ENGG1340

Notes for HKU · Spring 2024

Author: Jax

Contact: enhanjax@connect.hku.hk

MORE notes on my website!

1 About Linux

Linux is important because:

1. Open-source software, can edit however we want.
2. A lot of software development tools are available, software development is easier.
3. Linux is free

1.1 Access school Linux (ubuntu)

Access the HKU CS server's Ubuntu Linux system to try out system commands by running the following commands on your computer:

Terminal → `ssh cs_account@gatekeeper.cs.hku.hk` → `ssh cs_account@academy21.cs.hku.hk`

2 System commands

Compiling C++ programs

Use the command `g++ <input.cpp> -o <output>` to compile the input script into the output executable (extensionless).

2.1 Basic file management

Command	Meaning
`pwd`	**p**rint**w**orking**d**irectory
`ls (-la)`	**l**i**s**t
`cd <dir>`	**c**hange**d**irectory
`mv <dir> <dir2>`	**m**o**v**e
`cp (-r)  <file/dir> <file2/dir2>`	**c**o**p**y
`rm (-rf)  <dir>`	**r**e**m**ove
`mkdir <name>`	**m**a**k**e**dir**ectory
`rmdir <dir>`	**r**e**m**ove**dir**ectory (empty only)
`touch <file>`	New file in .
`cat <file>`	Show file content

Dir syntax	Meaning
`~`	Home
`.`	Current
`..`	Back

- The flag `-r` means to *recursively* execute, which the children of the directory will be affected as well.
- The flag `-f` means to forcefully execute.
- The flags `-la` means to show details and hidden files in output of `ls`.

2.2 File permissions

The permission string returned by `ls -l` consists of 10 characters: `- --- --- ---`.

- 1st char represents file type `-/d`
- 2nd char represents `u`
- 3rd char represents `g`
- 4th char represents `o`

Use the following command to edit permissions: `chmod ([u/g/o][+/-/=][rwx]),... <file>`

Key	**Meaning**
`-`	Not / don’t have
`d`	Directory type
`r`	Read permission
`w`	Write permission
`x`	Execute permission
`u`	User’s permissions
`g`	Group’s permissions
`o`	Other’s permissions
`+`	Add
`-`	Remove
`=`	Set to

2.3 File reading and modification

Command	**Description**
`wc (-lw)  <file>`	Return `[lines] [words] [bytes]` `-l` lines only `-w` words only
`cut -d’’ -f #,... <file>`	Return **columns of data** `-d’’` Delimiter character `-f #,...` Fields / columns #numbers array
`sort (-t’’ -k# -nr)  <file>`	Return **sorted data** `-t’’` field separator `-k#` Column #number to sort by (starting from 1) `-n` Numeric sort `-r` Reversed sort (big to small)
`uniq <file>`	Return **removed adjacent duplicate lines**
`diff <file> <file2>`	Return ***process* to transform file to file2**

The `diff` command outputs process, and it can inteperted with the following meaning:

Key	**Meaning**
`a`	Add
`d`	Delete
`c`	Change
`#,...[a/d/c]#,...`	Operate `a/d/c` line #number from file2 to file line #number
`> ’’`	Line content `’’` added
`< ’’`	Line content `’’` removed

2.4 Editing files using vi in the terminal

Command	Description
`vi <file>`	Open file
`:w`	Save
`:q`	Exit
`:wq`	Save and exit
`:q!`	Exit without saving

After opening a file in vi, the user is in command mode, which they could use the commands starting with `:` to edit the file. To enter insert mode, press `i`.

2.5 File IO

We can use the redirection operators to get the input and output of commands:

Key	Meaning
`> <file>`	Save output
`>| <file>`	Force overwrite output
`>> <file>`	Append output
`< <file>`	Get input from
`2>/2>>`	Save / append error
`<cmd> | <cmd2>`	Send cmd output to cmd2 as input (piping)

Note that the commands can be chained. Here's an example: `command < fileA > fileB 2>fileC`

2.6 Searching

Key	Meaning
`find <dir> -name ''`	Find files in dir by matching name
`grep (-E) '<(regex)/text>' <file>`	Find lines in file matching (regex)/text

2.6.1 Regular Expressions

Regular expressions, also known as regex, are a powerful tool for matching patterns in text. They are used in various programming languages and tools for tasks such as searching and replacing text.

Key	**Matches**
`.`	any character except newline
`*`	0 or more of the preceding token
`+`	1 or more of the preceding token
`?`	0 or 1 of the preceding token
`[abc]`	any character in the set
`[^abc]`	any character not in the set
`(abc)`	the pattern abc
`{n}`	n occurances of the preceding pattern
`^`	the start of the line
`$`	the end of the line

Note: The above keys are some of the most commonly used in regular expressions, but there are many more. The exact syntax and features available can vary between different programming languages and tools.

3 Git version control

3.1 Stage and snapshot commands

Command	Description
`git init`	Initializes new Git repo
`git clone`	Clones repo into a new directory
`git add .`	Adds all file changes in the working directory to the staging area
`git commit -m ''`	Commit all changes from staging area

We can use the `-m ''` flag to add a message to the action.

3.2 Branching commands

When we initialise a repo, we are working on a single branch called the **master** branch. The latest commit is usually called **HEAD**.

Branches can be created to allow changes to be made on different parts of a project **simultaneously**.

Command	Description
`git branch`	Lists all local branches in the current repo
`git branch <name>`	Make new branch
`git merge <branch>`	Combines branch history into the current branch
`git checkout (-b) <branch>`	Switches branches (after creating branch `branch`)
`git log`	Shows commit logs

We can also add messages to them: `git merge master -m 'Applied changes to master'`.

If the branch we're merging from hasn't been modified (up-to-date), the merge is called a *fast-forward merge*.

3.3 Sharing and updating commands

Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

Command	Description
`git remote`	Lists all remote repositories for the current repo
`git remote add <name> <url>`	Add new remote repo
`git push <remote> <branch>`	Uploads / update branch to the remote repo. `--force` Push even if it results in a non-fast-forward update.
`git pull <remote> <branch>`	Downloads / update branch from the remote repo. `--nocommit` Don't create new merge commit
`git fetch <remote> <branch>`	Fetches the branch from the remote repo

4 Shell scripting

A *bash* shell script has the extension `.sh`. It is a file containing a series of commands that the shell will execute.

All the scripts should start with #!. It indicates which program should be used to process the shell script.

Command	Description
`echo`	Print parameters as strings. `-n` No trailing new line in output.

4.1 Basics syntax

Read user input

Use `read <varname>` to read user input and store it in the variable.

Variables

Define variables with `varname=<val>`. Use `$varname` to access the value of the variable.

Command line arguments

We can pass in arguments to the bash file when we execute it. We can retrieve the variables by `$1, $2, $3...` and use `$#` for number of arguments.

Quotations

Double quotations allows variable substitution, but single quotations doesn't.

```bash
#!/bin/bash
val="world"
echo "Hello $val" # Hello world
echo 'Hello $val' # Hello $val
```

Command substitution

Use `$(<cmd>)` or `` `<cmd>` `` to substitute the output of the command.

Calculations

Use `$((<expr>))` or `let "varname=<exp>"` to evaluate the expression.

4.2 String operations

For the `$s` string variable:

Operation	Description
`${#s}`	Get the length of `$s`
`${s:<pos>:<len>}`	Extract substring at pos with length len
`${s/<from>/<to>}`	Replace the *first occurance* of matching

4.3 Control flow

4.3.1 Conditional statements

```
if [ condition1 ]
then
      echo "condition 1 met"
elif [ condition2 ]
then
      echo "condition 2 met"
else
      echo "No condition met"
fi
```

4.3.2 Conditional expressions

We must enclose the strings in double quotations so that comparison can work even if there are spaces in the strings.

Expression	True if
`[ "" ]`	length ¿ 0
`[ "" = "" ]`	strings are equal
`[ "" != "" ]`	strings are not equal
`[ #n -eq/ne/lt/le/gt/ge #n ]`	numbers are **g**reater / **l**ess (**t**han) ((**n**ot)**e**qual)
`[ -e/f/d/s/r/w/x <dir> ]`	dir **e**xists / is_**f**ile / is_**d**ir / **r**eadable / **w**ritable / e**x**ecutable

4.3.3 For-loops

The for-loop iterates through the list of items and executes the commands in the loop body for each item.

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

5 C++ basics

Basic differences to Python:

1. C++ is a *compiled* language, which means that the code needs to be compiled before it can be run.
2. C++ is a *statically typed* language, which means that the type of a variable is known at compile time.
3. C++ expressions end with a semicolon `;`.

5.1 Types

```
char 'b' // only one character
int 1 // integer
float 1.0 // floating point number
double 1.0 // double precision floating point number
bool true // boolean

const int a = 1; // constant variable that cannot be changed
```

5.2 Operators

```
n = ++i;      // equiv to i = i + 1, n = i;
n = i++;      // equiv to n = i, i = i + 1;
n = 1 && 0;   // logical and
n = 1 || 0;   // logical or
n = !3;       // logical not, converts any number > 0 to 0 (!3 => 0)

// C++ contains the usual operators: +, -, *, /, %
10 / 3 // If both types are integers, the result is an integer (3)
10.0 / 3 // If either type is a float, the result is a float (3.3333)
```

5.3 IO

A simple C++ program looks like this:

```
#include <iostream>  // lib provides the cout, endl funcs
using namespace std; // use the standard namespace
int main()
{                                   // functions are contained in {}
    char a;                         // limits the input to be a char
    cin >> a;                       // read input from user
    cout << "Hello " << a << endl; // prints Hello <input>
    return 0;                       // return 0
}
```

1. `>>` is the *extraction operator*
2. `<<` is the *insertion operator*
3. We can use `\` to escape special characters
4. `endl` is a special character that represents a new line, equiv to `\n`

5.4 Control flow

5.4.1 Basisc conditionals

```
if (a == 1) {
    // do something
} else if (a == 2) {
    // do something else
} else {
    // do something else
}
```

5.4.2 Switches

```
switch (ctrl_expr) {
    case const_1: // if ctrl_expr == const_1
        break; // man
    case const_2: // if ctrl_expr != const_1 && ctrl_expr == const_2
        break;
    default: // if no case matches (optional)
        break;
}
```

5.4.3 Tentary operators

```
(condition ? if_true : else_false) // returns the corresponding expression
```

5.4.4 Loops

```
for (int i = 0; i < 10; i++) { // initialize, check conditions, increment
    // do something 10 times
    continue;   // skip the rest of the loop
    break;      // break out of the loop
}
```

```
while (a < 10) { // check conditions first
    // do something
}

do { // execute first
    // do something
} while (a < 10); //then check conditions
```

6 Compilation and makefile

6.1 Separate compliation

We can use header files to separate the declaration and implementation of functions. This allows other programs to reuse the function.

A header file is a file with the extension `.h` that contains the function declarations. The implementation is in the `.cpp` file.

A simple header file looks like this:

```
// header.h
#ifndef HEADER_H
#define HEADER_H

int add(int a, int b);

#endif
```

Separate compliation has the following advantages:

- Faster compilation
- Easier to manage
- Easier to debug

6.2 Using makefile

`makefile` is used for describing the dependency of files for the `make` tool which smartly recompiles only the files that have changed. The file name is important, it must be named `makefile` or `Makefile` with no extension.

6.3 Makefile syntax

Here's a complete example of `makefile`, all you need to know is explained by the comments:

```
FLAGS = -pedantic-errors -std=c++11 # flags to pass in when compiling

gcd.o: gcd.cpp gcd.h # target: dependencies
	g++ $(FLAGS) -c gcd.cpp # <tab> action
# Target: gcd.o
# Dependencies: gcd.cpp gcd.h
# Action: Compile gcd.cpp into gcd.o

gcd_main.o: gcd_main.cpp gcd.h
	g++ $(FLAGS) -c $< # $< represents is the first dependency

gcd: gcd.o gcd_main.o
	g++ $(FLAGS) $^ -o $@ # $^: all dependencies, $@: target

clean: # a shorthand function to clean up
	rm -f gcd gcd.o gcd_main.o gcd.tgz

tar: # a shorthand function to tar (compress) the files
	tar -czvf gcd.tgz *.cpp *.h

.PHONY: clean tar # Declare clean and tar targets as phony targets, which
   are not files by rather shorthands
```