building at each of Singapore's polytechnics.

Zeus understands smalltalk like “hello” and agrees with any statement beginning “it’s”.

Prometheus asks about the temperature but Zeus does not know until Prometheus tells him.

Zeus doesn’t recognise “that’s”.

Reset the chatbot.
Zeus forgets the temperature.
This time Prometheus doesn’t answer.

Load the sample file.
Now Zeus can answer questions about SIT.

Prometheus: Who is Tan Thiam Soon?

Zeus: I don't know. Who is Tan Thiam Soon?

Prometheus: Tan Thiam Soon is the President of SIT.

Zeus: Thank you.

Prometheus: save as sample-plus.ini

Zeus: My knowledge has been saved to sample-plus.ini.

Prometheus: quit

Zeus: Goodbye!

Add some more
knowledge about SIT.

Save the extended
knowledge to a new
file.

Appendix B: Sample File

The sample knowledge base available from xSiTe (“ICT1002_Group Project Assignment_Sample.ini”) contains responses to some questions about SIT and the ICT Programme, some of which are shown in the sample dialogue above. You can examine its contents in any text editor.

Your program should be able to read this file and create new ones like it.