

# Assignment #1.1

OPERATING SYSTEMS

Deadline:	A1.1: As specified on the Moodle
Topics covered:	Process creation and management
Deliverables:	To submit all relevant files that will implement the uShell program. Please upload the files to the moodle. Your program must be compile-able in the VPL environment using the g++ compiler (-std=c++17). A main function has been provided by VPL.
Objectives:	To learn and understand the basic of the internals of a shell program.

## Programming Statement: A new shell program called uShell

In this assignment, you are tasked with the assignment to create a basic shell program called **uShell**. A shell program behaves typically in a similar fashion to a command prompt. Usually, it has the following capabilities:

- Ignore comments statement
- Perform internal commands
- Run programs and launch processes
- Setting of environment variables
- Perform input/output redirection
- Performing piping as a means of interprocess communications between processes

## 1 An overview of uShell

Your program will be referred to as **uShell**. It is to be run either in interactive mode or as a shell script. Thus, the input to **uShell** can either be from the terminal or a file.

### 1.1 Usage

**uShell** is to be run as follows:

```
uShell [-v]
```

The bracketed arguments are optional. Their meaning are as the following:

- If the **-v** argument is used, the Shell is running in the verbose mode. In the verbose mode, every command line entered is repeated.

Some examples:

- **uShell** – No arguments at all. Input from terminal.

- `uShell -v` – Input from terminal, but every line entered is printed before execution.
- `uShell -v < test1.txt > result1.txt` – Input is taken from `test1.txt` and output is directed to `result1.txt`. Support input and output redirection.

## 1.2 Input

The program reads a command line one at a time from the standard input till an `exit` command or EOF from the file input is encountered. Every command line is terminated upon a newline character. Every command line is either empty or a sequence of *word*, where every *word* is a character sequence of printable characters but not whitespace characters<sup>1</sup>. The length of the command line is usually limited by the parameter `ARG_MAX` defined by Linux<sup>2</sup>.

Hence, it is possible for a command line to be empty – it is simply a newline character or it is made up of only whitespaces. Furthermore, there is a special category of command line that can be ignored by the program – a comment. At any point in a command line, if the `#` sign is used, the rest of the line is treated as a comment and it can be ignored by the shell program. `uShell` does not consider quoting or backslash escaping.

## 1.3 Command line examples

The following shows some example of possible command lines. As you can see, there are both internal commands and external commands. The difference between the two will be made clear in later sections.

Command	Explanation
<code># comment haha</code>	<i>This is a comment</i>
<code>/usr/bin/gcc</code>	<i>An external command - executing a program with full path given</i>
<code>ls -l</code>	<i>An external command - executing a program to be found among the directory list in environment variable PATH</i>
<code>echo hello</code>	<i>An internal command - printing the word hello to standard output</i>
<code>echo hello # silly comment</code>	<i>An internal command - printing the word hello to standard output. silly comment is ignored because it occurs after the # symbol</i>

When standard input is used as input, there should be a prompt printed before accepting input. For example,

```
uShell> echo haha # whatever
haha
uShell> ..
```

As explained in the next section, the default prompt is `uShell` but it can be changed by the `changeprompt` internal command.

---

<sup>1</sup>Whitespace is the set of blank characters, commonly defined as space, tab, newline and possibly carriage return

<sup>2</sup>You may use Linux command `getconf ARG_MAX` to get the value.

## 1.4 Internal Commands

An internal command is directly executed by the shell itself without executing another program. The first word in the command determines which internal is to be executed. The internal commands<sup>3</sup> are:

- **echo**: print out the rest of the arguments.
- **exit <num>**: terminate the shell program with an exit value given by **num**. If **num** is not given, the default exit value is 0.
- **changeprompt <str>**: change the prompt to the string given by **str**.
- **setvar**: described in Section 1.5 on shell variables.

## 1.5 Shell Variables

Any word in a command line that begins with the character \$ and enclosed with curly braces refers to a shell variable. The default value of a variable is an empty string i.e., if a shell variable is used without any initialization, it is an empty string. The value can be changed via command line. The command to define and change the value of a variable is **setvar**. The usual syntax of the **setvar** command is as follows:

```
setvar <varname> <value>
```

The names of the shell variable are case-sensitive. To call out a shell variable, the '\$' character is used to indicate that a variable is used. The following examples demonstrate the usage of the **setvar** command.

```
uShell> setvar HAHA hoohoo # assign the value hoohoo to HAHA
uShell> echo ${HAHA} # calling out the value of HAHA
hoohoo
uShell> # hoohoo is printed on the screen.
uShell> setvar haha # variable haha is defined and given a default value.
uShell> echo ${Haha}123 # Attempting to call out the value of an undefined variable.
Error: Haha is not a defined variable.
uShell> echo ${HAHA}123 # disambiguate the beginning and end of the var name
hoohoo123
uShell> echo ${HAHA }123 # wrong use of curly braces. Would be read as 2 separate words.
${HAHA }123
uShell> echo $$${HAHA} # $ sign can be used together with variables
$hoohoo
uShell> echo ${${HAHA}} # nested use of curly braces are not supported
${hoohoo}
uShell> # So the replacement only happens once.
```

Please ensure that your program prints out the same behaviour as given above.

---

<sup>3</sup>Please note that internal commands **exit** and **changeprompt** are not required in this assignment.

## 1.6 changeprompt command

The examples of `changeprompt` are as follows:

```
uShell>changeprompt asd
asd>changeprompt gree sad
gree sad>changeprompt as_Asd    fd
as_Asd fd>changeprompt asfc # sad
asfc>changeprompt #asd
#asd>changeprompt asd asd#ds
asd asd#ds>exit
```

## 2 uShell.h

You will finish the following definition of some member functions in `uShell.cpp`, which is the only file required to submit.

```
#include <string>      // std :: string
#include <vector>       // std :: vector
#include <map>          // std :: map
/*! list of strings */
using TokenList = std::vector<std :: string >;
```

  

```
/************************************************************************
/*!
\brief
uShell class. Acts as a command prompt that takes in input and performs
commands based on the input
*/
/************************************************************************
class uShell
{
protected:

/*! Map of variables to string values */
std ::map<std :: string , std :: string > m_vars;

/*! String prompt that is displayed at the beginning */
std :: string m_prompt;

/*! decide whether to echo user input */
bool m_verbose;

/*! determine whether to exit */
bool m_exit;

/*! exit code determined during runtime */
int m_exitCode;
```

```

/***********************/
/*!
\brief
Get one line of input from std::cin. Removes the trailing \r if any
\param input
The input string to take the user input by reference
\return
return false if encountering the EOF
*/
/***********************/
bool getInput(std::string & input) const;

/***********************/
/*!
\brief
Print the user input without printing the starting and the trailing spaces.
\param input
The input string
*/
/***********************/
void printVerbose(std::string const & input);

/***********************/
/*!
\brief
Tokenize the input string into a list of strings which are separated by
spaces.
\param input
The input string given by the user
\param tokenList
Token list to fill up, passed by reference to add strings to it
*/
/***********************/
void tokenize(std::string const & input, TokenList & tokenList) const;

/***********************/
/*!
\brief
Search and replace all variables in the token list with the value from the map
\param tokenList
Token list that stores the variables to be searched
\return
Boolean value to decide whether there is a invalid variable found in the string.
Returning true means all valid. Otherwise,
Returning false means there is a invalid variable (abort the search).
*/
/***********************/
bool replaceVars(TokenList & tokenList) const;

```

```

/*****
/*!
\brief
Check whether each character is a number or digit from the given start and
end of a character array.
\param start
Pointer to the start of string to search from
\param end
Pointer to the end of string , where the search stops
\return
boolean value to represent if the string is valid
Returning true means each character in the given string is valid.
Otherwise , it returns false .
*/
/*****
bool isValidVarname(char const * start , char const * end) const;

/*****
/*!
\brief
Merge all the tokens from a given token list , separated by a space , into one
single string . Move semantics help in this case.
\param tokenList
The token list from which to take the tokens
\param startPos
The starting position to start merging from
\return
The merged string
*/
/*****
std::string mergeTokens(TokenList const & tokenList ,
                        unsigned startPos) const;

/*****
/*!
\brief
Echo the user input
\param tokenList
The list of data to read in
*/
/*****
void echo(TokenList const & tokenList);

/*****
/*!
\brief
Set the value of a variable

```

```

\param tokenList
The list of tokens from which to get the data value
*/
/***********************************************/
void setVar(TokenList const & tokenList);

public:

/***********************************************/
/*!
\brief
Creates the class object of uShell
\param bFlag
boolean value to decide whether to echo input
*/
/***********************************************/
uShell(bool bFlag);

/***********************************************/
/*!
\brief
Public function for external call. Execute in loops and waits for input.
\return
Exit code, of the exit command
*/
/***********************************************/
int run();

};


```

### 3 Functions to complete in this assignment

The following functions are to complete in this assignment.

#### 3.1 bool uShell::getInput(std::string & input) const

This function obtains a line from input stream and checks whether the last character, is a '\r', and removes it. This caters to file formats that has '\r' with '\n'.

#### 3.2 void uShell::printVerbose(std::string const & input)

This function finds the first last non-space positions and prints everything in between. Remember to print the line end.

### **3.3 void uShell::tokenize(std::string const & input, TokenList & tokenList) const**

This function gets the string stream from the input and obtains each token<sup>4</sup> item by item by pushing the item to token list.

### **3.4 bool uShell::replaceVars(TokenList & tokenList) const**

This function loops through all tokens in the token list.

In each iteration, it checks each token for comment. If there is comment sign, it removes all the following tokens (including this) and terminates the loop. Otherwise, it detects if each token has a variable within. Furthermore it checks if this token has '{' following the '\$' sign. If not, it tries to find the next '\$' sign. If there is a '{', it then searches for the first instance of '}' after '\${}'. If there is no '}' in the token, it just skips. However, if the '}' is in the token, it continues to do changing for the variable in the token string.

It checks if the variable name is valid by determining whether every character in between is a proper character (using the function **isValidVarname()**). If it is invalid, it aborts the attempt to change. Otherwise, it gets the string between '\${' and '}' and check whether the string is in the list of the variables. If not, it aborts the attempt to change and returns. If the search is successful, it replaces the variable with the correct string and updates the original token string. Please note that the assignment requires only one exchange per token.

This function then repeats the above steps, after finding the next dollar sign.

### **3.5 bool uShell::isValidVarname(const char \* start, char const \* end) const**

This function loops from **start** to **end**. In each iteration, it checks each digit if it's an alphabet or number.

### **3.6 std::string mergeTokens(TokenList const & tokenList, unsigned startPos) const**

This function merges the tokens into one string, adding a space in between each pair of token.

### **3.7 uShell::uShell(bool bFlag)**

This function initializes **m\_prompt**, **m\_verbose**, **m\_exit** and **m\_exitCode**. It also sets PATH into the map in **m\_vars**.

## **4 Functions to complete in the next assignment**

The following functions are to complete in the next assignment<sup>5</sup>.

---

<sup>4</sup>“Tokens” are smaller chunks of the string that are separated by a specified character, called the delimiting character.

<sup>5</sup>The functions are provided in **uShell\_helper.obj**.

#### **4.1 void uShell::echo(TokenList const & tokenList)**

This function prints all the tokens except the first, adding a space in between each pair of token.

#### **4.2 void uShell::setVar(TokenList const & tokenList)**

In this function, if there is no input, then setvar command just returns. Otherwise it has to check if the variable name is proper by ensuring that the first letter is an alphabet and also continue to loop through the rest of the characters of this variable to check whether the variable name is valid by using `isValidVarname()`. If there are more variables at the back, it merges them all together (using `mergeTokens()`) and sets it to the mapped variable. Otherwise, it just initializes to an empty string.

#### **4.3 int uShell::run()**

This function has a loop and firstly check whether an exit state occurs (e.g. due to that exit command is called ). If so, it stops. Otherwise it prints out the prompt, with the right arrow. Then it gets user input. If there are no more lines from the input (or EOF is read if it gets re-directed input from the file), it exits from the program. Otherwise, it clears the input buffer for next input. Obviously, it needs to skip if there is no input (e.g. empty line). It starts to tokenize the input otherwise. After this, it prints the input if verbose mode is set. It replaces all variables if possible. When the function calls for replacement `replaceVars()` returns false (thus an error has occurred), or when the token list size is zero<sup>6</sup>, it continues to process next line of input.

Next, it finds whether the token is an internal command. If so, it activates the internal command. Otherwise, it does the external command (that will be done in the following assignments).

Outside the loop, it returns exit code `m_exitCode`.

## **5 Set up your WSL environment**

The header file and the example of test cases are on the Moodle. The reference executable `uShell_ref` on the Moodle is compiled with g++ 11.3.0 on WSL/ubuntu 22.04. The OBJ files, `shellmain.obj` that implements `main()` and `uShell_helper.obj` that implements the functions in Section 4, can be used to link with your own `uShell.obj` when you compile the source code on your WSL. You are required to submit `uShell.cpp` only.

Please note that the `main()` function is not given but you are encouraged to prepare your own main function. The pseudo code for `main()`: check whether verbose is set. If it is set, turn on verbose mode. Otherwise, turn off verbose mode. Create the shell object with the given verbose mode setting. Call shell object `run()` function and return the value. Please note that internal commands `exit` and `changeprompt` are not required in this assignment.

To test whether your `uShell` works properly, you can follow the given steps to evaluate (some basic test cases): the reference executable `uShell_ref`, which is static linked. Open WSL on Windows and create a file `input.txt` with the following<sup>7</sup>:

---

<sup>6</sup>Please note that if the replacement clears comments, we have to check whether the token list is empty.

<sup>7</sup>You can remove some lines of the commands to construct different test cases.

```

echo echo at by change
setvar TTH2 hasd
echo ${TTH2}
setvar PTH2
echo ${PTH2}123
echo helloworld #So the replacement only happen once

```

As specified in this assignment, we can type in

```
./uShell_ref -v < input.txt > output_ref.txt
```

and type in

```
./uShell -v < input.txt > output.txt
```

At the end, you can type in the following to show the difference:

```
diff -y --strip-trailing-cr --suppress-common-lines output_ref.txt output.txt
```

If there is nothing different shown, it means your output matches with reference output. Otherwise, you should take a look at the difference.

If you fail to upgrade g++ in WSL to newer version 11, you can try the following steps:  
First, make sure your WSL-ubuntu 20.04 installation is up-to-date:

```
sudo apt update && sudo apt full-upgrade
```

Close down Ubuntu on WSL, and from PowerShell:

```
wsl -l -v
# Confirm the distribution name and adjust below if needed
wsl --terminate Ubuntu
```

To restart Ubuntu, you can type in the following:

```
sudo do-release-upgrade
```

## 6 Rubrics

The rubrics are as the following:

- Process comments. Your `uShell` should be able to ignore anything that comes after the `#` character. But `#` should not be part of a word. For example, `echo dff#sea` prints `dff#sea`.
- Support shell variables. Your `uShell` should be able to support the setting of the Shell variables as described in section 1.5 above. In particular, should the user attempt to
  - Call out an existing variable properly e.g.,  `${HAHA}123`, `abc${Haha}def`, the part of the command line must be replaced by the value of the existing variable.
  - Call out a non-existing variable properly e.g.,  `${HoHo}123`, `abc${Hoho}def`, but these variables have not been previously set with a `setvar` command. `uShell` should print out a suitable error message.

- Use whitespace within the curly braces e.g., \${HAHA }, \${HA HA}, your `uShell` must recognize them as two separate words.
- Attempts of nested use of calling out Shell variables are not supported. Hence only 1 replacement should happen for each of these occurrences.
- `echo` command. This `echo` command should be able to re-print the rest of the command line properly. Multiple whitespaces may be ignored and treated as if it is a single whitespace.
- Comments. Comments ought not to be verbose, but explain the big picture and use good variable names to indicate. Readability of your code is key here.
- Structure of your code. Avoid hard-coding.
- Pass the VPL test cases.

The list above is non-exhaustive. The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.