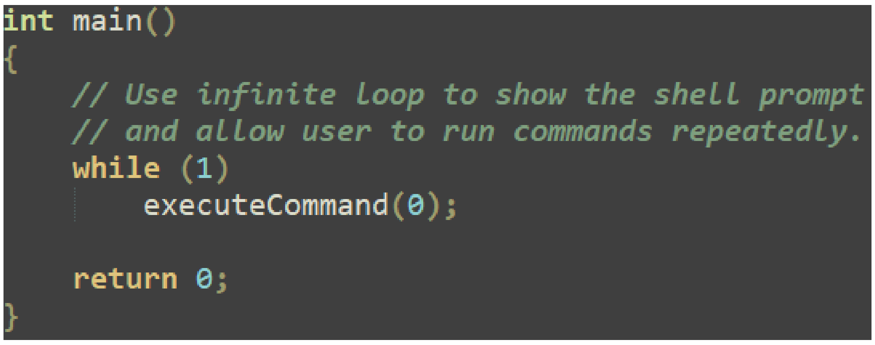
Custom Unix Shell

This document explains the internal design of **Custom Unix Shell** created using code written in C language. It is a miniature replica of the shells that are typically provided in Unix and Linux operating systems.

High-level Design

At a very high level, this **Custom Unix Shell** simply take commands from user and execute them continuously in an infinite loop. The command is read and parsed to determine the type of command being executed. After parsing is done, based on the type of command, appropriate functions are called in code to run that specific type of command. This delegation to other functions on the basis of type of command is managed by the executeCommand() function in the code.



```c
int main()
{
    // Use infinite loop to show the shell prompt
    // and allow user to run commands repeatedly.
    while (1)
        executeCommand(0);

    return 0;
}
```

The type of commands that are currently supported by this **Custom Unix Shell** are listed below:

(1) **Simple Command**: There is no pipe or redirection operator involved in this type of command.

(2) **Output Redirection Command**: The > output redirection operator is used in this type of command.

(3) **Output Append Command**: The >> output append operator is used in this type of command.

(4) **Input Redirection Command**: The < input redirection operator is used in this type of command.

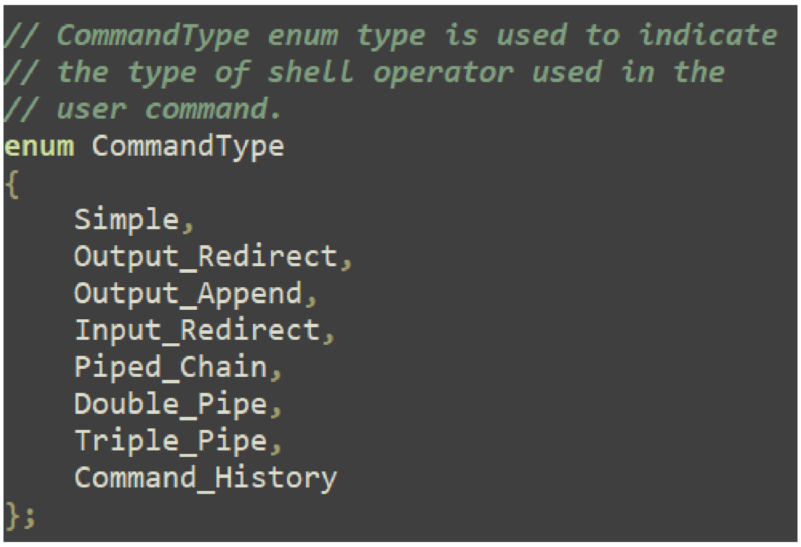
(5) **Piped Chain Command**: Multiple pipe | operators have been used and output of one pipe | is used as input to another pipe | in this type of command.

(6) **Double Pipe Command**: The double pipe || operator is used in this type of command. The output of one pipe is used as input to two other comma separated commands.

(7) **Triple Pipe Command**: The triple pipe ||| operator is used in this type of command. The output of one pipe is used as input to three other comma separated commands.

(8) **History Command**: The user is trying to run “cmdhist” command in which we are trying to replicate the functionality of “history” command provided by some Unix shells. This is the **additional feature** I have supported in Custom Unix Shell.

There is an enum CommandType in the program which is used to indicate the type of command being executed by the Custom Unix Shell.



```
// CommandType enum type is used to indicate
// the type of shell operator used in the
// user command.
enum CommandType
{
    Simple,
    Output_Redirect,
    Output_Append,
    Input_Redirect,
    Piped_Chain,
    Double_Pipe,
    Triple_Pipe,
    Command_History
};
```

<u>Low-level Design</u>

The design and implementation of the specific types of commands is explained below:

(1) **<u>Simple Command</u>**: This type of simple command execution is handled by executeNormal() function in code. The simple command is executed by created a child process using fork() and then running the command by using execvp() system call inside the child process code. The parent process simply waits for the child to execute and then prints the PID and status of the child process.

(2) **<u>Output Redirection Command</u>**: This type of command execution is handled by executeOutputRedirect() function. The child process is created and it runs the command to the left of > operator. Instead of writing to terminal, the child process writes output to the write end of the pipe. In the parent process, the data is read from the read end of that pipe and written to the file.

(3) **<u>Output Append Command</u>**: This type of command execution is handled by the same executeOutputRedirect() function as in (2) above. The parameter iAppend is passed as 1 to this function and therefore in the last step, instead of writing data to the file, the data is appended to the file.

(4) **<u>Input Redirection Command</u>**: This type of command execution is handled by executeInputRedirect() function. The child process is created and it closes standard input & reads from the file instead. After that is done, the command is normally executed using execvp() system call.

(5) **Piped Chain Command**: This type of command execution is handled by executePipedChain() function. As first step, all the different commands in the piped chain are split into separate commands by calling splitChainedCommands() function. Further each of these separate commands is executed by calling executeSingleCommand() function repeatedly inside a for loop. A pipe is used to receive the output of one command at one end and the other other end of pipe is used as input to the next command. The executeSingleCommand() simply creates a child process using fork() and this child process executes one given command using execvp() system call.

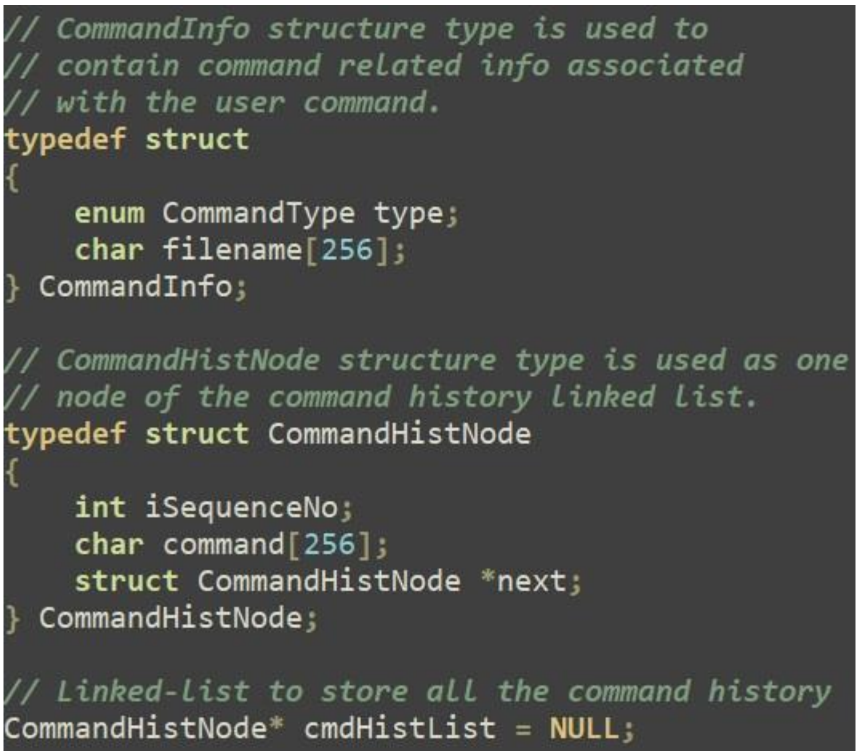
(6) **Double Pipe Command**: This type of command execution is handled by executeDoublePipe() function. The given command is split into 3 commands. After that the three commands are passed to the executeDoublePipeHelper() function. Initially a child process is created using fork() and the first command is executed in the child process using execvp(). The output of the first command is written to the write end of a pipe and not to standard output. In the parent process, the read end of this pipe is read and data is copied to a char buffer.

Further this char buffer is copied to write end of another pipe. A child process is then created which reads from this pipe and runs the second command using execvp(). Similar steps are then repeated to run the third command.

(7) **Triple Pipe Command**: This type of command execution is handled by executeTriplePipe() function. The given command is split into 4 commands. After that the four commands are passed to the executeTriplePipeHelper() function. Initially a child process is created using fork() and the first command is executed in the child process using execvp(). The output of the first command is written to the write end of a pipe and not to standard output. In the parent process, the read end of this pipe is read and data is copied to a char buffer.

Further this char buffer is copied to write end of another pipe. A child process is then created which reads from this pipe and runs the second command using execvp(). Similar steps are then repeated to run the third and fourth commands.

(8) **History Command**: As mentioned earlier, this is the **additional feature** supported in Custom Unix Shell. The user is trying to run “cmdhist” command in which we are trying to replicate the functionality of “history” command provided by some Unix shells. While running each of the commands in executeCommand() function, we keep a record of the command being executed along with the sequence number of command by calling insertIntoCommandHistory() function. This function will store the command in a linked list whose data structures are shown below:



```c
// CommandInfo structure type is used to
// contain command related info associated
// with the user command.
typedef struct
{
    enum CommandType type;
    char filename[256];
} CommandInfo;

// CommandHistNode structure type is used as one
// node of the command history linked list.
typedef struct CommandHistNode
{
    int iSequenceNo;
    char command[256];
    struct CommandHistNode *next;
} CommandHistNode;

// Linked-list to store all the command history
CommandHistNode* cmdHistList = NULL;
```

Further when user runs “cmdhist” command on CustomUnixShell, this is handled by executeCommandHistory() function. There are two cases in execution.

If the user has not passed any argument with the command, the executeCommandHistory() function will simply show the complete history of all the previously executed commands by traversing the “cmdHistList” linked list.

If the user has passed a sequence number as an argument to the cmdhist command, then the executeCommandHistory() function will simply run that command from the history by calling executeCommand() function and passing that sequence number as argument. The executeCommand() function further takes help of getCommandFromHistory() helper function to retrieve the command with that sequence number from the “cmdHistList” linked list.