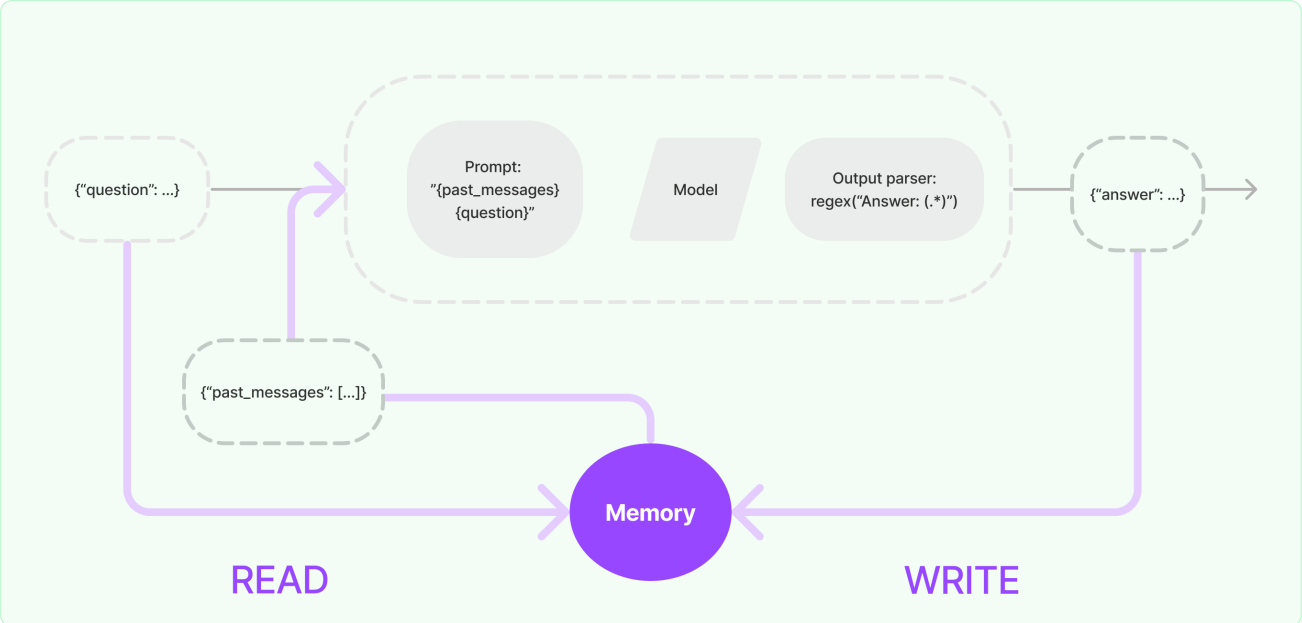
Memory

Most LLM applications have a conversational interface. An essential component of a conversation is being able to refer to information introduced earlier in the conversation. At bare minimum, a conversational system should be able to access some window of past messages directly. A more complex system will need to have a world model that it is constantly updating, which allows it to do things like maintain information about entities and their relationships.

We call this ability to store information about past interactions "memory". LangChain provides a lot of utilities for adding memory to a system. These utilities can be used by themselves or incorporated seamlessly into a chain.

A memory system needs to support two basic actions: reading and writing. Recall that every chain defines some core execution logic that expects certain inputs. Some of these inputs come directly from the user, but some of these inputs can come from memory. A chain will interact with its memory system twice in a given run.

1. AFTER receiving the initial user inputs but BEFORE executing the core logic, a chain will READ from its memory system and augment the user inputs.
2. AFTER executing the core logic but BEFORE returning the answer, a chain will WRITE the inputs and outputs of the current run to memory, so that they can be referred to in future runs.



Building memory into a system

The two core design decisions in any memory system are:

- How state is stored
- How state is queried

Storing: List of chat messages

Underlying any memory is a history of all chat interactions. Even if these are not all used directly, they need to be stored in some form. One of the key parts of the LangChain memory module is a series of integrations for storing these chat messages, from in-memory lists to persistent databases.

- Chat message storage: How to work with Chat Messages, and the various integrations offered

Querying: Data structures and algorithms on top of chat messages

Keeping a list of chat messages is fairly straight-forward. What is less straight-forward are the data structures and algorithms built on top of chat messages that serve a view of those messages that is most useful.

A very simply memory system might just return the most recent messages each run. A slightly more complex memory system might return a succinct summary of the past K messages. An even more sophisticated system might extract entities from stored messages and only return information about entities referenced in the current run.

Each application can have different requirements for how memory is queried. The memory module should make it easy to both get started with simple memory systems and write your own custom systems if needed.

- Memory types: The various data structures and algorithms that make up the memory types LangChain supports

Get started

Let's take a look at what Memory actually looks like in LangChain. Here we'll cover the basics of interacting with an arbitrary memory class.

Let's take a look at how to use ConversationBufferMemory in chains. ConversationBufferMemory is an extremely simple form of memory that just keeps a list of chat messages in a buffer and passes those into the prompt template.

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

When using memory in a chain, there are a few key concepts to understand. Note that here we cover general concepts that are useful for most types of memory. Each individual memory type may very well have its own parameters and concepts that are necessary to understand.

What variables get returned from memory

Before going into the chain, various variables are read from memory. This have specific names which need to align with the variables the chain expects. You can see what these variables are by calling `memory.load_memory_variables({})`. Note that the empty dictionary that we pass in is just a placeholder for real variables. If the memory type you are using is dependent upon the input variables, you may need to pass some in.

```
memory.load_memory_variables({})
```

```
    {'history': "Human: hi!\nAI: whats up?"}
```

In this case, you can see that `load_memory_variables` returns a single key, `history`. This means that your chain (and likely your prompt) should expect and input named `history`. You can usually control this variable through parameters on the memory class. For example, if you want the memory variables to be returned in the key `chat_history` you can do:

```
memory = ConversationBufferMemory(memory_key="chat_history")
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
    {'chat_history': "Human: hi!\nAI: whats up?"}
```

The parameter name to control these keys may vary per memory type, but it's important to understand that (1) this is controllable, (2) how to control it.

Whether memory is a string or a list of messages

One of the most common types of memory involves returning a list of chat messages. These can either be returned as a single string, all concatenated together (useful when they will be passed in LLMs) or a list of ChatMessages (useful when passed into ChatModels).

By default, they are returned as a single string. In order to return as a list of messages, you can set `return_messages=True`

```
memory = ConversationBufferMemory(return_messages=True)
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
    {'history': [HumanMessage(content='hi!', additional_kwargs={},
example=False),
  AIMessage(content='whats up?', additional_kwargs={}, example=False)]}
```

What keys are saved to memory

Often times chains take in or return multiple input/output keys. In these cases, how can we know which keys we want to save to the chat message history? This is generally controllable by `input_key` and `output_key` parameters on the memory types. These default to None - and if there is only one input/output key it is known to just use that. However, if there are multiple input/output keys then you MUST specify the name of which one to use

End to end example

Finally, let's take a look at using this in a chain. We'll use an LLMChain, and show working with both an LLM and a ChatModel.

Using an LLM

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory


llm = OpenAI(temperature=0)
```

```
# Notice that "chat_history" is present in the prompt template
template = """You are a nice chatbot having a conversation with a
human.

Previous conversation:
{chat_history}

New human question: {question}
Response:"""
prompt = PromptTemplate.from_template(template)
# Notice that we need to align the `memory_key`
memory = ConversationBufferMemory(memory_key="chat_history")
conversation = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory
)
```

```
# Notice that we just pass in the `question` variables - `chat_history`
gets populated by memory
conversation({"question": "hi"})
```

Using a ChatModel

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory


llm = ChatOpenAI()
prompt = ChatPromptTemplate(
    messages=[
        SystemMessagePromptTemplate.from_template(
            "You are a nice chatbot having a conversation with a
human."
        ),
        # The `variable_name` here is what must align with memory
        MessagesPlaceholder(variable_name="chat_history"),
```

```
        HumanMessagePromptTemplate.from_template("{question}")
    ]
)
# Notice that we `return_messages=True` to fit into the
MessagesPlaceholder
# Notice that `"chat_history"` aligns with the MessagesPlaceholder
name.
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)
conversation = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory
)
```

```
# Notice that we just pass in the `question` variables - `chat_history`
gets populated by memory
conversation({"question": "hi"})
```

Next steps

And that's it for getting started! Please see the other sections for walkthroughs of more advanced topics, like custom memory, multiple memories, and more.