Shell Command & Shell Script

2023-2024

COMP2113B/C Programming Technologies / ENGG1340B/C Computer Programming II

Dr. Chenxiong Qian & Dr. T.W. Chim (E-mail: cqian@cs.hku.hk & twchim@cs.hku.hk)

Department of Computer Science, The University of Hong Kong

We are going to learn…

- Useful Shell commands
- Shell Script
  - A “Hello World” example
  - Variables
  - No quote, ‘single quote’ and “double quote”
  - String operations
  - Flow of control (if-else, for loop)
  - Mathematical operations
  - Arguments

What is the programming syntax of shell script?

Useful

Shell Commands

- *Please refer to Module 1 for details!*
- *You must try them in order to remember them!*

Directory manipulations

Command	Meaning
pwd	It prints the name of the **present working directory**.
ls ls -l	It lists the content in the present working directory. It lists the content in long format, which contains the file size, owners, last modification date, etc.
cd dir cd ~ cd .. cd .	It **changes** the current directory to dir. Changes to the home directory. Changes to the parent directory. Changes to the current directory. Hence, this command is valid yet has no effect actually.
mkdir dir	It **creates** a new directory with name dir.
rmdir dir	It **removes** the directory dir. This only works if dir is **empty**.
rm -rf dir	It **removes the non empty directory** dir and all the subdirectories & files.
mv dir dir2	If dir2 **does not exist**, it **renames** the directory from dir to dir2. Otherwise, it **moves** dir into dir2.
cp –r dir1 dir2	**copy** dir1 into dir2 including sub-directries

File manipulations 1

Command	Meaning
pico a.cpp	It starts a simple text editor to edit the file **a.cpp**. (Can use “Ctrl” + various letters to issue different control commands in it) You will learn another editor called **vi** or **vim** (Vi IMproved) in Module 1.
g++ a.cpp -o a.o	It invokes the g++ compiler to compile the program **a.cpp** into an executable **a.o**. Notice that an executable program does not need to have an extension .exe.
./a.o	It invokes the program **a.o**.

File manipulations 2

Command	Meaning
cp file1 file2	Copy file1 into file2.
mv file dir mv file1 file2 mv dir1 dir2	If dir is a directory, it moves the file into dir. If the two arguments are the same type (e.g., both file1 and file2 are files), it renames file1 to file2. The same for directories. If dir2 exists, then mv moves dir1 to dir2.
rm file rm –rf dir	Remove file. Remove recursively all files and directories in dir.
touch file	Create an empty file named file.
cat file	Display the content of file.

Others

Command	Meaning
wc file	It **counts** the number of lines, words, and characters in **file**.
sort file	It **sorts** the lines of **file** into alphabetical order.
cut -d, -f1 file	It returns specific **columns of data**. It divides each line according to the delimiter specified by the flag **–d**, and returns the column specified by the flag **–f** (the field number starts from 1).
grep 'abc' file	It **returns the lines** in file that contains "**abc**". More sophisticated pattern matching can be specified using regular expression (Please use with the flag **-E**).
uniq file	It **removes adjacent duplicate** lines so that only one of the duplicated lines remains.
diff file1 file2	Display lines that are **different** in **file1** and **file2**. Intuitively, **diff** matches all lines that are common in both files and displays those unmatched lines.
spell file	It displays all **incorrect words** in **file**.

Examples

- **What is the full path of your default directory when you startup your shell?**

```
$ pwd
/home/teacher/twchim
```

Your **present working directory** should be different.

- **What are the directories in the root directory?**

```
$ cd  /
$ ls
… list of directories …
```

- **How to go back to your home directory?**

```
$cd  ~
```

Examples

- **Copy the source code hello.cpp to hello2.cpp**

```
$cp hello.cpp hello2.cpp
```

- **Rename hello2.cpp to backup.cpp**

```
$ mv hello2.cpp backup.cpp
```

- **Create a directory “backup” and move backup.cpp in it.**

```
$ mkdir backup
$ mv backup.cpp backup
```

Wildcards

- **The Linux shell has a mechanism to generate a list of file names matching a pattern**

Wildcard	Meaning
*	Matches any **string** or nothing.
?	Matches any single **character**.

```
$mv hello.* backup
$cd backup
$ls hello.*
hello.cpp     hello.o
```

File permission & security

- You can use the list directory command ls -l to return the permission code of files / directories.

```
$ touch file
$ ls -l file
-rw-------.   1   twchim   gopher    0    Aug  24 14:00   file
```

File permission & security

Type	User permissions			Group permissions			Others permissions		
-	r	w	-	-	-	-	-	-	-

Type

- If it is a dash “**-**“, that means it is a normal file.
- If it is a “**d**”, it means it is a directory.

File permission & security

Type	User permissions			Group permissions			Others permissions		
-	r	w	-	-	-	-	-	-	-

- **User permissions**
  - 3 bits representing **Read** (**r**) , **Write** (**w**), **Execute** (**x**) permission of the file owner on the file.
  - Because the permission is “**rw-**“, the owner can **R**ead and **W**rite the file, but cannot e**x**ecute the file.

File permission & security

- To change the permission of the files/ directories

```
chmod [who]operator[permissions] filename
```

who

value	meaning
u	user (owner)
g	group
o	other
a	ALL (including user, group and other)

operator

value	meaning
+	Add permission
-	Remove permission
=	Set the permission

permissions

value	meaning
r	Read permission
w	Write permission
x	Execute permission

- Grant(**+**) execute(**x**) permission to user(**u**):
- Grant(**+**) read(**r**) and write(**w**) permission to all(**a**):
- Remove(**-**) read(**r**) and write(**w**) from group(**g**) and other(**o**)

```
$ chmod u+x file
```

```
$ chmod a+rw file
```

```
$ chmod go-rw file
```

Examples

- **List the permission of the files with prefix “hello.”**

```
$ ls -l hello.*
-rwx------.  1  twchim  gopher 76   Aug 23 9:30   hello.cpp
-rwx--x--x.  1  twchim  gopher 5981  Aug 23 9:30   hello.o
```

- **Take away the execute permission (x) on hello.o from user (u), what will happen?**

```
$ chmod u-x hello.o
$./hello.o
bash: ./hello.o: Permission denied
```

Shell Script

Please refer to Module 2 for details!

Motivation

You will learn lots of shell commands in Module 1. To execute a sequence of shell commands but don't want to re-type those commands every time, what can we do?

```
#!/bin/bash
g++ gen4.cpp -o gen4.o
./gen4.o < gen4_input.txt > gen4_output.txt
sort gen4_output.txt | uniq > sort_uniq.txt
spell sort_uniq.txt > misspell.txt
diff sort_uniq.txt misspell.txt | grep -E "^<"
```

gen4.sh

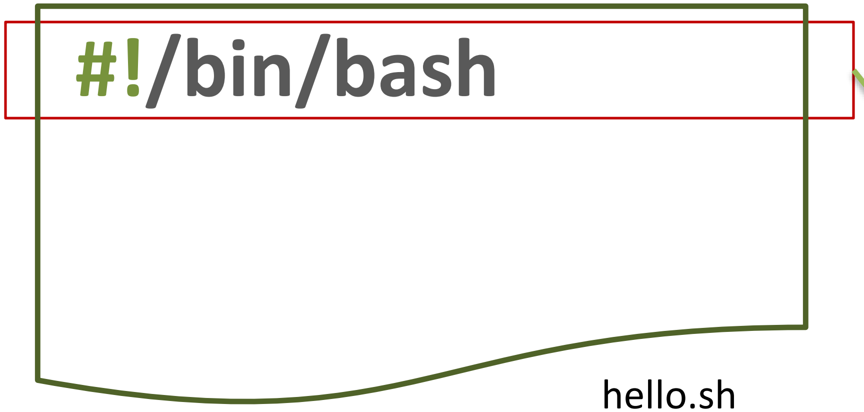
```
$./gen4.sh
 < loop
 < polo
 < pool
```

Answer: You can do it by saving these commands in a file. We call that a **shell script**.

A Hello World Example

My first shell script

- **#!/bin/bash** must appear in the first line of the shell script.
- The path after **#!** indicates which program should be use to process the shell script. In this case, it is the path to the **bash** program (we are using **bash shell**).



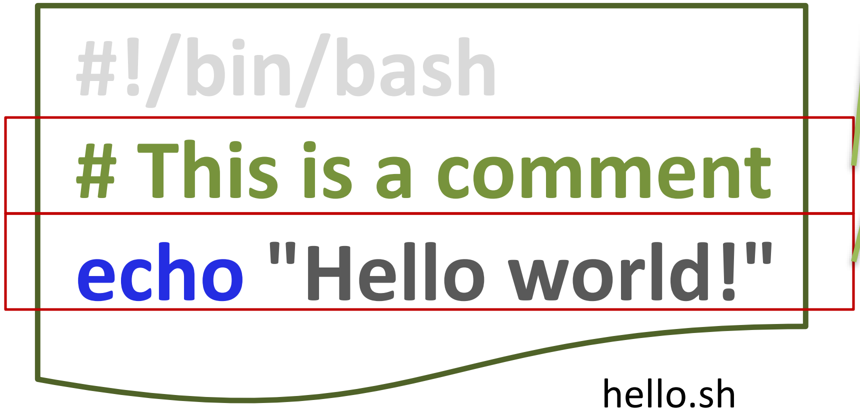
```
#!/bin/bash
```

hello.sh

#!/bin/bash

- Different machines may install the **Bash shell** at different location. You can use the **which bash** command to locate the correct path to the **Bash shell**.

Comments and echo



```
#!/bin/bash
# This is a comment
echo "Hello world!"
```

hello.sh

Commenting

- Except the first line, any string after the **#** sign are regarded as **comment** in the shell script.

echo "Hello world!"

- echo is actually a shell command!
  It prints the following string or the value of a variable.
- Use the flag **-n** if you do not want to output the trailing newline.
  echo -n "Hello World!"

Execute(x) permission to run

- The shell script has to be made executable by **granting** (**+**) the **execute** (**x**) permission to the script **user** (**u**).

No execute (x) permission!

- We need to have the execute permission to execute a shell script!

```
#!/bin/bash
# This is a comment
echo "Hello world!"
```

hello.sh

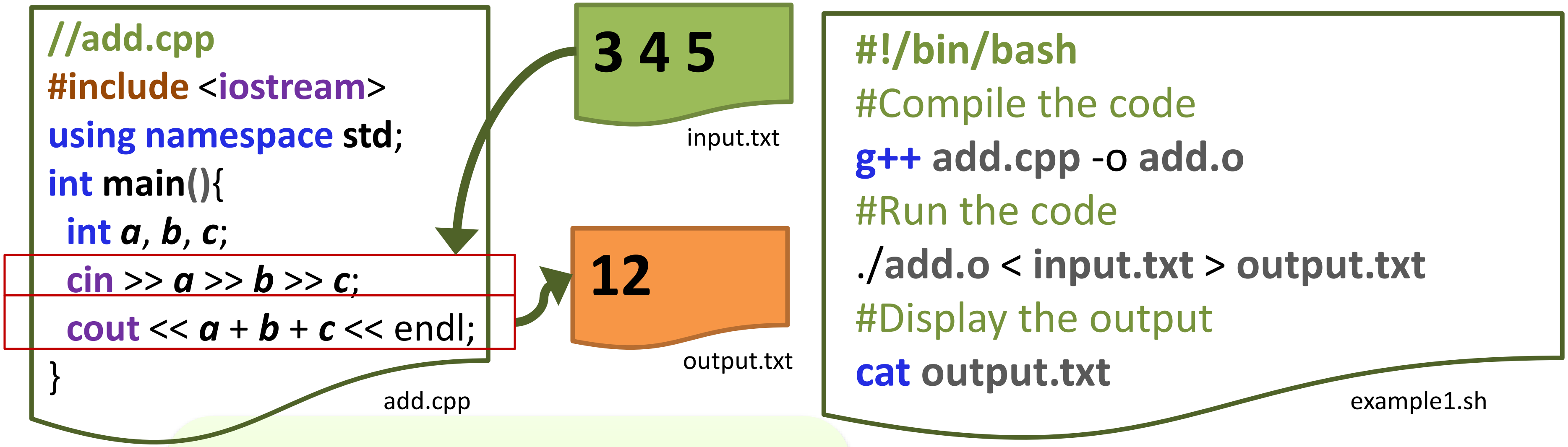
```
$ ./hello.sh
Bash: ./hello.sh: Permission denied
```

```
$ chmod u+x hello.sh
$ ./hello.sh
Hello World!
```

Shell script is very useful

- A common usage of shell script is to group a number of commands so that they can be repeated easily.

Note: < and > are used to redirect input from a file and redirect output to a file.



```cpp
//add.cpp
#include <iostream>
using namespace std;
int main(){
  int a, b, c;
  cin >> a >> b >> c;
  cout << a + b + c << endl;
}
```

add.cpp

```
3 4 5
```

input.txt

```
12
```

output.txt

```bash
#!/bin/bash
#Compile the code
g++ add.cpp -o add.o
#Run the code
./add.o < input.txt > output.txt
#Display the output
cat output.txt
```

example1.sh

VERY USEFUL ☺!
I can compile the code, run the executable, and display the result just by typing **./hello.sh**

```
$  chmod u+x example1.sh
$ ./example1.sh
12
```

An interpreted language

- Shell script is an **interpreted language**, but not a compiled language.
  - The program written in shell script is **parsed and interpreted by the shell every time the program is executed**.
  - Unlike C++, we do not need to compile the shell into a binary executable format before executing the program.
- Interpreted languages allow us to **modify the program more quickly** by simply editing the script.
  - However, the programs are usually **slower** because parsing and interpretation is needed during execution time.

Variables

string variable only

- There is only 1 variable type in shell scripts: **string**.
- Variable name is **case sensitive**.
- The following statement initializes a variable ***a*** with value "**cat**".

```
a="cat"
```

No space! No space!

IMPORTANT!!!!!!

There must be NO SPACE before and after the = sign.

- Use the dollar sign **$** to **retrieve the value** of a variable.

```
echo $a
```

A space

Spacing is critical!

✗ Space before and after =

- Any space **BEFORE** or **AFTER** = **will cause problem** as the shell will interpret the variable as a **command**.

```bash
#!/bin/bash
a = "Apple pie"
echo $a
```

example2a.sh

```
$  ./example2a.sh
./example2a.sh: line 2 : a: command not found
```

✓ NO space before and after =

- **[Setting value] NO $ sign** when **setting** the value of a variable.
- **[Retrieve value] Use $ sign** when **retrieving** the value of a variable.

```bash
#!/bin/bash
a="Apple pie"
echo $a
```

example2b.sh

```
$  ./example2b.sh
Apple pie
```

Variables

- We can use a variable **without declaration**.
- Bash shell **creates the variable automatically when a variable is used**.

The read command

- The **read** command reads a string from user input and stores it to the variable following the command.
- The variable ***name*** is automatically created, we do not need to declare it.

```
#!/bin/bash
echo "What is your name?"
read name
```

example3.sh

```
$./example3.sh
What is your name?
Chim
```

Variables

- We can use a variable **without declaration**.
- Bash shell **creates the variable automatically when a variable is used**.

Use $ when retrieve value

- **[Retrieving value]** A dollar sign is required when **retrieving** the value of a variable.

```
#!/bin/bash
echo "What is your name?"
read name
echo "Hello $name"
```

example3.sh

```
$./example3.sh
What is your name?
Chim
Hello Chim
```

Quoting

Specifying strings

- **Quoting** is very important on the command line and in shell script.
  - **Unquoted**
  - **'Single quote'**
  - **"Double quote"**

Unquoted

- We can specify a string value **without any quoting**, but this method only works if the string value consists of a **single word**.

```bash
#!/bin/bash
a=Apple
echo $a
b=Apple pie
echo $b
```

example4.sh

Error: Unquoted word with space

- With the space, "**pie**" is interpreted as a shell **command**, therefore the shell returns **command not found**.

```
$ ./example4.sh
 Apple
 ./example4.sh: line 4: pie: command not found
```

Single quote

- Any value between the pair of **single quote** will be set as value of the string.

```
#!/bin/bash
a='Apple pie'
echo $a
```

example5.sh

```
$./example5.sh
Apple pie
```

Single quote

- Any value between the pair of **single quote** will be set as value of the string.
  - However, it **does not support variable substitution**.

```
#!/bin/bash
a='Apple pie'
echo $a
b='$a\$'
echo $b
```

example5.sh

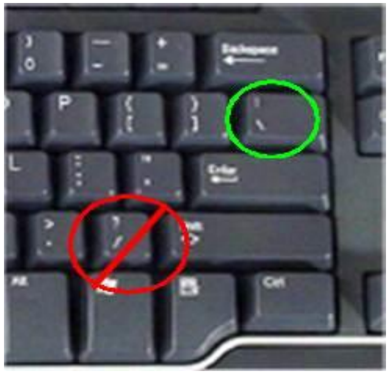
```
$./example5.sh
Apple pie
$a\$
```

Note that in '$a\$', $a is **NOT** substituted by the value of variable ***a*** if we use **single quote**.

Double quote

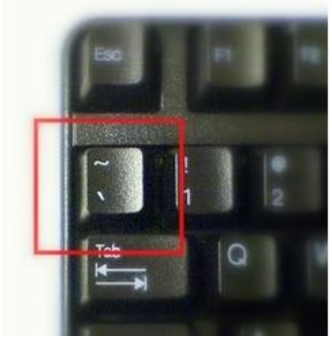
- Different from single quote, **double quote will handle three special characters** instead of directly including them into the strings.

Symbol	Meaning
$	**Dollar sign -** Variable substitution.
\	**Backslash -** Escape special character.
\` \`	**Back quotes -** Enclose bash commands



Backslash

Where is the back quote button on keyboard?



Back quote

Double quote

```bash
#!/bin/bash
a="Apple pie"
b="$a"
echo $b
```

example6.sh

```
$./example6.sh
Apple pie
```

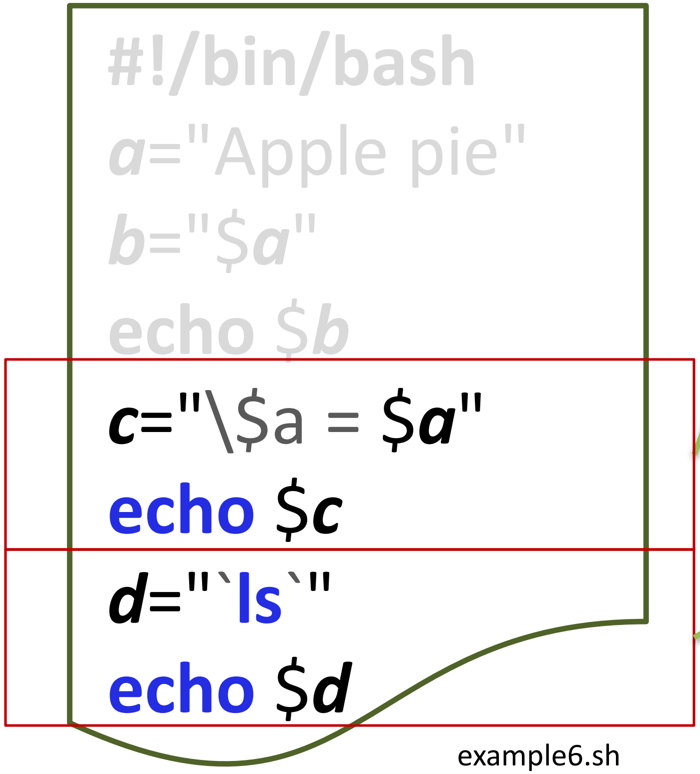
Supports value with space

- We can use double quote to mark value with space.

Supports variable substitution

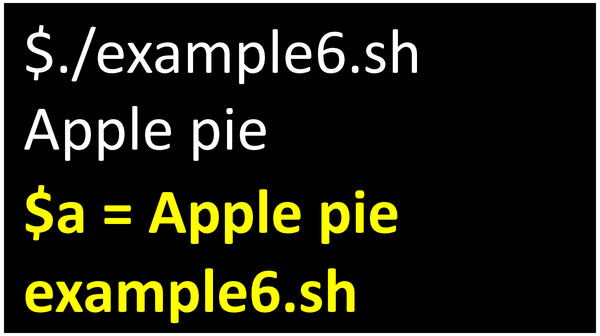
- Double quote supports **variable substitution**, therefore the value of variable ***b*** is "**Apple pie**" but **NOT** "**$a**".

Double quote



```
#!/bin/bash
a="Apple pie"
b="$a"
echo $b
c="\$a = $a"
echo $c
d="`ls`"
echo $d
```

example6.sh



```
$./example6.sh
Apple pie
$a = Apple pie
example6.sh
```

Supports escape characters

- We can include **escape characters** inside double quotes.
- E.g., **\$** is interpreted as a single dollar sign character so that **\$a** will not be substituted with the value of variable $***a.***

\`Back quote\` = shell **command**!

- \`Back quote\` mark the **shell command**.
- Therefore, the **\`ls\`** will be executed as shell **command** and replaced by **the result of the command**.
  (i.e., **\`ls\`** returns the directories and files in the current directory, which is **example6.sh** in this example)

Double quote

- With **back quotes** \` \`, we can store the output of a shell **command** in a variable for further processing.

```
Apple
Banana
Cherry
```

file

```
#!/bin/bash
a="`cat file`"
echo $a
```

example7.sh

```
$ ./example7.sh
Apple Banana Cherry
```

Question:
How about processing **each word** inside the file?

Answer:
We need some “**for** each” mechanism, we will talk about that shortly.

String operations

String Operation

- We can perform a number of operations on strings.
  - Get string length
  - Substring
  - Replace a part of a string

String length

- **String length.** Given any string ***a***, the following returns the number of characters in ***a***.

$$\boxed{\$\{\#a\}}$$

```
#!/bin/bash
a="Apple"
echo "The length of \"$a\" is ${#a}"
```

example8.sh

```
$./example8.sh
The length of "Apple" is 5
```

\" is an escape character

${#a} returns the string length (i.e., 5)

Substring

- **Substring (use “:”).** Given any string ***a***, the following returns the substring of ***a*** starting from position ***pos*** and has length ***len***.

```
${a:pos:len}
```

```
#!/bin/bash
a="Pine apple"
echo "$a substring 5:5 is ${a:5:5}"
```

example9.sh

```
$./example9.sh
Pine apple substring 5:5 is apple
```

P	i	n	e		a	p	p	l	e
0	1	2	3	4	5	6	7	8	9

Note: The first character in string is having index 0.

Replace

- **Replace** (use “/”). Given any string ***a***, the following returns the string formed by replacing the **first occurrence** of ***from*** with ***to***.

$$\boxed{\$\{a/from/to\}}$$

```
#!/bin/bash
a="Apple pie"
from="pie"
to="juice"
echo -n "Apple pie after replace \"$from\" by \"$to\" "
echo "becomes ${a/$from/$to}"
```

example10.sh

```
$./example10.sh
Apple pie after replace "pie" by "juice" becomes Apple juice
```

Note: with the flag **-n**, **echo** will not have an endline so the next **echo** will continue output on the same line.

Flow of Control

If-else statement

- The basic syntax of the if-statement is shown below.

```
if [ condition ]
then
#perform some action
fi
```

```
if [ condition 1 ]
then
   echo "Action 1"
elif [ condition 2 ]
then
   echo "Action 2"
else
   echo "Action neither"
fi
```

[condition] for string

- We can perform string comparison in the conditions.

String comparisons	Meaning
["$*string*"]	**True** iff the length of $***string*** is non-zero
["$*string1*" **==** "$*string2*"]	**True** iff the strings are equal
["$*string1*" **!=** "$*string2*"]	**True** iff the strings are different
["$*string1*" **\>** "$*string2*"]	**True** iff $***string1*** is sorted after $***string2***
["$*string1*" **\<** "$*string2*"]	**True** iff $***string1*** is sorted before $***string2***

Notice that we enclose $***string1*** or $***string2*** with **double quote** so that comparison can **work even if there are spaces inside** $***string1*** or $***string2***.

Spacing is critical!

```
if ["$ans"=="Y"]
```

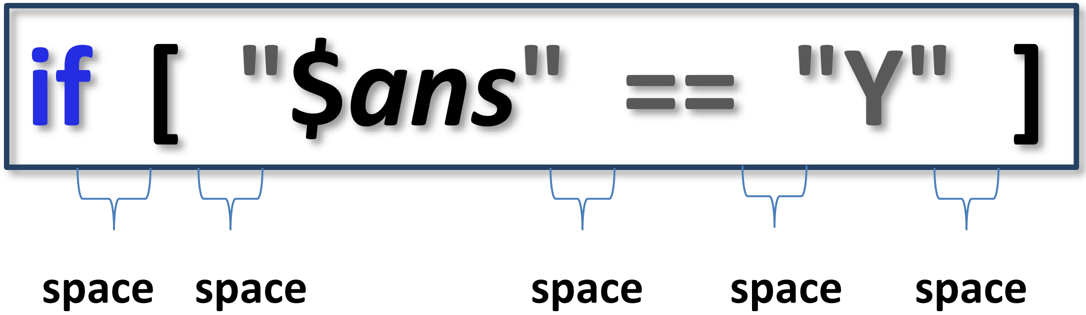
✗

- If there is **NO SPACE** between the items, the shell will mis-interpret them as a shell **command**. Therefore returning **“Command not found”**

```
$./example11.sh
Do you want to remove all .cpp files? (Y/N)
Y
./example11.sh: [Y==Y] command not found
```

```
#!/bin/bash
echo "Do you want to remove all .cpp files? (Y/N) "
read ans
if ["$ans"=="Y"]
then
    rm -rf *.cpp
    echo "All .cpp files are removed!"
fi
```

example11.sh (wrong)



```
if [ "$ans" == "Y" ]
```

space space space space space

✓

[condition] for file

- It is also very convenient to do file checking in the conditions.

File checking	Meaning
[-e $*file*]	True iff ***file*** **exists**.
[-f $*file*]	True iff ***file*** **is a file**.
[-d $*file*]	True iff ***file*** **is a directory**.
[-s $*file*]	True iff ***file*** **has size > 0**.
[-r $*file*]	True iff ***file*** **is readable**.
[-w $*file*]	True iff ***file*** **is writable**.
[-x $*file*]	True iff ***file*** **is executable**.

Note: the file permission testing is on the one who executes the script.

[condition] for file

- If **hello.cpp** does not exist:

```
$./example12.sh
hello.cpp not found!
```

Let’s try to implement a script to **compile** and **run** **hello.cpp**, and return error message(s) if any.

```
#!/bin/bash
if  [ -e hello.cpp ]
then




else
    echo "hello.cpp not found!"
fi
```

example12.sh

[condition] for file

- If **hello.cpp** is error free

```
$./example12.sh
Hello World!
```

```
#!/bin/bash
if  [ -e hello.cpp ]
then
    rm *.o
    g++ hello.cpp -o hello.o
    if [ -e hello.o ]
    then
        ./hello.o



    fi
else
    echo "hello.cpp not found!"
fi
```

example12.sh

[condition] for file

- If **hello.cpp** contains compilation error:

```
$./example12.sh
Compilation failed!
(compilation errors
returned by the compiler…)
```

```
#!/bin/bash
if  [ -e hello.cpp ]
then
    rm *.o
    g++ hello.cpp -o hello.o 2> error.txt
    if [ -e hello.o ]
    then
        ./hello.o
    else
        echo "Compilation failed!"
        cat error.txt
    fi
else
    echo "hello.cpp not found!"
fi
```

example12.sh

[condition] for command

- The condition can be a **shell command**.
- The condition is evaluated to true if the command is executed successfully.

```
$  ./example13.sh
cp: cannot stat `file'123 : No such file or directory
Command failed
$ touch file123
$ ./example13.sh
Command executed successfully
$ls file*
file123 fileabc
```

```
#!/bin/bash
if cp file123 fileabc
then
  echo "Command executed successfully"
else
   echo "Command failed"
fi
```

example13.sh

for loop

- We use a **for**-loop to iterate through a list of strings.

```
#!/bin/bash
list="1 2 3 4 5"
for i in $list
do
  echo "This is iteration $i "
done
```

example14.sh

```
$./example14.sh
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
```

for loop

```
#!/bin/bash
list=`ls *.cpp`
for name in $list
do
  cp $name  "$name.backup"
done
```

backup.sh

`ls *.cpp`

- The command ls *.cpp returns all files in the current dir with .cpp as the file suffix (file extension).

```
$ touch a.cpp b.cpp
$./backup.sh
$ ls *.cpp*
a.cpp  a.cpp.backup
b.cpp  b.cpp.backup
```

With **for** loop, you can write a script to backup all .cpp files in the current directory ☺

Mathematics operations

Mathematics operations

- It is less common to use shell scripts for mathematical calculations. Nevertheless, we can still perform mathematical operation using the **let** command.

```
#!/bin/bash
a=10
let "a=$a*$a-$a/$a"
echo $a
```

example15.sh

```
$./example15.sh
99
```

[condition] for numbers

- If we want to perform mathematical comparisons in the conditions of the **if**-statement, special syntax is needed.

Integer comparisons	Meaning
[$***a*** **-eq** $***b***]	**True** if ***a*** = ***b***
[$***a*** **-ne** $***b***]	**True** iff ***a*** != ***b***
[$***a*** **-lt** $***b***]	**True** iff ***a*** < ***b***
[$***a*** **-le** $***b***]	**True** iff ***a*** <= ***b***
[$***a*** **-gt** $***b***]	**True** iff ***a*** > ***b***
[$***a*** **-ge** $***b***]	**True** iff ***a*** >= ***b***

Mathematics operations

- Note that shell scripts compare strings and integers differently.

```
$./example16.sh
By string comparison,
99 is larger
By integer comparison,
100 is larger
```

```
#!/bin/bash
a=100
b=99
echo "By string comparison, "
if [ $a \> $b ]
then
  echo "$a is larger"
else
  echo "$b is larger"
fi

echo "By integer comparison, "
if [ $a -gt $b ]
then
  echo "$a is larger"
else
  echo "$b is larger"
fi
```

example16.sh

Arguments

Getting arguments

- **Command line arguments** are labeled as ***$0***, ***$1***, ... up to ***$9***. In particular, ***$0*** is the name of the shell script.
  - Command line arguments after ***$9*** are labeled as ***${10}***, ...
  - **$#** returns the number of arguments when user execute the shell script.

```
$./example17.sh sun mon tue
There are 3 arguments
$0 = ./example18.sh
$1 = sun
$2 = mon
$3 = tue
```

```
#!/bin/bash
echo "There are $# arguments"
echo "\$0  = $0"
echo "\$1  = $1"
echo "\$2  = $2"
echo "\$3  = $3"
```

example17.sh

END

2023-2024

COMP2113B/C Programming Technologies / ENGG1340B/C Computer Programming II

Dr. Chenxiong Qian & Dr. T.W. Chim (E-mail: cqian@cs.hku.hk & twchim@cs.hku.hk)

Department of Computer Science, The University of Hong Kong