Shell Scripting Basics

What is a shell?

Think of a shell as a translator between you and your computer's operating system. It's like a command-line interface that lets you interact with your computer by typing commands instead of clicking buttons. When you type a command, the shell interprets it and tells the computer what to do. It's the gateway to accessing files, running programs, and managing your system, all through text-based commands.

The default shell for many Linux distros is the GNU Bourne-Again Shell (bash).

Intro to bash shell

When a shell is used interactively, it displays a `$` when it is waiting for a command from the user. This is called the shell prompt.

```
[username@host ~]$
```

If shell is running as root, the prompt is changed to `#`. The superuser shell prompt looks like this:

```
[root@host ~]#
```

What is a bash script?

A bash script is a series of commands written in a file. These are read and executed by the bash program. The program executes line by line.

For example, you can navigate to a certain path, create a folder and spawn a process inside it using the command line.

You can do the same sequence of steps by saving the commands in a bash script and running it. You can run the script any number of times.

Advantages of bash scripting

Bash scripting is a powerful and versatile tool for automating system administration tasks, managing system resources, and performing other routine tasks in Unix/Linux systems. Some advantages of shell scripting are:

- Automation
- Portability
- Flexibility
- Accessibility
- Integration
- Debugging

How to identify a bash script?

File extension of .sh.

By naming conventions, bash scripts end with a .sh. However, bash scripts can run perfectly fine without the sh extension.

Scripts start with a bash bang.

Scripts are also identified with a shebang. Shebang is a combination of bash # and bang ! followed the the bash shell path. This is the first line of the script. Shebang tells the shell to execute it via bash shell. Shebang is simply an absolute path to the bash interpreter.

Below is an example of the shebang statement.

```
#! /bin/bash
```

The path of the bash program can vary. We will see later how to identify it.

Quick recap of commands

`date`: Displays the current date

`pwd`: Displays the present working directory.

`ls`: Lists the contents of the current directory.

`echo`: Prints a string of text, or value of a variable to the terminal

GREP command

`Grep` is a useful command to search for matching patterns in a file. `grep` is short for "global regular expression print".

If you are a system admin who needs to scrape through log files or a developer trying to find certain occurrences in the code file, then `grep` is a powerful command to use.

Syntax : *grep [OPTION...] PATTERNS [FILE...]*

In the above syntax, grep searches for PATTERNS in each FILE. `Grep` finds each line that matched the provided PATTERN. It is a good practice to close the PATTERN in quotes when `grep` is used in a shell command.

GREP continued..

man grep

Options	Description
-c	This prints only a count of the lines that match a pattern
-h	Display the matched lines, but do not display the filenames.
–i	Ignores, case for matching
-l	Displays list of a filenames only.
-n	Display the matched lines and their line numbers.
-v	This prints out all the lines that do not matches the pattern

-e exp	Specifies expression with this option. Can use multiple times.
-f file	Takes patterns from file, one per line.
-E	Treats pattern as an extended regular expression (ERE)
-w	Match whole word
-o	Print only the matched parts of a matching line, with each such part on a separate output line.

Pipe command

In Linux, a pipe command, represented by the symbol |, is used to connect the output of one command to the input of another. This allows you to chain together multiple commands and create powerful combinations to perform complex tasks.

syntax:

command1 | command2

In this example, the output of command1 is passed as input to command2. This allows you to perform operations on the output of one command without needing to save it to a file or use temporary variables.

Pipe command continued..

1. List all files and directories and give them as input to `grep` command using piping in Linux

```
ls | grep file.txt
```

In this first we are using `ls` to list all file and directories in the current directory, then passing its output to `grep` command and searching for file name `file.txt`. The output of the ls command is sent to the input of the grep command, and the result is a list of files that match the search term.

Wc command

wc stands for **word count**. As the name implies, it is mainly used for counting purpose.

- It is used to find out **number of lines**, **word count**, **byte and characters count** in the files specified in the file arguments.
- By default it displays **four-columnar output.**
- First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.

Syntax:

```
wc [OPTION]... [FILE]...
```

Wc command options

1. -l: This option prints the **number of lines** present in a file. With this option wc command displays two-columnar output, 1st column shows number of lines present in a file and 2nd itself represent the file name.

2. -w: This option prints the **number of words** present in a file. With this option wc command displays two-columnar output, 1st column shows number of words present in a file and 2nd is the file name.

3. -c: This option displays **count of bytes** present in a file. With this option it display two-columnar output, 1st column shows number of bytes present in a file and 2nd is the file name.

4. -m: Using **-m** option ‘wc’ command displays **count of characters** from a file.

And many more…

How to create a bash script?

Create a file named hello_world.sh

```
touch hello_world.sh
```

Find the path to your bash shell.

```
which bash
```

Edit the file hello_world.sh using a text editor of your choice and add the above lines in it.

```
#! /usr/bin/bash

echo "Hello World"
```

Run the script

Provide execution rights

Modify the file permissions and allow execution of the script by using the command below:

```
chmod u+x hello_world.sh
```

chmod modifies the existing rights of a file for a particular user. We are adding +x to user u

Run the script

```
./hello_world.sh
```

or

```
bash hello_world.sh
```

Define a variable

We can define a variable by using the syntax variable_name=value. To get the value of the variable, add $ before the variable.

```
#!/bin/bash

# A simple variable example

greeting=Hello

name=Tux

echo $greeting $name
```

Arithmetic expressions

Operator	Usage
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
%	modulus

Arithmetic expressions contd..

*Note the spaces, these are part of the syntax

```
expr 16 / 4
```

```
expr 20 - 10
```

Numerical expressions can also be calculated and stored in a variable using the syntax: *var=$((expression))*

Let's try an example.

```
#!/bin/bash
var=$((3+9))
echo $var
```

Operation	Syntax	Explanation
Equality	num1 -eq num2	is num1 equal to num2
Greater than equal to	num1 -ge num2	is num1 greater than equal to num2
Greater than	num1 -gt num2	is num1 greater than num2
Less than equal to	num1 -le num2	is num1 less than equal to num2
Less than	num1 -lt num2	is num1 less than num2
Not Equal to	num1 -ne num2	is num1 not equal to num2

How to read user input

Sometimes you'll need to gather user input and perform relevant operations.

In bash, we can take user input using the `read` command: *read variable_name*

To prompt the user with a custom message, use the `-p` flag.

```
read -p "Enter your age" variable_name
```

Conditional Statements

Syntax:

```
if [[ condition ]]
then
        statement
elif [[ condition ]]; then
        statement
else
        do this by default
fi
```

To create meaningful comparisons, we can use AND `-a` and OR `-o` as well.

The below statement translates to: If `a` is greater than 40 and `b` is less than 6.

```
if [ $a -gt 40 -a $b -lt 6 ]
```

Loops

Looping with numbers:

In the example below, the loop will iterate 5 times.

```
#!/bin/bash

for i in {1..5}

do

    echo $i

done
```

Looping with strings:

We can loop through strings as well.

```
#!/bin/bash

for X in cyan magenta yellow
do
        echo $X
done
```

While loop

While loops check for a condition and loop until the condition remains true. We need to provide a counter statement that increments the counter to control loop execution.

In the example below, ((i += 1)) is the counter statement that increments the value of i.

```
#!/bin/bash

i=1

while [[ $i -le 10 ]] ; do

   echo "$i"

  (( i += 1 ))

done
```

Putting it all together

How would you use shell scripting to solve the following problem? Within a folder named "lab13," which includes various .txt files, accomplish the following tasks:

1. Retrieve the names of all the .txt files in the "lab13" directory.

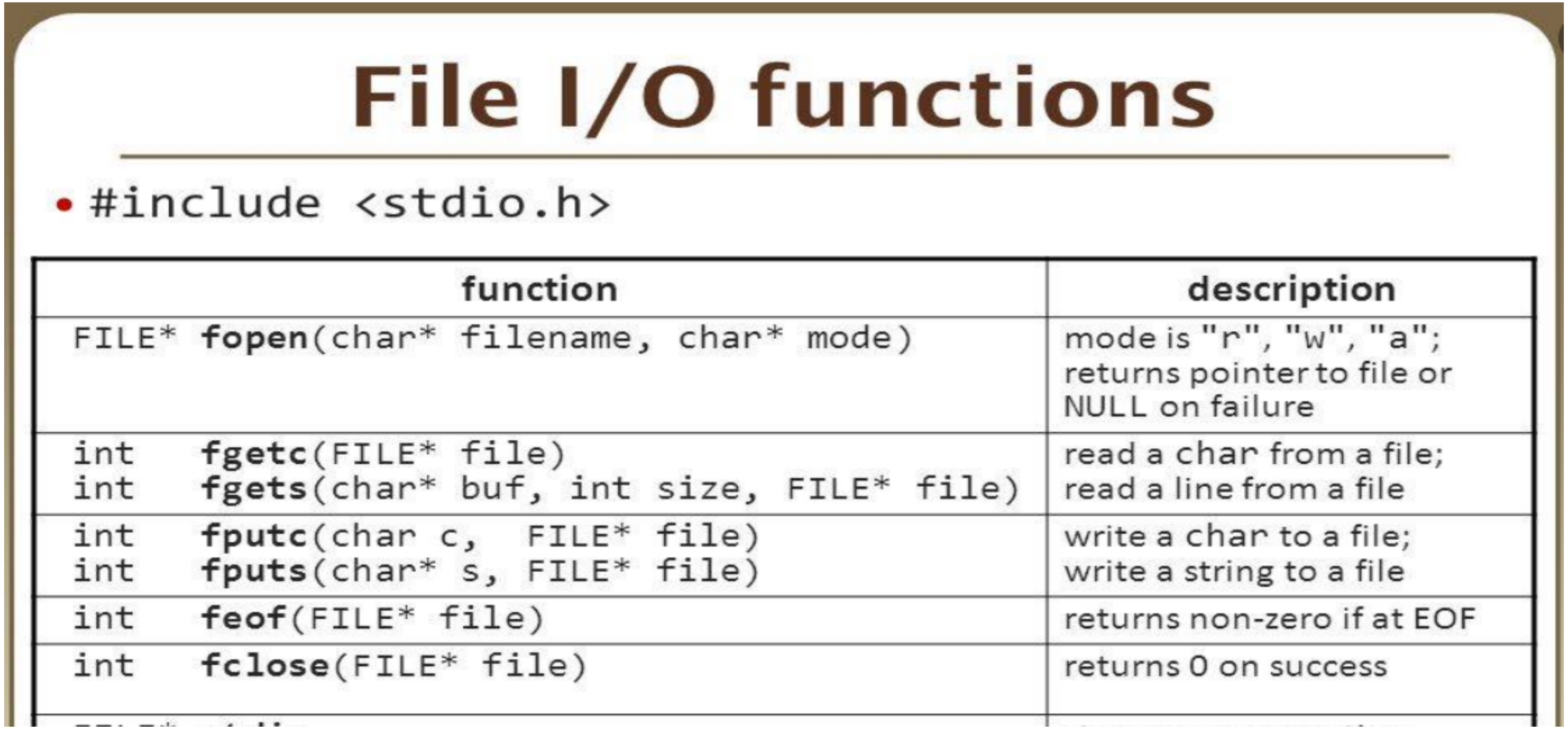
2. For each file, determine the number of occurrences of the word "hello."

3. Display the filename and the count of "hello" occurrences for each file.

4. Save the filename and the count of "hello" occurrences for each file in a separate file named "output.txt".

Makefiles and FILE I/O

Revising FILE I/O functions



File I/O functions

- `#include <stdio.h>`

function	description
`FILE* fopen(char* filename, char* mode)`	mode is `"r"`, `"w"`, `"a"`; returns pointer to file or NULL on failure

`int fgetc(FILE* file)`
`int fgets(char* buf, int size, FILE* file)`
	read a char from a file;
	read a line from a file

`int fputc(char c, FILE* file)`
`int fputs(char* s, FILE* file)`
	write a char to a file;
	write a string to a file

`int feof(FILE* file)`	returns non-zero if at EOF
`int fclose(FILE* file)`	returns 0 on success
[illegible]	[illegible]

Introduction to Makefiles

- Makefile is a script containing instructions for compiling and linking C programs.
- Used to automate the build process, managing dependencies and ensuring that only necessary files are compiled.
- Makefiles are especially helpful for large projects with multiple source files.

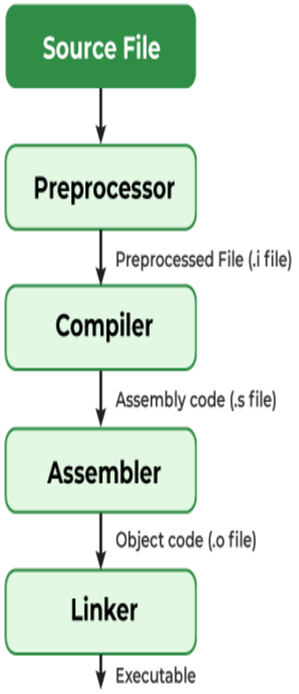
Compilation Process:Behind the scenes

Preprocessing:

- Before compilation, the preprocessor handles directives and macros in the source code.
- Command: `gcc -E source_file.c -o output_file.i`
- File Extension: `.i`

Compilation:

- In this stage, the compiler translates the preprocessed source code into assembly code.
- Command: `gcc -S input_file.i -o output_file.s`
- File Extension: `.s`



Compilation Process continued...

Assembly:

- Optionally, the assembly code can be converted into object code.
- Command: `gcc -c input_file.s -o output_file.o`
- File Extension: `.o`

Linking:

- Finally, the linker combines object files and libraries to produce the final executable.
- Command: `gcc input_file1.o input_file2.o -o output_file`

Basic Structure of Makefile

Target:

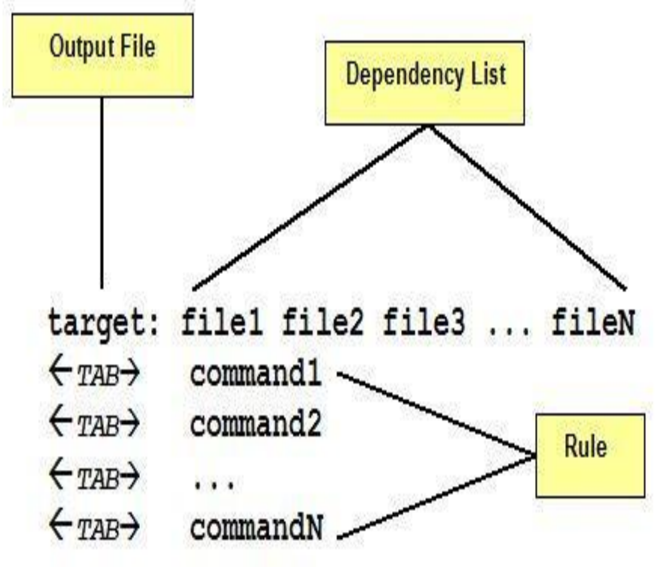
- Defines the output file or action to be performed.
- Typically corresponds to the name of the executable or object file.

Dependencies:

- Specifies the files required for building the target.
- If any dependency changes, the target is recompiled.

Commands:

- Contains the actions needed to build the target from its dependencies.
- Consists of shell commands preceded by a tab.



Dependency tree

A dependency tree is like a family tree for files in a project. Just as family members depend on each other, files in a project depend on one another to work correctly. The tree shows which files need others to function properly.

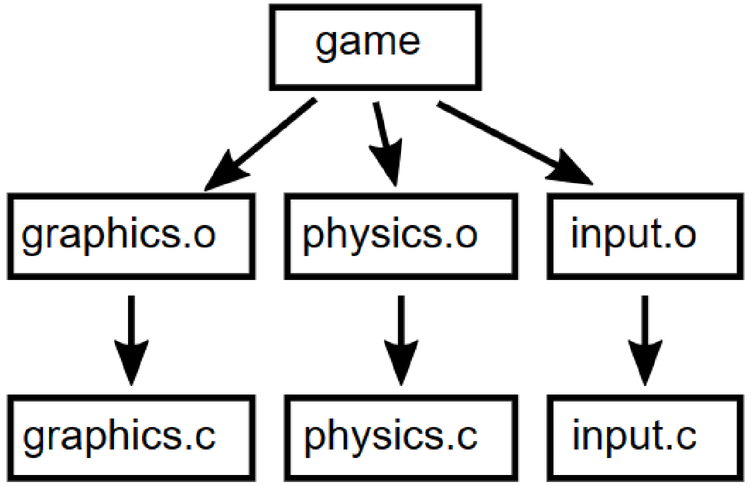
For example, think of a main program file as the "parent" in the tree. It needs other files, like helper files or data files, to do its job. These needed files are like its "children" in the tree.

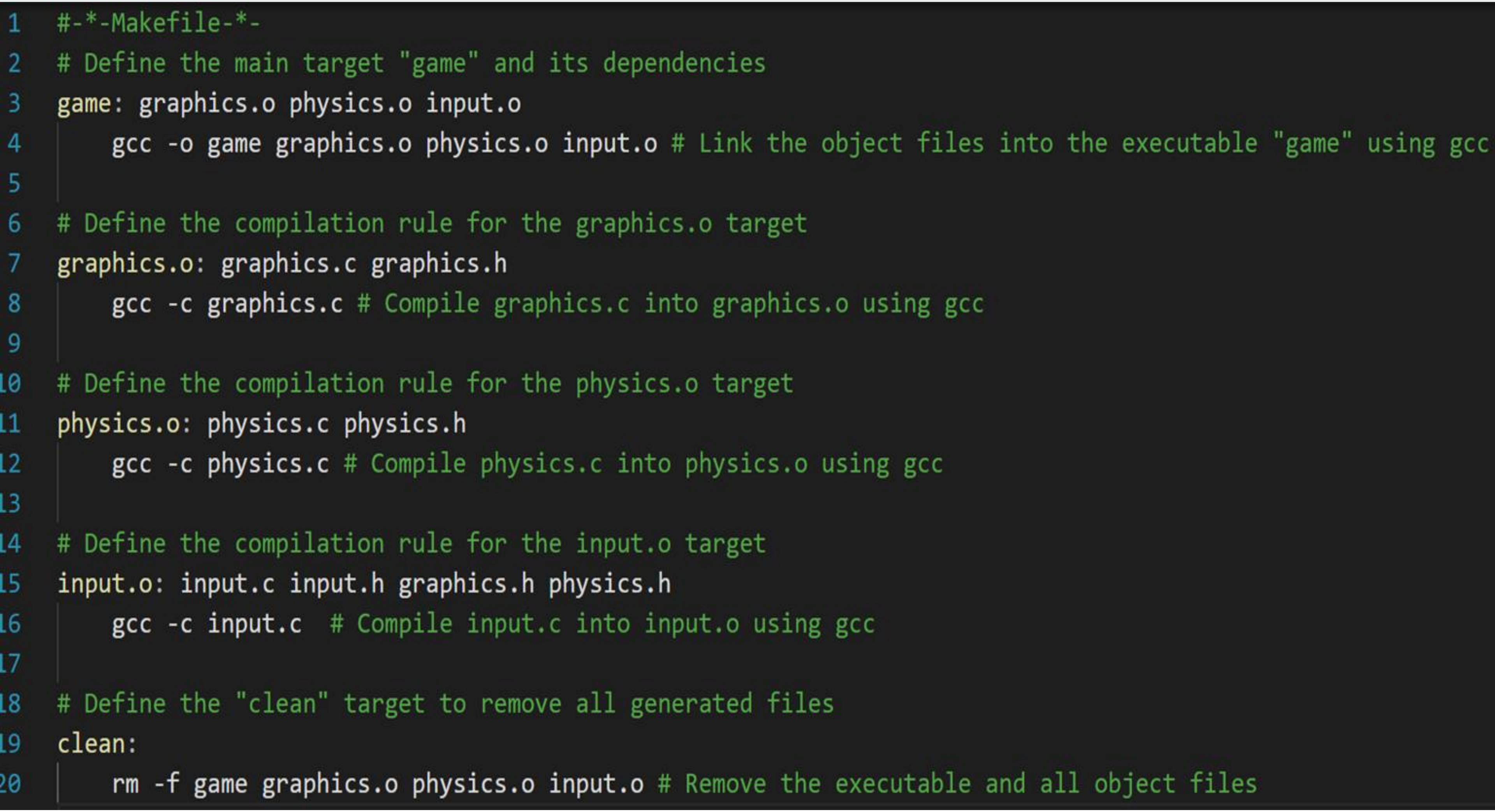
When we make changes to a file, the dependency tree helps us understand which other files might be affected. It shows us the connections between files so we can see how changes in one file might impact others.

Converting Dependency tree to Makefile

For compiling the given files we will have to write “gcc graphics.c physics.c input.c -o game” every time which is tedious.

Instead we can write a Makefile and then we can just pass the command “make” to compile.





```makefile
#-*-Makefile-*-
# Define the main target "game" and its dependencies
game: graphics.o physics.o input.o
	gcc -o game graphics.o physics.o input.o # Link the object files into the executable "game" using gcc

# Define the compilation rule for the graphics.o target
graphics.o: graphics.c graphics.h
	gcc -c graphics.c # Compile graphics.c into graphics.o using gcc

# Define the compilation rule for the physics.o target
physics.o: physics.c physics.h
	gcc -c physics.c # Compile physics.c into physics.o using gcc

# Define the compilation rule for the input.o target
input.o: input.c input.h graphics.h physics.h
	gcc -c input.c  # Compile input.c into input.o using gcc

# Define the "clean" target to remove all generated files
clean:
	rm -f game graphics.o physics.o input.o # Remove the executable and all object files
```