Linux Development Environment

Part 1. Development environment

To build a complete development environment, we need the following items :

1. remote mechanism `ssh` and `puTTY`
2. shell `bash`, `ksh` and shell script
3. editor `vim` and `nvim`
4. compiler `gcc`, `g++`, `clang`, `make`, `ninja` and `cmake`
5. debugger `gdb`, `gdb-tui` and core dump ⇐ *can be replaced by IDE*
6. tester `libgtest.a` and valgrind
7. profiler by native `clock_gettime()`
8. version control `git`, `gitlab` and `bitbucket`
9. documentation `doxygen`
10. other stuffs : `meld` (for directory comparison), `vscode`, PostgreSQL

In CDOI

- from local machine running windows, remote desktop to windows server
- with that windows server, we can use email, office, internet browser
- from that windows server, remote desktop to Redhat linux server (with GNOME)
- from same windows server, process control (using `ssh`) to start TraderRun

In JP-EMM

- from local machine running LVDI (virtual desktop infrastructure) login to windows server (my workspace)
- with my workspace, we can use email, office, internet browser, visual studio and bitbucket
- from my workspace, `ssh` (using `puTTY`) to Redhat linux `DEV` box
- from my workspace, `ssh` (using `puTTY`) to jumphost, then `ssh` (without `puTTY`) again to Redhat linux `PROD` box

- jumphost is not a software, it is a linux server with minimal command set, allowing user to `ssh` server with higher security

Technology stack	CDOI	JP-EMM	Yubo
local machine	windows	LVDI my workspace	Ubuntu18.0.4
remote method	remote desktop and `ssh`	`puTTY`, `ssh` and jumphost	`puTTY`, `ssh`
linux	Redhat with GNOME	Redhat with `ksh`	
C++ compiler	`gcc` and `g++`	`gcc7.3` and `g++7.3`	`g++10.1`
C++ debugger	`gdb` and `valgrind`	`gdb` and `valgrind`	`gdb` and `valgrind`
C++ IDE	Eclipse CDT	`vim`, `cmake` and `ninja`	`nvim`, `cmake` and `make`
source control	subversion SVN	bitbucket and `git`	gitlab and `git`
other tools		confluence - wiki	
		symthony - communication	
		pyramid - authenication system	

Linux desktop

- Linux GUI (or desktop) is installed separately as X Window System or X11
- Linux GUI (or desktop) example : GNOME and KDE

Hostname and user ID are two different things :

- the former is for machine, such as `DEV` box `psia0p549` and `hkxu3031`, `PROD` box `emm_xxxx`
- the latter is for account, such as personal account `n733607` and shared accounts `a_emm_uat`, `a_emm_prod`, `a_emm_prod_ro` (`ro` read only)

`puTTY DEV` box

`hostname = psia0p549` or `psia0p549.svr.apac.jpmchase.net`

`hostname = hkxu3031` or `hkxu3031.svr.apac.jpmchase.net`

`login    = n733607` small letter `'n'`

`passcode =` single sign on SSO

`puTTY PROD` box through jumphost

`hostname = psia0p549` or `psia0p549.svr.apac.jpmchase.net`

`login    = n733607` small letter `'n'`

`passcode = securID` token generated from mobile `myTecHub`

Development environment setup in Yubo 20200825

Ubuntu 18.0.4 is also called Bionic. Shortcuts in Ubuntu 18.0.4 include (these commands are not available in `ssh`) :

- create new shell windows `ctrl alt t`
- create new shell tab in same shell windows `ctrl shift t`
- switch between different applications `alt tab`
- switch between different shell windows `alt ~`
- switch between different shell tab in same shell windows `ctrl pageup`
- invoke command line and then launch terminal / view system log `alt F1` then type `terminal` or type `log`
- install `vimium` for browser to web access with `vim` liked experience

Two main methods for installing linux softwares :

- install using `apt install` (however the packages installed this way are usually outdated ...) or
- download using `wget` or `curl` or clicking link provided, then build it yourself using `cmake + make` or `bootstrap` for `boost` library

Ubuntu

```
>> sudo add-apt-repository ppa:ubuntu-toolchain-r/test
>> sudo apt update                (which update packages information)
>> sudo apt install gcc-11 g++-11 (which includes gdb)
>> sudo apt install python3
```

Vim and NeoVim

```
>> sudo apt install vim
>> sudo apt install neovim
>> sudo apt install python3-neovim
>> sudo apt install python3-pip                                   python module installer, preparation for pynvim
>> pip3 install pynvim                                            python module for neovim
>> which nvim

:checkhealth                                                      go inside neovim, run command to check health
## Python 3 provider (optional)                                   if we can see this in checkhealth, it means neovim is ok
  - INFO: Searching for python3 in the environment.
  - INFO: Executable: /usr/bin/python3
  - INFO: Python3 version: 3.6.9
  - INFO: python3-neovim version: 0.4.1
  - OK: Latest python3-neovim is installed: 0.3.1

vimplug for vim
>> sudo apt install curl
>> curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
   https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim     install vim-plug manager
>> vim ~/.vimrc
please see later section for complete config, then install plugin by
:PlugInstall                                                      install vim-plug plugin

vimplug for neovim
>> mkdir ~/.config/nvim
>> curl -fLo ~/.local/share/nvim/site/autoload/plug.vim --create-dirs \
   https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim     install vim-plug manager
>> nvim ~/.config/nvim/init.vim
please see later section for complete config, then install plugin by
:PlugInstall                                                      install vim-plug plugin
```

Symbolic link to latest version C++ and python

Do not overwrite binary, update symbolic link instead

```
>> ls / -Ral 2>/dev/null | grep g++                               check the location of compiler, or equivalently
>> find / -name g++* 2>/dev/null
>> cd /usr/bin                                                    symbolic links are all here
>> sudo ln -sf g++-10 g++
>> sudo ln -sf gcc-10 gcc
>> g++ --version                                                  verify version
10.1.0

>> sudo apt install python3.8                                     need to specify version number, otherwise it is old version
>> ls / -Ral 2>/dev/null | grep python                            check the location of interpreter, or equivalently
>> find / -name python* 2>/dev/null
>> cd /usr/bin                                                    symbolic links are all here
>> sudo ln -sf python3.8 python                                   point symbolic link python  to python3.8
>> sudo ln -sf python3.8 python3                                  point symbolic link python3 to python3.8
>> python --version                                               verify version
3.8.0
```

CMake

Method 1 : using `apt`, however this approach may not get the latest version

```
>> sudo apt install cmake
>> cmake --version
3.10.2
```

Method 2 : download binary `cmake-3.18.2-Linux-x86_64.tar.gz` from `https://cmake.org/download`

```
>> cd ~/Downloads
>> ls
cmake-3.18.2-Linux-x86_64.tar.gz                                its here
>> tar -xvzf cmake-3.18.2-Linux-x86_64.tar.gz                   unzip it
>> ls cmake-3.18.2-Linux-x86_64/bin
cmake
>> ./cmake --version                                            ./ to invoke this binary instead of the old one
3.18.2
>> find / -name cmake 2>/dev/null                               search for existing symbolic link
>> cd /usr/bin                                                  symbolic link is here
>> sudo ln -sf ~/Downloads/cmake-3.18.2-Linux-x86_64/bin/cmake cmake   symbolic link updated
>> cmake –version
3.18.2
```

Valgrind

Method 1 : using `apt`, however this approach may not get the latest version

```
>> sudo apt install valgrind
>> valgrind --version
3.13.0
```

Method 2 : download binary `valgrind-3.16.1.tar.bz2` from `http://www.linuxfromscratch.org/blfs/view/svn/general/valgrind.html`

```
>> cd ~/Downloads
>> ls
valgrind-3.16.1.tar.bz2                                         its here
>> tar -xvjf valgrind-3.16.1.tar.bz2                            unzip it
>> sed -i 's|/doc/valgrind||' docs/Makefile.in                  These 4 steps are copied from above site.
>> ./configure --prefix=/usr --datadir=/usr/share/doc/valgrind-3.16.1
>> make
>> sudo make install
>> valgrind –version
3.16.1
```

Google Test library (libgtest)

```
>> sudo apt install libgtest-dev
>> cd /usr/src/gtest
>> mkdir build                                                  sudo if no permission to modify /usr/src
>> cd build
>> cmake ..                                                     sudo if no permission to modify /usr/src
>> make -j4                                                     sudo if no permission to modify /usr/src
>> sudo cp *.a /usr/lib
>> ls /usr/lib | grep gtest
libgtest.a
libgtest_main.a
```

Gitlab and ssh key

```
>> sudo apt install ssh
>> sudo apt install git

Prepare for ssh and git
>> mkdir .ssh
>> cd .ssh
>> ssh-keygen -t rsa -b 4096 -C "Testing SSH key."
>> touch config
>> chmod 644 config
>> vim config

Host my_gitlab_server
    HostName 10.250.22.65
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/id_rsa

Copy public key to gitlab, then clone
>> cd ~
>> git clone ssh://my_gitlab_server/dick/YLibrary.git
```

Build C++ project and run

```
Set library path
>> find / -name libpthread* 2>/dev/null
>> find / -name librt* 2>/dev/null
>> echo PATH=$PATH:/usr/lib...

Build project and run
>> cd YLibrary
>> cd Oms
>> mkdir build
>> cd build
>> cmake ..
>> make -j4
>> ./Test arg0 arg1
```

Boost library

Download latest boost library

```
>> cd ~
>> wget https://dl.bintray.com/boostorg/release/1.74.0/source/boost_1_74_0.tar.bz2
>> tar -xvjf boost_1_74_0.tar.bz2
>> cd boost_1_74_0
>> ./bootstrap.sh --prefix=/usr --with-python=python3            (this is NOT build process)
>> ./b2 stage -j4 threading=multi link=shared                    (this is the build process, where -j4 means build using 4 threads)
>> sudo ./b2 install threading=multi link=shared                 (this step install header to /usr/include/boost, need root access)
>> find ./ -name *boost*.so*
./stage/lib/libboost_atomic.so
./stage/lib/libboost_chrono.so
./stage/lib/libboost_coroutine.so
./stage/lib/libboost_date_time.so
./stage/lib/libboost_filesystem.so
./stage/lib/libboost_iostreams.so
./stage/lib/libboost_regex.so
./stage/lib/libboost_system.so
./stage/lib/libboost_stacktrace_backtrace.so
./stage/lib/libboost_thread.so
./stage/lib/libboost_timer.so ... and many others ...
>> ll /usr/include/boost
in C++ project, include path /usr/include
in C++ source,  include <boost/module.hpp>
```

Some useful lockfree queue library in web

Here are 3 libraries, reader writer queue, atomic queue, and Facebook Folly library.

```
>> cd ~/dev
>> git clone https://github.com/cameron314/readerwriterqueue.git
>> git clone https://github.com/max0x7ba/atomic_queue.git
>> git clone https://github.com/facebook/folly.git
```

Share drive

Open file explore, type (please google Network attached storage, Network drive, SMB) :

```
smb://10.250.22.86/devshare
```

Saving the terminal

After upgrading to `Python3.8`, my Ubuntu fails to launch `gnome` terminal again. Here is the diagnotics and solution :

- `Alt F1` launch command line, type `terminal`, it fails to launch `gnome` terminal without any error
- `Alt F1` launch command line, type `log` to view system log, I can see some Python-related errors on launching terminal
- when `gnome` terminal is started, it runs the script `/usr/bin/gnome-terminal`, the first line specifies that the interpreter is :

  ```
  #!/usr/bin/python3.8
  ```

The solution is to change the interpreter back to `python3.6`. How can we modify that script without `gnome` terminal? The answer is to download another lightweight terminal, `termit` is a good choice. `Alt F1` to launch command line, type `store`, search for `termit`, install and open it, then :

```
>> sudo nvim /usr/bin/gnome-terminal
```

Part 1. SSH and puTTY

Typical TCP protocols, default ports and corresponding Apps :

- HTTP port 80
- FTP port 21
- SSH port 22 `puTTY`
- telnet protocol port 23 `telnet`
- remote desktop protocol RDP port 3389 remote desktop

What is puTTY?

- `puTTY` is an open source terminal emulator for remote access that supports multi-protocols, including `ssh`.
- Installation of `ssh` and installation of `puTTY` are two separate steps.

What is public key authentication?

- public key can be generated from private key, but not the other way round
- public key is for encryption by the public for message to a target
- private key is for decryption by the target for message from public

What is SSH?

- SSH stands for secured shell using public key cryptography. It is a TCP protocol.
- As compared to telnet, telnet is clear text including password, while `ssh` is secured, as it encrypts all text.
- As compared to RDP, RDP supports GUI (windows-experience), `ssh` is command line (bash-liked).

What is SSH key for *bitbucket* and *Gitlab*?

- in client machine, generate public and private key using command `ssh-keygen`
- in server machine, register the client-public-key as authorised key using *bitbucket* (in JPMorgan) or *Gitlab* (in Yubo), ***OR***
- in server machine, register the client-public-key as authorised key by concaternation to file `/root/.ssh/authorized_keys`
- we can see that there are many client-public-keys inside `/root/.ssh/authorized_keys`
- server can then send encrypt message (by public key) that only that client can decrypt (by private key)
- public key in server is also called authorised key
  private key in client is also called `ssh` key
- public key in server can be hashed to a shorter finger-print for easy identification
  private key in client can be protected by further encryption using passphrase

Possible hand shaking mechanism :

- client makes a connection to server
- server sends its public key (for example `server_pub`) without encryption to client
- client sends its public key (for example `client_pub`) encrypted by `server_pub` to server
- server decrypts `client_pub` using its private key (for example `server_pri`) and looks for `client_pub` in `/root/.ssh/authorized_keys`
- server may be serving multiple clients, however it can identify which client it is talking to from `/root/.ssh/authorized_keys`
- now both client and server has public key of the counterparty, they can therefore communicate safely by :
- server to client messages are encrypted by `client_pub` in server side
- client to server messages are encrypted by `server_pub` in client side

Making ssh connection between local machine and dev machine in production site

There are two machines :

- local machine `10.250.6.80`
- dev machine in production site `10.250.2.12`

In this section we will see how to make `ssh` connection from local machine to dev machine, and from local machine to Gitlab server. By setting config files inside folder `~/.ssh` appropriately, `ssh` connection can be made very easy. First of all, this is how we connect to dev machine from local machine using traditional `ssh` command :

```
>> ssh root@10.250.2.12
password = Welcome@Dev!@#$
```

We can preset `ssh` key in both local machine and production-site dev-machine, so that there is no need to enter password whenever we make `ssh` connection to `root@10.250.2.12`. There is a hidden `.ssh` folder under user home directory in local machine, it contains all `ssh` related files.

```
In local machine 10.250.6.80
>> ~/.ssh
>> ll
authorized_keys
config
gitlab_server.pub  //  public-key for connection to gitlab_server
gitlab_server      // private-key for connection to gitlab_server
```

The public-key and private-key for connection to Gitlab server exist already. Now I am going to create extra key pair for connection to dev-machine. There are two config files `config` and `authorized_keys`. The former records filepaths of all private-keys for making `ssh` connection to various servers from this local machine, whereas the latter records content of public-keys for accepting `ssh` connection from various clients to local machine.

We need the following steps to setup `ssh` key :

- generate `ssh` public-private-key in local machine, saved under `~/.ssh`
- copy private-key-path to local machine config `~/.ssh/config`
- copy public-key to dev-machine location `/root/.ssh`
- `cat` public-key to dev-machine location `/root/.ssh/authorized_keys`

As a result, we want to have :

- machine `10.250.6.80` stores `ssh` private-keys for connection to various servers in `~/.ssh/config`
- machine `10.250.2.12` stores `ssh` public-key of its possible clients in `/root/.ssh/authorized_keys`

Let's try once :

```
In local machine 10.250.6.80
>> cd ~/.ssh
>> ssh-keygen -b 4096 -t rsa -C "Dick (dev-machine)" // -C capital denotes comment in public-key, appear in server authorized_keys
Enter file prefix : dev_machine                      // -b is key size
>> ll
authorized_keys    // public -key value from various clients
config             // private-key path to various servers
dev_machine.pub    // public -key for ssh to dev-machine
dev_machine        // private-key for ssh to dev-machine
gitlab_server.pub  // public -key for ssh to gitlab_server
gitlab_server      // private-key for ssh to gitlab_server
>> nvim config
```

```
Host yubo_gitlab_server                    // alias to replace ygit.yubo.local
    HostName ygit.yubo.local
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/gitlab_server      // private-key for connection to gitlab-server from local host
    ServerAliveInterval 15                 // these 2 lines avoid auto closing ssh connection
    ServerAliveCountMax 1                  // these 2 lines avoid auto closing ssh connection

Host dev                                   // alias to replace 10.250.2.12
    HostName 10.250.2.12
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/dev_machine        // private-key for connection to dev-machine from local host
    ServerAliveInterval 15
    ServerAliveCountMax 1
```

Then transfer the public key to dev machine using `scp` :

```
>> scp dev_machine.pub root@10.250.2.12:/root/.ssh
Enter password : Welcome@Dev!@#$
>> ssh root@10.250.2.12
Enter password : Welcome@Dev!@#$

In dev-machine 10.250.2.12
>> cd /root/.ssh
>> cat dev_machine.pub >> authorized_keys            // > redirect by overwrite, >> redirect by append (there are existing keys)
>> cat authorized_keys
ssh-rsa j+vdsbJnmAeCkQKU/yyJFfBNlh+Wn9emVd3Ub0bCX5eKfbvbwICh2xKBpufC8Y2ZsLkhkvmlNmO58fYglCm0DAw== Dave (dev-machine)
ssh-rsa FtiKDEfgOfA0bJnmweWEGWYIEU/yyJFfBNlh+Wn9emVWfYglCm0h2xETHE2ZsLkhkvq0Hq8DAEG0bCX5eKfbvIC== Paul (dev-machine)
ssh-rsa l1hDqIrdXAVFtiKDEfgOfA07E0sPQx8q0HqlfU7gZ6fN437SiSmW9sQrnnTXp3MrenKc5wsjMNX3cSLfSzHIQQw== Jack (dev-machine)
real key is much longer ...
>> rm dev_machine.pub
```

From now on, we can `ssh` from local-machine to dev-machine by :

```
>> ssh root@dev
```

instead of prompting for password everytime :

```
>> ssh root@10.250.2.12
Enter password : Welcome@Dev!@#$
```

Making ssh connection between local machine and Gitlab server?

How about connecting Git from local machine? Whenever we invoke :

```
>> git remote -v
>> git fetch --all
>> git push
>> git pull dev master
```

Git makes `ssh` connection to Yubo Gitlab server. In order to avoid entering password everytime we `git push`, Git will look for `ssh` key inside directory :

```
    ~/.ssh./config
```

However, there are more than one `ssh` keys in `~/.ssh./config`, how does Git know which one to use? It is specified in Git config :

```
>> cat ~/.gitconfig

[user]
    name  = Dick Chow
    email = dick.chow@yubosecurities.com
[url "git@yubo_gitlab_server"]
    InsteadOf = git@10.250.29.65
[core]
        editor = nvim
[merge]
        tool = meld
```

Therefore Git will :

- read `yubo_gitlab_server` from `~/.gitconfig` then ...
- read `~/.ssh/gitlab_server` from `~/.ssh./config`
- make `ssh` connection to `ygit.yubo.local` without explicit password

Can we do similar thing for connection between home machine and local machine?

Yes of course, by following the same procedure, create and transfer `ssh` key from home machine to local machine, store it inside :

```
>> cd ~/.ssh
>> cat home_gen_key.pub >> authorized_keys
>> cat authorized_keys
display the ssh-public-key generated in home machine, for connection to office local machine
```

Part 2. Shell command

Linux offers two ways to access kernel :

- command via bash / korn shell
- programmatic way using system call

There are two meanings for root :

- root directory `/`
- root user

There are two meanings for filesystem :

- how files are physically managed, such as FAT32 in windows, EXT4 in linux
- how files are arranged, such as

`/bin` and `/usr/bin`	executables or symbolic link to executable (such as `gcc`, `gdb`, `cmake`, `nvim`)
`/boot`	stuffs for booting linux
`/dev`	stuffs for system devices
`/etc`	configuration files
`/home`	folder of users (this is the only folder in which users can modify without root-user privilege)
`/lib` and `/usr/lib`	libraries such as `.a` and `.so` (such as `libpthread`)
`/opt`	optional softwares
`/tmp`	temporary files that can be removed on reboot
`/var`	temporary files for backup, logging, app icons

There are two types of shells :

- login shell — opens new session on login, it housekeeps a individual set of processes,
- non login shell — reuses existing session without login, it shares the same set of processes of parent shell

They source different config files in sequence :

- login shell — started by `login` and `ssh`, it sources `~/.bash_profile` → `~/.profile`
- non login shell — started by `bash`, it sources ...
- in windows 10's bash — started by `bash`, it sources `~/.profile` → `~/.bashrc`

`~/.profile`	config for user (under `home`)
`~/.bash_rc`	config for `bash` (under `home`) such as prompt, for example : add these 2 lines for *smart-cmd-history*
`~/.vimrc`	config for `vim` (under `home`)
`~/.ssh/config`	config for `ssh` (inside a hidden folder)
`~/.gitconfig`	config for `git` (under `home`)

↓

```
bind '"\e[A": history-search-backward'
bind '"\e[B": history-search-forward'
```

Here are commands for login, query info, adding user, deleting user etc.

`>> login`	invokes login shell (create a new set of processes)	
`>> bash`	invokes non login shell (reuse same set of processes)	
`>> echo $$`	print current process ID	
`>> exit`	quit current session (going back to original session before calling `login`, `ssh` or `bash`	
`>> id`	print username	
`>> whoami`	print username	
`>> hostname`	print hostname, such as `DESKTOP-DQUQDHA` (my machine at home)	
`>> hostname -I`	print hostname as address, such as `192.168.8.7` (my machine at home)	
`>> su`	switch to root, require its password	*require root user password*
`>> su <username>`	switch to another user, require its password	*require another user password*
`>> sudo command`	run `command` once on behave of root-privilege	*require current user password*
`>> sudo -u user command`	run `command` once on behave of `user` privilege	*require current user password*
`>> sudo -i -u user`	login as `user` in interactive mode	*require current user password*
`>> sudo adduser <username>`	add new user	
`>> sudo deluser <username>`	delete existing user	
`>> sudo passwd <username>`	modify existing user password	
`>> sudo usermod -l <un1> <un0>`	modify existing user name `un0` to `un1`	
`>> sudo usermod -g <gn> <un>`	assign existing user `un` to group `gn`	
`>> sudo groupmod -n <gn1> <un0>`	modify existing group `gn0` to `gn1`	
`>> cat /etc/passwd`	list all existing users, there are a lot of abnormal users, normal users have `uid` greater than `1000` with output format `username:x:uid:gid:fullname:homedir:default_login_shell`	

Linux command format is `>> command -option0 -option1 arg0 arg1`

- order among options and arguments can be switched
- add single quotation around argument if it is a string containing space

Linux command catergories :

- navigate filesystem — `ls, cd, cp, mv, rm, rmdir, mkdir, pwd, find, man, clear, chmod`
  `pushd, popd, dirs, dirname, basename` (please check `~/dev/bash/*.sh`)
- file access — `head, tail -f, less, more`
- file merge / split — `cat, join, split, diff, cmp` — `cat, tail, less` are 3 important tools for reading log
- file row sorting — `sort, uniq, wc` — `find` *searches pattern in filename*
- file pattern searching — `grep, cut, awk, sed` — `grep` *searches pattern in file content*
- cascade commands — pipe `|` and redirection `>`

symbol

`.`	current directory	
`..`	parent directory	
`~`	home directory	
`/`	root directory	in C++, means comment, in `vim`, means search
`\`	escape character	in C++, means escape character too
`.ABC`	hidden file `.ABC`	
`. ABC.sh`	source shell script `ABC.sh`	
`$`	variable	
`#`	comment	in python, means comment too
`@`	suppress echo	
`'this is a str'`	enclose argument with `''` if it contains space	

navigate filesystem

`ls -alR`	list hidden file `-a`, in long format `-l` and recursively `-R`	
`lsof`	list all opened files, including socket, pipes etc	
`cd /`	moving to root directory	
`cd ~`	moving to home directory `/home/n733607`	
`cd -`	moving to previous location	
`cp file new_file`	copy file to the same directory with new filename	i.e. `file2file`
`cp file new_dir`	copy file to new directory with the same filename	i.e. `file2dir`
`cp dir new_dir -R`	copy directory to new directory recursively	i.e. `dir2dir`
`mv file new_file`	move file to the same directory with new filename	i.e. `file2file`
`mv file new_dir`	move file to new directory with the same filename	i.e. `file2dir`
`mv dir new_dir`	move directory to new directory (no `-R` needed)	i.e. `dir2dir`
`rm file`	remove file	
`rm -r dir`	remove everything inside directory recursively (including the directory)	
`rm -r dir/*`	remove everything inside directory recursively (excluding the directory)	
`rm -rf dir`	remove everything with forced (i.e. protected files are removed toom, useful in `git`)	
`rmdir dir`	remove directory only when it is empty (hence this can be done by `rm -r dir`)	
`mkdir dir`	make new directory	
`mkdir -p dir/x/y/z`	make new directory (all required directories in the middle)	
`pwd`	print working directory	
`ln -s /mnt/d/dev dev`	create symbolic link `dev` pointing to folder `/mnt/d/dev`, `-s` means soft link (what?)	
`ln -sf /mnt/d/dev2 dev`	modify existing symbolic link `dev` pointing to new folder `/mnt/d/dev2`	
`find dir0 dir1 dir2`	list all files recursively under `dir0`, `dir1` and `dir2`	
`find dir0 dir1 -name '*str*'`	list all files recursively under `dir0`, `dir1` so that filename contains `str`, don't forget `''`	
`find dir -name '*str*' 2>/dev/null`	lot of permission denied error `2` is generated, so forward them to null space `/dev/null`	
`find dir -name '*str*' 2>/dev/null	grep 'pattern'`	find all files having filename `*str*` and search their content for `'pattern'`	
`chmod -R 744 /home/dick/f0/f1/f2`	change mode recursively, everyone can read all files under `f2`	
	change mode for chain of parents of `f2` has to be done manually, or by script	

file access

`head -30 file`	display first 30 rows of file (result is shown after prompt)
`tail -30 file`	display last 30 rows of file (result is shown after prompt)
`tail -f  file`	display with follow, i.e. append data as file grows
`less file`	display a file one page at a time, press space-bar to continue and press q to quit
	difference between `head/tail` and `more/less` is that the latter shows result in cleared screen
`hexdump -C file`	display file content in heximal and ASCII

file merge and split

`cat file1`	concatenate (vertically) and forward to standard output
`cat file1 file2 > file3`	concatenate (vertically) and redirected to `file3`
`join file1 file2`	join (horizontally) with column 1 in `file1` and column 1 in `file2` as index
`join -1 n -2 m file1 file2`	join (horizontally) with column `n` in `file1` and column `m` in `file2` as index
`split -b 3KB file prefix`	split file into size 3KB, save as files with specified prefix and auto-gen alphabet suffix
`split -b 5GB file prefix -d`	split file into size 5GB, save as files with specified prefix and auto-gen numeric suffix
`split -l 100 file prefix -d`	split file into 100 lines
`diff file1 file2`	display extra line and miss line on comparison
`cmp  file1 file2`	display location of first difference on comparison

Join operation means considering both files as space-delimited files, then :

- *each row is a vector / an entry*
- *each column is an attribute*
- *join operation picks one column from* `file1` *and one column from* `file2` *as index*
- *join the row from* `file1` *and the row from* `file2` *sharing the same index*

file row sorting

The following commands consider each row as an entry, i.e. they cannot sort the columns.

`echo -e '10\n9\n8' | sort`	sort rows alphabetically, output `10-8-9` (`\n` creates rows, as `sort` cannot sort col)
`echo -e '10\n9\n8' | sort -n`	sort rows numerically `-n`, output `8-9-10` (`\n` creates rows, as `sort` cannot sort col)
`sort -k2 file`	sort all rows in file according to column 2 (`-k2`)
`sort -m file1 file2 file3`	merge sorted files without sorting, this is the last step of merge sort
`sort -n file`	sort numerically, instead of alphabetically
`sort -r file`	sort reversely (descending order)
`sort -t= -k3 file`	sort with new delimiter `=` according to the 3rd column
`uniq file`	remove adjacent duplicated rows
`uniq file -i -c`	remove adjacent duplicated rows, case insensitive `-i`, output count `-c`
`wc file`	output line count / word count / character count
`sort < file1 > file2`	read `file1`, sort it, write result to `file2` using redirection
`cat file1 | sort > file2`	read `file1`, sort it, write result to `file2` using pipes
`cat file file | sort | uniq`	removal occurs

file pattern searching

- `grep` is a selection of rows. `cut` is a selection of columns.
- `grep` is a row iterator : for each row, perform a pattern search, outputs matched rows
- `awk` is also a row iterator : for each row, perform a condition test, invoke action on rows that fulfill the condition
- `awk` condition is programmable, defaulted to be all-true
- `awk` action is programmable, defaulted to be printing

`grep pattern file`	search file content for pattern, output matched rows
`grep pattern file -i`	search file content for pattern, output matched rows (case insensitive)
`grep pattern dir -R`	search file content for pattern, for all files inside directory recursively
`grep pattern . -R --include="*.cpp"`	search file content for pattern, for all `.cpp` files inside current directory recursively
`grep -E 'queue|order' ~/*.txt`	search file content for `queue` or `order` (`-E` for regex, `|` for OR, `.*` for AND)
`grep -E 'queue.*09:30:00' ~/*.txt`	search file content for `queue` message at `09:30:00`
`grep -E 'queue.*09:30:00|order.*09:30:00' ~/*.txt`	

Comparison between `grep` and `find` in searching filename (not file content) :

`ls /usr/bin | grep pattern`	search file with name containing `pattern`
`find /usr/bin -name *pattern*`	search file with name matching `*pattern*`, wild card is necessary for `find`

`cut -b10 file`	select byte `#10` of all rows, no delimiter allowed
`cut -c10 file`	select character `#10` of all rows, no delimiter allowed
`cut -f2,5-7 file`	select field (column) `#2,5-7` of all rows, delimited by `\t` (column index starts with `#1`)
`cut -f2,5-7 -d= file`	select field (column) `#2,5-7` of all rows, delimited by `=`
`cut -f5- file --output-delimiter=*`	select field (column) `#5` and above of all rows, output using new delimiter `*`
`cut -f5- file --output-delimiter=$'\t'`	select field (column) `#5` and above of all rows, output using new delimiter `\t`

General form of `awk` is like a lambda defined inside `'...'`

```
awk ‘condition { action0; action1; }’ file
awk ‘BEGIN { init0; init1; init2; } condition { action0; action1 } END { reduce0; reduce1; reduce2 }’ file
```

which means :

- perform initialization in `BEGIN` block (optional)
- for each row in `file`, if `condition(row[n])` is true, then perform `action0(row[n])` and `action1(row[n])`
- perform reduction in `END` block (optional)
- statements may contain pre-defined variable
  - `$0` for whole row (sometimes we need to invoke `$1=$1` to force `awk` to re-evaluate `$0`)
  - `$1` for 1st column
  - `$2` for 2nd column
  - `$n` where `n` can be a user-defined variable
  - `FILENAME` for current filename
  - `FS` for current field separator
  - `OFS` for current output field separator (start from 1)
  - `NR` for current row number (start from 1)
- statements may contain user-defined variable, without explicit declaration
- all variables can be manipulated like C++, can be considered as integer or string depending on context

`awk ‘/abc/’ file`	if a row contains `"abc"` print the row	(search pattern `/pattern/`)
`awk ‘$1~/abc/’ file`	if a row's 1st field contains `"abc"`, print the row	(search pattern `~/pattern/`)
`awk ‘$1~/^abc/’ file`	if a row's 1st field starts with `"abc"`, print the row	(`^` means start-with)
`awk ‘$3>100 {print $1}’ file`	if a row's 3rd field is greater than 100, print the 1st field	
`awk ‘$3>100 {print $3,$2 “---” $1}’ file`	if a row's 3rd field is greater than 100, print `$3+$2---$1`, where default delimiter is `+`	
`awk ‘$3>100 && length($1)<10 {n++}’ file`	if a row fulfills the condition, increment `n`, we should print `n` in `END` block	
`awk ‘$3>100 {total+=$3; n++}’ file`	find average by accumulating `total` and `n`, we should print `n` in `END` block	

Example 1

Read 1000 rows from a csv file, using space delimited, print all trade messages before 10am, sorted by field number 6.

```
head -1000 ~/20161011.txt | awk ‘BEGIN {FS=”,”; OFS=” ”} $2<“10:00:00” && $3==“trade” {$1=$1; print $0}’ | sort -k6
```

Example 2

Read all rows from a csv file, accumulate the 3rd field for all trade messages, print the sum.

```
cat ~/folder/20161011.txt | awk ‘BEGIN {FS=”=”;} /trade/ {total+=$3} END {print “total=” total;}’
```

Example 3

Read all rows from a csv file, convert delimitor to tab. Using two approaches : `cut` and `awk`.

```
cut file -f1- -d ‘,’ --output-delimiter=$’\t’
awk ‘BEGIN {FS=”,”; OFS=”\t”} {$1=$1; print $0}’ file
```

environment variables and library symbols

`var=1234`	declare variable `var`	
`echo $var`	print variable `var`, `$` denotes variable	
`echo $PS1`	command shell prompt	
`echo $PATH`	path to executable	usually include `/usr/local/bin:/usr/bin:/bin`
`echo $LD_LIBRARY_PATH`	path to shared library	usually include `/usr/local/lib:/usr/lib:/lib`
`echo 'This is a test' > fn`	create file `fn` with content	
`touch fn`	create file `fn` without content	
`env`	list all environment variable	
`export`	update environment variable	
• *example 1*	`export PS1="\e[0;33m[\u@\h \W]\$ \e[m "` (where `\e[0;33m` and `\e[m` is about setting color)	
• *example 2*	`export PS1='logname'@'hostname -s':'$PWD >'`	
• *example 3*	`export LD_LIBRARY_PATH=/usr/lib64:$HOME/runtime/lib:$LD_LIBRARY_PATH`	
`source ~/.profile`	invoke profile shell script, which is a hidden file `.profile` under home directory `~/`	
`source script.sh`	invoke any shell script (in some systems, command `.` is the same as command `source`)	
`objdump --syms executable`	list symbol table of `executable`, there is a long list, its better to `grep` something you are interested in	
`ldd executable`	list library dependency of `executable`	

redirection and pipes

The difference between redirect `>` and pipe `|` is that :

- `<` redirect file to program-input — `cmd < input.txt`
- `>` redirect program-output to file (overwrite) — `cmd > output.txt`
- `>>` redirect program-output to file (append) — `cmd >> output_accumulate.txt`
- `1>file` redirect `std::cout` to file
- `2>file` redirect `std::err` to file
- `&>file` redirect both `std::cout` and `std::err` to file — thus `&` on LHS of `>` means `cout` and `err`
- `2>&1` redirect `std::err` to `std::cout`, where `&` in `&1` means not-a-file — thus `&` on RHS of `>` means not-a-file
- `|` forwards program-output to another program-input — `cmd0 | cmd1 | cmd2`
- `;` or `&&` sequential program invocation without input-output relation — `cmd0 ; cmd1 ; cmd2` or `cmd0 && cmd1 && cmd2`

The following two commands are equivalent :

```
>> prog < temp.txt              we never have : temp.txt > prog
>> cat temp.txt | prog
```

The following two commands are also equivalent :

```
>> prog0 > temp.txt && prog1 < temp.txt
>> prog0 | prog1
```

Both unnamed pipe (`|`) and named pipe (`mkfifo`) are considered as file in linux.

```
>> ls -l | grep str             This is unnamed pipe.
>> mkfifo pipename              This is named pipe.
>> ls -al
drwxr--r--. foldername
-rwxr--r--. filename
prwxr--r--. pipename            This is named pipe.
```

process vs job

Processes form an inheritance hierarchy :

- when a process invokes another process, the former is parent process, the latter is child process
- when a process is invoked in shell, the `bash` shell is parent process
- every process has a parent process, `init` process is parent of all processes
- orphan process is one having parent process killed, `init` process then becomes its parent
- dead process yet still remains in process table, is called zombie process
- daemons are background processes that offer service to other processes

Job is a group of processes (most of case, it is single process). Processes are grouped together for ease of management.

- a job must be executed by a shell
- a job normally contains one process, however it can be a group of processes when ...
- if it is a pipeline in shell, like `./test arg0 arg1 | grep pattern > output` (one process is `test`, another is `grep`) or
- if it invokes system call `fork()`, which makes identical copies of address spaces (one for parent process, one for child process)
- when we apply state-transition on a job, all the processes are updated together

```
#include<unistd.h> // fork and pid
int main()
{
    // program usually forks at the beginning ...
    pid_t child_pid = fork();
    if (child_pid != 0)
    {
        pid_t this_pid = getpid();
        std::cout << "\n pid = " <<  pid; // a unique pid
        std::cout << "\ncpid = " << cpid; // parent process gets a non zero child_pid, suppose it is 12345
        run_parent();
    }
    else
    {
        pid_t this_pid = getpid();
        std::cout << "\n pid = " <<  pid; // must be equivalent to child_pid in parent process, i.e. 12345
        std::cout << "\ncpid = " << cpid; // child process gets a zero child_pid
        run_child();
    }
}
```

Job states

- we can run a job in foreground by `./test arg` (occupies shell input and output) or
- we can run job in background by `./test arg &` (detached from shell input but occupies shell output)
- possible job states include : foreground job / background job / stopped job (can be restarted) / terminated job
- possible job state transitions include :

`./test arg`	initiating foreground job
`./test arg &`	initiating background job
`ctrl-z`	foreground job → stopped (once a job is stopped, it returns the shell to user)
`ctrl-c`	foreground job → terminated
`fg jid`	running in background → running in foreground, where `jid` is job id
`fg jid`	stopped → running in foreground, the job occupies the shell again (hence `fg jid` can do 2 things)
`bg jid`	stopped → running in background, the job does not occupy the shell

- we cannot bring foreground job directly to background without stopping it
- we cannot terminate background job without bring it to foreground, unless we `kill -9 pid` for each process in the job

Command `ps` and `jobs`

- each process has a `pid` while each job has a `jid` plus one or more `pid`
- `pid` usually consists of 4-5 digits, while `jid` looks like `[1]`, `[2]`, `[3] ...`
- command `ps` shows informations of all processes of different shells in the machine
- command `jobs` shows informations of all jobs of current shell in the machine

`glances`	list running proces with beautiful textUI, need to install	
`top / htop`	list running process (keep updating, press `q` to quit), `htop` has a better text-UI	
`top -H -p pid`	list running threads associated to process `pid`, option `-H` shows individual threads of target process	
`ps`	list running process (not informative, it needs to work with options)	I prefer to use `ps aux`
`jobs`	list existing job	I prefer to use `jobs -l`
`kill pid`	kill process `pid`	
`kill -9 pid`	kill process `pid` by sending signal, where signal 9 is `SIGKILL`	
`sleep 10`	sleep this thread for 10 seconds, shell is blocked	
`nice +3 exe arg`	run process `exe` with argument `arg` and nice value `+3` (nice value ranges from `-20` to `+19`, `-20` is the highest)	
`nice -3 exe arg`	run process `exe` with argument `arg` and nice value `-3`, `sudo` is needed for negative nice value	
`renice -n 9 -p pid`	change nice value of an existing process `pid`	

There are two formats for running `ps` command : `BSD` format and `UNIX` format.

`ps au`	`BSD` format does not have a `dash`
`ps au | grep pn`	`BSD` format that lists cpu usage of the process name `pn`
`ps a`	listing all sessions, not just current session
`ps u`	listing usual items `user, pid, %cpu, %mem, vsz, rss, stat, start, time, command`
`ps -ef`	`UNIX` format does have a `dash`
`ps -e`	listing all processes, including many irrelevant processes (better not to use this option)
`ps -f`	listing default items, excluding `vsz` and `rss`
`ps -o item1 ...`	listing customized items, such as :

`ps -o pid,ppid,cmd,%cpu,%mem,vsz,rss`	
`ps -o pid,ppid,cmd,%cpu,%mem,vsz,rss --sort=+%mem | head -n 10`	sorted in ascending order of `%mem` and list top 10 only
`ps -o pid,ppid,cmd,%cpu,%mem,vsz,rss --sort=-%mem | grep pattern`	sorted in descending order of `%mem` and list pattern only

where

```
pid  = process id
ppid = parent process id
%cpu = relative cpu usage
%mem = relative mem usage
vsz  = absolute virtual memory usage (size in KBytes)
rss  = absolute physical memory usage (size in KBytes)
```

Example

Run program `test` that keeps writing to `std::cout` forever until it reads an `ENTER` from `std::cin`. If we run `test` in background, we need to redirect its output to a file, otherwise its output will occupy the shell and we cannot use any job command (including `fg` or `ctrl-c`) to communicate with it. It will then run forever.

```
>> ./test arg > output0 &
>> ./test arg > output1 &
>> ./test arg > output2 &
>> ./test arg > output3 &
>> jobs                                                    list all jobs status (pid is not shown)
>> jobs -l                                                 list all jobs status with pid
[1]  8301  Running ./test arg0 > output0
[2]  28394 Running ./test arg1 > output1
[3]- 28014 Running ./test arg2 > output2                   - means previous job
[4]+ 28331 Running ./test arg3 > output3                   + means current job
>> echo $1                                                 print pid of latest background job
```

Besides, if program `test` runs to the point at which it waits for input from `std::cin`, as `test` is detached from `std::cin`, `test` will be put to stopped status aftering running for a while. In this case, we may need to bring it back to foreground.

```
// After a while ...
>> jobs -l
[1]  8301  Stopped (tty input) ./test arg0 > output0        (tty input) means that it is waiting for terminal input
[2]  28394 Stopped (tty input) ./test arg1 > output1
[3]- 28014 Stopped (tty input) ./test arg2 > output2
[4]+ 28331 Stopped (tty input) ./test arg3 > output3
>> fg 1                                                    bring jid 1 (not pid) to foreground
```

This problem happens in `TraderRun` too, which reads `std::cin` as a control. However we want to run `TraderRun` in background.

Solution 1

- if we consider `test` to be a server program, then do not read `std::cin` as a control, read socket or pipe instead
- if we consider `test` to be a local executable, then it must have a text-user-interface, do not run it in background

Solution 2

Lets try to construct a named pipe (first in first out) using command `mkfifo`.

```
>> mkfifo p0
>> mkfifo p1
>> mkfifo p2
>> cat p0 | ./test arg0 > output0 &
>> cat p1 | ./test arg1 > output1 &
>> cat p2 | ./test arg2 > output2 &
>> jobs -l
[1] 4000 Running    cat p0
    4001            | ./Test 7 > output0 &
[2]-4002 Running    cat p1
    4003            | ./Test 7 > output1 &
[3]+4004 Running    cat p2
    4005            | ./Test 7 > output2 &
>> ps au                                                   you can see all 6 processes running
>> cat output0                                             keep growing, i.e. running
>> cat output1                                             keep growing, i.e. running
>> cat output2                                             keep growing, i.e. running
>> echo x > p0                                             enter anything ('x' in this case) to pipe to stop test
>> echo y > p1                                             enter anything ('y' in this case) to pipe to stop test
>> echo z > p2                                             enter anything ('z' in this case) to pipe to stop test
>> jobs -l                                                 all gone
```

cpu info

`cat /proc/meminfo`	list memory information
`cat /proc/cpuinfo`	list cpu information
`lscpu | grep "CPU MHz"`	list cpu information (frequency spec and real frequency)
`cpufreq-info | grep "CPU"`	list cpu frequency
`sudo cpufreq-set -d 4.2Ghz`	set cpu min (down) frequency (with this setting, `clock_gettime` resolution becomes 15ns)
`sudo cpufreq-set -u 4.2Ghz`	set cpu max (up) frequency
`cset`	set cpu affinity
`uptime`	time since last system reboot

system info

`uname -a`	list linux version
`software --version`	list software version
`which software`	list software location or its symbolic link
`getconf -a`	list system config (`getconf LEVEL1_DCACHE_LINESIZE` for cacheline size)
`ulimit -a`	list system limit

```
core file size          (blocks, -c) 0           useful for generating gdb coredump   >> ulimit -c unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63712
max locked memory       (kbytes, -l) 16384       useful for mlock(), avoid page fault >> ulimit -l unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queue     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192        useful for allocating hugh stack mem >> ulimit -s unlimited
max user processes              (-u) 63712
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

`df`	list filesystem usage [disk full forbids `vim` to save changes]
`mpstat -A`	list processor status : usage and interrupt (`-A` to list all CPUs)
`prtstat pid`	list process `pid` status : page fault, CPU time
`vmstat`	list virtual memory : page swap-in `si`, page swap-out `so`
`netstat -tupnl`	list network status : `protocol`, `local addr`, `remote addr`, `pid` (`t`TCP, `u`UDP, `p`port, `n`num addr, `l`listening)
`netstat -rn`	list network status : `r`routing table for each interface
`sudo tcpdump -v -A -s0`	sniff targetted packets : `-v verbose`, `-A ascii`, `-s0` snapshot unlimited printed
`-i lo port 12345`	sniff targetted packets : `-i` interface `lo` and specific port number
	note that `tcpdump` does not work for windows subsystem for linux

other tools

`cmd --help`	show manual, with output like `cat`
`man cmd`	show manual, with output like `less` (enter `q` to quit)
`watch cmd`	run command `cmd` once every two seconds (default), for example `watch ls` to check files
`watch cmd -n 1`	run command `cmd` once every one second

In linux, archiving files into single file and compression of file are two separate processes. `tar` makes archive and generate `.tar` file, while `gzip` compress file into `.gz` file, alternatively we can also use `bzip2` to compress file into `.bz2` file. Download is done by `curl`

`Curl “https://url” -o file`	download from specified `url` and save as `file`
`file.tar.gz`	an archived and `gzip` compressed portfolio of files
`file.tar.bz2`	an archived and `bzip2` compressed portfolio of files
`gzip file`	compress a file
`gzip -d file.gz`	decompress a file
`tar -cvzf file.tar.gz folder`	compress `folder` into `file.tar.gz`
`tar -xvzf file.tar.gz`	`x` extract from `tar`, `v` verboses files extracted, `z` decompress by `gzip`, `f` read input as file
`tar -xvjf file.tar.bz2`	`x` extract from `tar`, `v` verboses files extracted, `j` decompress by `bzip2`, `f` read input as file

network tools

`ifconfig`	list IP address
`ping ygit.yubo.local`	ping machine `ygit.yubo.local` using ICMP via any interface
`ping ygit.yubo.local -I itf0`	ping machine `ygit.yubo.local` using ICMP via interface (network card) `itf0`
`dig  ygit.yubo.local`	resolve domain name `ygit.yubo.local` into IP using DNS server
`nslookup ygit.yubo.local`	resolve domain name `ygit.yubo.local` into IP using DNS server

How to set DNS in *Ubuntu* :

```
network ► wired (click the gear icon) ► IPV4
set DNS = 10.250.22.51 (Yubo DNS server), 8.8.8.8 (google DNS server), 8.8.4.4
```

If DNS server fails frequently, we can hardcode the ***domain name*** to ***IP*** mapping in a local config file `/etc/hosts`. Thus when domain name is needed, local machine will go to file `/etc/hosts` first, if domain name is not found, it will then request the DNS server.

```
>> sudo nvim /etc/hosts

// hard code git server as ...
10.250.29.65 ygit.yubo.local
10.250.22.51 dns.yubo.local
```

network tools – nc

Set a machine (`192.168.8.9`) as chat server (listening to a custom port) :

```
>> nc -l -vv -p 5000
```

Connect to that machine on that port :

```
>> nc      192.168.8.9 5000 or
>> teklnet 192.168.8.9 5000
```

then we can type any thing on both sides, the 2 machines can talk.

network tool – ssh

How to run ssh server in WSL (`192.168.8.9`)? How to ssh into WSL from ubuntu notebook (`192.168.8.12`)? Firstly uninstall and install openssh in WSL, set custom port (such as `2222`) as the ssh server port, in order to avoid collision with ssh server port in windows.

```
>> sudo apt remove openssh-server
>> sudo apt update
>> sudo apt install openssh-server
>> sudo nvim /etc/ssh/sshd_config
.. modify : PasswordAuthentication to yes
.. append : AllowUsers ktchow1
```

Run ssh daemon as service in WSL :

```
>> service ssh status
>> sudo service ssh start
>> sudo service ssh --full-restart
>> service ssh status
```

Allow port `2222` in firewall : `firewall setting >> advanced setting >> inbound rules >> new rules` :

- TCP port `2222`
- UDP port `2222`

Finally goto client :

```
>> ssh ktchow1@192.168.8.9 -p 2222
```

Part 3. VIM and NeoVIM Editor

Font style bug

Font style bug in windows subsystem for linux *WSL*

- need to reset font style to Consolas (codepage 437) whenever using `vim` `nvim` or `less`
- solution : open `regedit`
  goto `computer\HKEY_USERS\S-1-5-21-3974791996-2029213106-1576070703-1001\Console`
  add `DWORD` with key `CodePage` and value `437 (decimal)`

Various mode of operations

Seven modes of operations

• normal mode	`ESC`	cursor is positioned on a character
• insertion mode	`i`	cursor is positioned between two characters
• visual mode	`v`	
• visual line/block mode	`shift-v ctrl-v`	
• command mode	`:`	
• bash command mode	`:!`	
• terminal mode (vim 8.0)		

Jump between different modes

• normal mode to insert mode	`i a A o` all `c-command`	back to normal mode by one `ESC`
• normal mode to visual (block) mode	`v ctrl-v shift-v`	back to normal mode by two `ESC ESC`
• normal mode to command (bash) mode	`: :!`	back to normal mode by two `ESC ESC`

Normal mode

Navigation

`10hjkl`	navigation (alternatives are : `BACKSPACE ENTER - SPACE`)	
`w`	jump to the start of next word, delimited by non-alphabets	(capital `W` delimted by space)
`b`	jump to the start of previous / current word, delimited by non-alphabets	(capital `B` delimted by space)
`t<char>`	jump to the character before `char` in the same line	
`f<char>`	jump to the character right on `char` in the same line	
`0`	jump to the first `char` of this line (which may be a space)	
`0w  j0w`	jump to the first word of this line / next line	
`$`	jump to the last `char` of a line	
`$b  k$b`	jump to the last word of a this line / prev line	
`:300`	jump to the start of line 300	
`{`	move cursor up by one block defined by empty-line	
`}`	move cursor down by one block defined by empty-line	
`[[`	move cursor to the prev starting bracket (as the first character of a line)	
`]]`	move cursor to the next starting bracket (as the first character of a line)	
`[]`	move cursor to the prev closing bracket (as the first character of a line)	
`][`	move cursor to the next closing bracket (as the first character of a line)	
`%`	move cursor to matched bracket / square bracket / brace	
`ctrl-u`	move cursor up by half page without scrolling	
`ctrl-d`	move cursor down by half page without scrolling	
`ctrl-o`	move cursor to original position	
`ctrl-a / ctrl x`	move cursor to next integer and increment / decrement by one	
`ctrl-z`	put `vim` to background	
`ctrl-e`	without moving pointer, scroll page up by 1 line	
`ctrl-y`	without moving pointer, scroll page down by 1 line	
`zz zt zb`	without moving pointer, scroll to bring cursor pointed line to the centre / top / bottom	
`gg`	go home	
`gd`	go to declaration of variable on which cursor lies (search in same file only)	
`gD`	go to definition of variable on which cursor lies (search in same file only)	
`G`	go to end	

empty-line-block is parag separated by empty-lines
bracketed-block is parag separated by various brackets

Navigation by moving cursor

gg	scroll without moving cursor
ctrl-u	ctrl-e
[[	
{	
k	
0 T/F b h - l w t/f $	zz
j	
}	
]]	
ctrl-d	ctrl-y
/pattern	
G	

Fundamental difference between vim and notepad
- cursor (char vs line)
- cursor cannot move beyond last char in a line

Modification

I	start insertion mode before current line	*and switch to insert mode*
i	start insertion mode before cursor	*and switch to insert mode*
a	start insertion mode after cursor	*and switch to insert mode*
A	start insertion mode after current line	*and switch to insert mode*
o	(small o) new line below current line	*and switch to insert mode*
O	(capital O) new line in current line	*and switch to insert mode*
p	(small p) paste yanked-content below current line	
P	(capital P) paste yanked-content in current line (likes windows)	
~	swap case for one character	
r	replace one character	
x	delete one character	
dd yy	delete / copy current line	
cw dw yw vw	change / delete / copy / select current word starting from cursor position	
ciw diw yiw viw	change / delete / copy / select whole current word regardless of cursor position	
ci{ di{ yi{ vi{	change / delete / copy / select content within current bracket { excluded	
ca{ da{ ya{ va{	change / delete / copy / select content within current bracket { included	
ctx dtx ytx vtx	change / delete / copy / select the rest of current line, start from cursor to char x exclusively	
cfx dfx yfx vfx	change / delete / copy / select the rest of current line, start from cursor to char x inclusively	
C D	change / delete the rest of current line, start from cursor to the end	
>>	indentation (at the start) for current line, even when cursor is in the middle of line	
.	repeat the last action, ignoring all navigation commands (for example cw includes everything up to ESC)	
f0 r1 f0 r1	on the same time line, keep finding 0 and replace it with 1	
u	undo once	
ctrl-r	redo once	
50i= ESC	insert 50 characters =, hence it draws a double line, it is effective only when you ESC	

OP I-----ia-----A

op

i is consistent with std::vector::insert

Find and replace

*	find all instances of the word where cursor currently lies	*method 1*
/str	search for str in current file, all matched strings are highlighted	*method 2*
yiw, / ctrl-r 0	yank a word (to register 0), search it by / ctrl-r 0	*method 3*
yiw, ciw, ctrl-r 0	yank a word (to register 0), move to another word, replace it with yanked word	
n	goto next match	*for method 1,2*
N	goto previous match	*for method 1,2*
:noh	no highlighted text	*for method 1,2,3*
:s/str0/str1/g	substitute s str0 by str1 for all matches (globally g) in current line	*method 4a*
:%s/str0/str1/g	substitute s str0 by str1 for all matches (globally g) in current file (%)	*method 4b*
:%s/str0/str1/gc	substitute s str0 by str1 for all matches (globally g) in in current file, prompt for confirmation for each match	
:10,20s/str0/str1/g	substitute s str0 by str1 for all matches (globally g) in between line 10 to 20	*method 4c*
:g/str/cmd	global command which finds all lines with str and apply cmd on each of them, where d is a common cmd	
:g/str/d	global command which finds all lines with str and delete	
:g!/str/d	global command which finds all lines without str and delete	
:g/^$/d	global command which deletes all empty lines	
^$	in regex, ^$ denotes empty lines (^ and $ denotes start and end of a line respectively)	

Note
- g means global command or global option
- pattern str for both :s and :g can be regex
- / is a separator, can be any character as long as all 3 separators are consistent

Command mode

command-line-mode `:` and bash-command-line `:!`

`:q!` `:qa`	quit without asking / quit all files
`:wq!`	write and quit without asking
`:w` `:wa`	write without quit / write all files
`:123`	goto line number `123`
`:tabnew` (`t` in nerdtree)	create new tab
`:new` (`s` in nerdtree)	create new windows in the same tab, horizontally (i.e. upper and lower)
`:vnew` (`v` in nerdtree)	create new windows in the same tab, vertically (i.e. LHS and RHS)
`:vnew` and `:term`	open a terminal inside `vim`, go back to other windows by : `ESC` + `ctrl\` + `ctrlN` + `ctrlW` + arrow
`ctrl-pageup` or `gt`	switch between tabs
`ctrl-w w`	switch between windows
`ctrl-z`	put current process (i.e. `vim`) to background and return control to shell
`>> fg` (pin shell)	put background process (i.e. `vim`) to foreground and return back to `vim`
`:source %`	source current file, in particular if this file is a `bash` script or `.vimrc`
`:so %`	source current file, in particular if this file is a `bash` script or `.vimrc`
`:sh`	to switch to `bash`, type `exit` to back to `vim`
`:lcd`	list current directory
`:ls`	list all `vim` buffers
`:db 4`	delete `vim` buffer number `4`
`:!ls`	list all folders and files
`:!bash_cmd`	run `bash` command without switching to `bash`
`:!g++ %`	run `g++` on current file, `%` means current file
`:!grep str -r .`	run `grep` for pattern `str` on current location recursively (compare two searches `:!grep str` with `\str`)

- For commands that are built-in `vim`, they can be invoked by `:cmd`. *(please find out the list of commands)*
- For commands that are not built-in `vim`, they can be invoked by `:!cmd`.
- In both cases, command outputs are displayed in shell (for `vim`), displayed inside `nvim` (for `nvim`).

Builtin make and quickfix

`nvim` comes with a builtin `make`, builtin `make`, it builds files with current directory as the standing point, so start `nvim` from project root.

`:make`	does not work if makefile is not in current directory
`:build/debug/make`	start building debug
`:build/release/make`	start building release, however this cumbersome, we should set `makeprg` first, by typing ...
`:set makeprg=` `cd\ build/debug\;make`	which set builtin `make` as a combo of `cd` plus builtin `make`, where `\` means special char behind

The main difference between `vim` builtin `make` and external `make`, is that the former can port its output to `quickfix`, so we can ...

`:copen 40`	open `quickfix` to view all build errors, there are shortcuts from errors to corresponding files
`:cclose`	close `quickfix`
`:cp :cc :cn`	jump to previous / current / next error

Visual mode

`v`	jump to visual mode (normal select likes windows)
`vw viw vi{ va{ vtx vfx`	jump to visual mode, various selection methods ...
`ctrl-v`	jump to visual block mode (select char matrix)
`shift-v`	jump to visual line mode (select complete lines)
`shift-v G`	select all (starting from current line to the end)
`v + hjkl + x`	highlighed block, yank and delete
`v + hjkl + d`	highlighed block, yank and delete
`v + hjkl + y`	highlighed block, yank
`0wh v0d i backspace`	delete evey character before the first word, append current line to the end of previous line
`G shift-v { d gg 5j p`	goto the bottom, copy and del a truck of code, paste it near to the front

Visual block mode

`ctrl-v c zzz ESC`	change selected block to `zzz`, effective after pressing `ESC`
`ctrl-v d`	delete selected block
`ctrl-v >`	insert indent before selected block
`ctrl-v :'<,'> norm I xxx`	insert `xxx` before selected block (`'<,'>` is selected block, `norm` means "as-if" normal mode, `I` is insert)

Recording macros

`q<char>`	start recording action sequence to `char`
`q`	stop recording to `char`
`@<char>`	replay recorded action sequence to `char`
recording example	*add quotes around each row*

`qw~h i'ESC$ i',ESC j0wq`	played by `@w @w @w`
`qw i{ESC$ i: ESC j0wi'ESC $i'},ESC j0w`	played by `@w @w @w`

Split screen (not for differing)

1. split screen in bash can be done by , after invoking `tmux`, we can apply `tmux` commands, which all start with `ctrl-b`

- there are two layers in `tmux`, multi-windows in one `tmux` and multi-panes in one window (we usually use multi-pane)
- create a new window by `ctrl-b c`
- create a horizontol split on current window by `ctrl-b %` (i.e. LHS pane and RHS pane)
- create a vertical split on current window by `ctrl-b "` (i.e. up pane and down pane)
- switch to desired window by `ctrl-b w 0/1/2`
- switch to desired pane by `ctrl-b arrow`
- switch to next pane by `ctrl-b o`
- close current window by `ctrl-b x`

Update the config of `tmux`

```
>> nvim ~/.tmux.conf

# from v2.1
set -gq mouse on
# before v2.1
set -gq mode-mouse on
set -gq mouse-resize-pane on
set -gq mouse-select-pane on
set -gq mouse-select-window on
```

2. split screen in `vim` can be done by command `:split` or `:vsplit`

```
>> nvim fileA
```

`[nvim] :split fileB`	split screen horizontally, `fileB` at the top while `fileA` at the bottom
`[nvim] :vsplit fileB`	split screen vertically, `fileB` on the LHS while `fileA` on the RHS
`[nvim] :new`	split screen horizontally with new empty file
`[nvim] :vnew`	split screen vertically with new empty file

If you want to open new file on RHS instead of LHS, please enter the following command before `vsplit` :

`[nvim] :set splitright` `[nvim] :vsplit fileB`	after that, all vertical split opens up on the RHS (work for both `:vsplit` and `:vnew`)
`[nvim] ctrl-w w`	switch to next window
`[nvim] ctrl-w W`	switch to prev window
`[nvim] ctrl-w h/j/k/l`	switch to target window
`[nvim] ctrl-w H/J/K/L`	move current window left / down / up / right

3. Plugins like *Nerdtree* and *CtrlP* also provide facilities for splitting screens or tabs inside `vim`. Not for bash of course.

- in *Nerdtree* select desired file and press `i`, selected file is opened up in horizontal split
- in *Nerdtree* select desired file and press `s`, selected file is opened up in vertical split
- in *CtrlP* perform fuzzy search and press `ctrl-x` to open up horizontal split
- in *CtrlP* perform fuzzy search and press `ctrl-v` to open up vertical split

4. Open new tab (not new split) and close existing tab by

`[nvim] :tabnew`	open new tab
`[nvim] :tabc`	close existing tab
`[nvim] :gt`	goto next tab
`[nvim] :gT`	goto previous tab

File comparison

1. comparison in bash : using command `diff` which shows unmatched lines in `git` like format in the shell

```
>> diff file0 file1
>> diff -r folder0 folder1
```

compare two files
compare two folders recursively

```
< line_100 in file0
< line_101 in file0
< line_102 in file0
---
> line_90 in file1
> line_91 in file1
```

for example, line 100-102 in file0 is matched with line 90-91 in file1

2. comparison in `vim` : using one of these two equivalent commands in bash
- `vim` will then be invoked ... type `ctrl-w w` to switch between windows, type `zr` to unfold folded sections

```
>> vimdiff file0 file1
>> vim -d file0 file1
```

3. comparison in existing `vim`
- if an active file is opened in `vim`, if we want to compare it with another file in `pathA/fileB`

```
:vert diffsplit pathA/fileB
```

`dp` diff push / diff set / diff put
`do` diff pull / diff get / diff obtain
`zo` zip open
`zc` zip close

4. install `vim` plug-in by adding this line `Plug 'will133/vim-dirdiff'` in `~/.vimrc`

`[nvim] :PlugInstall` run this command in `vim` to install
`[nvim] :DirDiff folder0 folder1` run this command in `vim` to compare two folders

5. install software like `meld` or `kompare` which offer graphical interface

Remark

Compare the differences between

```
:vnew
:vsplit filepath
:vert diffsplit filepath
```

Fast switching between coding and compiling

1. open two terminals by `ctrl-shift-t`, one for `vim` and one for `make` — switch windows by `alt-tilde`
2. open one terminal, open two tabs inside the terminal by `ctrl-alt-t` — switch windows by `ctrl-pageup` or `gt`
3. open one terminal, split windows inside the terminal by `tmux` — switch windows by `ctrl-b w`
4. instantiate one `nvim`, split windows inside `nvim` by `:split` or `:vsplit` (one for `:make`) — switch windows by `ctrl-w w`
   instantiate one `nvim`, split windows inside `nvim` by `:vnew`, then run `:term` in one of them — useful for `gdb` inside `nvim`
5. instantiate one `nvim` in build folder, compile by `:make`, read error by quickfix `:copen`, run program by `:!./my_exe`
6. instantiate one `nvim`, put `nvim` to background by `ctrl-z`, compile by `make`, bring `nvim` back to foreground by `fg` as follows :

```
[nvim] ctrl-z
>> make -j4
compile error in line 123
>> fg
[nvim] :123
[nvim] zz
```

Vim NeoVim config

Config can be found in the following path (if they do not exist, please create one) :

- `vim` config can be found in `~/.vimrc`
- `nvim` config can be found in `~/.config/nvim/init.vim`

Here are some of the options (or run in command mode, such as `:set number`) :

`set number`	enable line number
`set nonumber`	disable line number
`set relativenumber`	enable relative line number
`set incsearch`	enable incremental search
`set hlsearch`	enable highlight of matched pattern
`set smartcase`	enable smart case (when pattern has no capital letter, it is ignore-case, otherise it is case-sensitive)
`set clipboard=unnamedplus`	connect clipboard of `vim` and other prog (i.e. `y` in `vim` then `ctrl-p` in prog, or `ctrl-v` in prog then `p` in `vim`)
`g:xyz`	means global scope variable `xyz`
`:w <CR> :so % <CR>`	save and source current `.vimrc`, making it effective without quiting `vim`

We can also do key-mapping. The following commands show various mapping :

`:map`	recursive mapping that works in all vim-modes
`:noremap`	non-recursive mapping that works in all vim-modes
`:nmap`	recursive mapping that works in normal-mode only
`:nnoremap`	non-recursive mapping that works in normal-mode only
`:vmap`	recursive mapping that works in visual-mode only
`:vnoremap`	non-recursive mapping that works in visual-mode only

The following statements customize various mapping in config file :

`nmap <C-n> :NERDTreeToggle<CR>`	setting `ctrl-n` to turn on/off NerdTree
`nnoremap <C-c> :!g++ -o  %:r.out % -std=c++11<CR>`	setting `ctrl-c` to run `g++`
`nnoremap <C-x> :!./%:r.out`	setting `ctrl-x` to run built executable

where `:!` runs bash
where `%` is current filename
where `%:r` is current filename without extension
where `<CR>` is carriage return or enter
where `<C-x>` is `ctrl` together with key `x`, hence `<abc>` means simultaneous key-in of `abc`, `abc` means sequential key-in of `abc`.

In Vim or NeoVim config, we can define function :

```
function! my_fct0()
    ...
endfunction
```

where `function!` means if it is already defined, then overwrite

```
function my_fct1()
    ...
endfunction
```

where `function` means if it is already defined, then error

Define function can be called in `vim` by command mode `:call my_function0`

Vim and NeoVim plugin

Vim and NeoVim plugins can be installed using managers, common managers include :

- vim plug
- vim bundle, also called vundle
- pathogen

There are 3 important paths for `vim` and `neovim` respectively :

- path of config (fixed)
- path of plugin manager (fixed, probably due to `curl`)
- path of plugin package (customizable, but better use the path stated in stackoverflow)

```
 vim config file  : ~/.vimrc
 vim-plug manager : ~/.vim/autoload/plug.vim
 vim-plug folder  : ~/.vim/plugged
nvim config file  : ~/.config/nvim/init.vim
nvim-plug manager : ~/.local/share/nvim/site/autoload/plug.vim
nvim-plug folder  : ~/.local/share/nvim/site/plugged
```

Firstly, install package managers using `curl`, run the following command in shell :

```
>> curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
   https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```
for installing vim plug

```
>> curl -fLo ~/.local/share/nvim/site/autoload/plug.vim --create-dirs \
   https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```
for installing neovim plug

Secondly, edit config file, add a section specifying path to plugin packages :

- for `vim`, add a section in config `~/.vimrc`, insert plugin-of-interest in between :

```
call plug#begin('~/.vim/plugged')        # where arg is path for storing plugin packages
Plug('name_of_plugin0')
Plug('name_of_plugin1')
...
Plug('name_of_pluginN')
call plug#end()
```

- for `neovim`, add a section in config `~/.config/nvim/init.vim`, insert plugin-of-interest in between :

```
call plug#begin('~/.local/share/nvim/site/plugged')        # where arg is path for storing plugin packages
Plug('name_of_plugin0')
Plug('name_of_plugin1')
...
Plug('name_of_pluginN')
call plug#end()
```

Run command with `vim` to install plugin (where `%` means current file) :

```
:source %
:PlugInstall
```

Run command with `vim` to uninstall plugin :

```
:source %
:PlugClean
```

Sample config file for `neovim`

```
set number
set relativenumber
set background=dark
set nocindent
set noautoindent
set nosmartindent
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab

call plug#begin('~/.local/share/nvim/site/plugged')
Plug 'scrooloose/nerdtree', { 'on': 'NERDTreeToggle' }
Plug 'scrooloose/syntastic'
call plug#end()
map <C-n> :NERDTreeToggle<CR> # In vim config, <C-n> means ctrl-n, <CR> means ENTER.
```

How to get latest version Neovim?

As the Neovim shipped with Ubuntu is old. We need to add the following `nvim` repository `ppa:neovim-ppa/stable` into `apt` :

```
>> sudo add-apt-repository ppa:neovim-ppa/stable
>> sudo apt-get update
>> sudo apt-get install neovim
>> nvim --version
v0.4.4
```

navigation in folder : Nerdtree / CtrlP
compilation : cmake-make-command, Copen
debug and run : cmake-terminal
other : deoplete, TT comment

Plugin 1 - NerdTree

How to use NerdTree?

`ctrl-n`	to toggle nerdtree on/off
`ctrl-w w`	to switch between different splits
`gt`	to switch between different tabs
`i` in the tree	open file with a horizontal split into up and down
`s` in the tree	open file with a vertical split into left and right
`t` in the tree	open file in new tab
`R` in the tree	to refresh files in directory tree

Plugin 2 - CtrlP

CtrlP is for fuzzy file search. It searches for filename (not file content).

`ctrl-p`	to turn on CtrlP, type keyword to search, scroll up and down with arrows and enter, or ...
`ctrl-d`	to switch between search-by-filename mode and search-by-filepath mode
`ctrl-v`	to open file in vertical split
`ctrl-x`	to open file in horizontal split
`gt` in normal mode	to switch between tabs
`ctrl-a` or `ctrl-e`	equivalent to moving to front / end of the search pattern (use them only after entering `ctrl-p`)

Plugin 3 - NeoVim quickfix `copen`

Quickfix is shipped with `nvim`, there is no need to install. It offers solution to quick compile-and-debug iteration.

`:copen`	open quickfix inside `nvim`, if we have just run `:make`, then it displays compile error and links to source code
`:copen 40`	open quickfix inside `nvim`, with 40 rows
`:vert copen`	open quickfix inside `nvim` vertically
`:cclose`	close quickfix

With quickfix, compile-and-debug iteration becomes :

`:setlocal makeprg=scripts/build.sh`	`:setlocal my_build_command` where `my_build_command` can be `make` or any scripts
`:make debug`	`:make arg` is equivalent to invoking `my_build_command arg`
`:copen`	display the terminal output from `:make arg` command

then navigate in quickfix windows, go to the line of interest and enter, it jumps to corresponding source code, debug and repeat.

Plugin 4 - CTags

Install CTags first. CTags seems to be supported by original `nvim`.

```
>> sudo apt install exuberant-ctags
>> which ctags
>> which etags
>> cd ~/dev/YLibrary
>> ctags -R .          generate a tag file
>> cat tags
>> nvim ~/.ctags

--recurse=yes
--exclude=.git
--exclude=vendor
```

Plugin 5 - deoplete

Deoplete is a better version of you-complete-me. It needs the following prerequisites :

- latest version `nvim` (please refer to previous page)
- *Message pack* in python, but first we need to install python `pip`

```
>> sudo apt install python-pip
>> pip install -U msgpack
```

After that we can install *deoplete* through *vim-plug*. Now add the following lines in `.config/nvim/init.vim` :

```
if has('nvim')
    Plug 'Shougo/deoplete.nvim', { 'do': ':UpdateRemotePlugins' }
else
    Plug 'Shougo/deoplete.nvim'
    Plug 'roxma/nvim-yarp'
    Plug 'roxma/vim-hug-neovim-rpc'
endif
:let g:deoplete#enable_at_startup = 1
```

then run `:source %` and `:PlugInstall`.

Plugin 6 - Vim airline

This is a status bar at the bottom of the active window. Installed by adding this line in `.config/nvim/init.vim` :

```
Plug 'bling/vim-airline'
```

then run `:source %` and `:PlugInstall`.

Plugin 7 - TomTom tcomment

This is for fast comment and uncomment.

- cursor lands on a line, enter `gcc`
- select a block of lines, enter `gc`

Plugin 8 - Fugitive

Here are the shortcut :

- open `git`, enter `:Git` (with colon, capital `G`)
- diff file, enter `dd` (no colon)
- diff push, enter `dp` (no colon)
- diff obtain (pull), enter `do` (no colon)
- zip open and zip close
- commit, enter `cc` (no colon)

Part 4A. Compiler gcc and g++

Very often, the gcc command can be made very long with a lot options and filenames, in order to avoid repeatedly typing the same long string and changing options every time, we have build-system that facilitates the job. Common build-systems include :

- make program which reads makefile as config file and invokes gcc
- ninja program which reads build.ninja as config file and invokes gcc
- msvs program in Visual Studio which reads vcproj as config file and invokes cl.exe

However, even writing makefile itself is a tedious job, cmake program is therefore introduced as a generic tools :

- cmake program which reads CMakeLists.txt as config file and generates makefile, build.ninja or vcproj for various build-systems.

C++ compiler

gcc is both the compiler and the linker, we can either do it step by step or everything in single step :

- preprocessing from .cpp to .i
- compilation from .i to .s
- assemble from .s to .o
- linking from .o to .exe or .a or .so

► .o is object file (corresponds to .obj in windows)
► .a is static library (corresponds to .lib in windows)
► .so is shared library (corresponds to .dll in windows)

- library in linux are prefixed with lib and ended with .a or .so, such as libxyz.a or libpthread.so
- we don't need to write the whole library name in config, we just need to write -lxyz and -lpthread instead

```
>> sudo apt install gcc-8 g++-8      install newer version
>> g++ --version                     list compiler version
g++ 7.3.0                            in my case, it still points to old version, why?

>> find / -name 'g++*' 2>/dev/null
/usr/bin/g++-7
/usr/bin/g++-8
/usr/bin/g++-10
>> cd /usr/bin
>> ls -l | grep g++                  we can tell which are executable and which are symbolic link by their size
lrwxrwxrwx 1 root   root   ------5 g++    -> g++-7
lrwxrwxrwx 1 root   root   -----22 g++-7  -> x86_64-linux-gnu-g++-7
lrwxrwxrwx 1 root   root   -----22 g++-8  -> x86_64-linux-gnu-g++-8
lrwxrwxrwx 1 root   root   -----23 g++-10 -> x86_64-linux-gnu-g++-10
lrwxrwxrwx 1 root   root   ------5 x86_64-linux-gnu-g++ -> g++-7
-rwxr-xr-x 1 root   root   1047488 x86_64-linux-gnu-g++-7
-rwxr-xr-x 1 root   root   1063896 x86_64-linux-gnu-g++-8
-rwxr-xr-x 1 root   root   1215440 x86_64-linux-gnu-g++-10
>> ln -sf g++-8 g++
>> ls -l | grep g++
lrwxrwxrwx 1 root   root   ------5 g++    -> g++-8
lrwxrwxrwx 1 root   root   -----22 g++-7  -> x86_64-linux-gnu-g++-7
lrwxrwxrwx 1 root   root   -----22 g++-8  -> x86_64-linux-gnu-g++-8
lrwxrwxrwx 1 root   root   -----23 g++-10 -> x86_64-linux-gnu-g++-10
...
>> g++ --version
g++ (Ubuntu 8.4.0-1ubuntu1~18.04) 8.4.0
```

GNU C library

Very often, when we use multi-threading or interprocess communication, such as sockets, pipes or shared memory, we need to link with GNU C library, which is a bunch of headers, .a and .so libraries. Please read :

https://www.gnu.org/software/libc/manual/html_node/index.html

About unused variable error

- using attribute [[maybe_unused]] my_class var;
- using casting to void trick, which does nothing, but using the variable

```
(void)var;
```

Compile C++ program

Let's try project with one source code.

`>> g++ -std=c++17 test.cpp`	default output `a.out`
`>> g++ -std=c++17 -o test test.cpp`	desired output `test`

Now perform compilation and linking in single steps.

```
>> g++ -std=c++17 -o test test0.cpp test1.cpp test2.cpp -I. -I/home/n733607/proj1 -lpthread -lchrono
```

Now perform compilation and linking in two separate steps.

```
>> g++ -std=c++17 -c test0.cpp test1.cpp test2.cpp -I. -I/home/n733607/proj1
>> g++ -std=c++17 -o test test0.o test1.o test2.o -lpthread -lchrono
```

where the options are (please read `https://www.rapidtables.com/code/linux/gcc.html`) :

`-c`	for compilation only, no linking invoked	
`-g, -g0, -g1, -g3`	for generating debugger `gdb` information	(we usually use `-g` for debug mode)
`-O0, -O1, -O2, -O3`	for optimizating execution time and code size	(we usually use `-O2` for release mode)
`-o filename.out`	for specifying output filename	
`-Iheaderpath0 -Iheaderpath1 -I.`	for specifying include header path, `-I.` means including current path	
`-Llibpath0 -Llibpath1 -L.`	for specifying linking library path, `-L.` means linking current path	
`-llibname0 -llibname1`	for specifying linking library name	
`-w`	for disabling all warning messages	
`-Wall`	for enabling all warning messages	
`-Wextra`	for enabling extra warning messages	
`-Dmacro[=value]`	for defining macro with optional value, `[]` means optional	

If your program involves `std::thread` but forget to link `-lpthread`, it will throw exception `std::thread operation not permitted`.

`-g`	default debug information	`-O0`	optimize for compile time
`-g0`	no debug information	`-O1`	optimize for execution time
`-g1`	minimum debug information	`-O2`	optimize more for execution time
`-g3`	maximum debug information	`-O3`	optimize even more for execution time

Debug information and optimization are two separate flags, so we can turn on debug information even for `O3`. Debug/release mode is just a labelled combo of flags.

Makefile as a DAG

Both `makefile` and `build.ninja` not procedural program. Instead, they simply specify inter-dependency among multiple source codes and object files in cplusplus project in terms of rules. Each rule is defined by `target`, `prerequisites` and `command`, where each `target` and `prerequisites` is either a label or a filename (for existing file / to-be-created file). Therefore it's liked forming a DAG with :

- vertex = `target` or `prerequisites`
- edge = rule connecting `target` and `prerequisites`

Make program then starts tracing from the ultimate target (the first rule in `makefile`), to all leaves (rules with no dependency) using topological sort in the DAG, hence resolving the ***build sequence***. `ninja` is just a faster alternative to `make`. Make file is named `makefile` or `Makefile`, what we need to do, is to `cd` to that folder, then run `make` :

```
>> cd target_path
>> ls make*
makefile                                yes it is here
>> make clean                           clean existing executable or library, it does not make again automatically
>> make                                 make will look for makefile in current directory
>> make VERBOSE=1                       make will look for makefile in current directory, all make messages are shown
>> make -j4                             make with 4 threads to speed up the compilation process
```

Makefile for single project

Normally, each rule is defined by `target`, `prerequisites` and `command` with syntax :

```
target : prerequisites
    command
```

Using the following notations :

`#`	comment
`\`	new line
`@`	suppress echo (otherwise the command itself will be echoed in shell)
`$(var)`	dereference variable `var` (no need to declare variable, but dereference variable with `$`)
`$@`	`target` filename
`$*`	`target` filename without extension
`$<`	the first `prerequisites` filename only
`$^`	all `prerequisites` filenames
`$?`	all `prerequisites` filenames that are newer than `target` only
`%.xyz`	for each label or filename `wildcard.xyz`
`-c/-g/-o`	same as `g++`, stands for compilation only, with debug info and output filename
`-I/-L/-l`	same as `g++`, stands for include path, library path and linked library

Firstly let's try a naive makefile having only one rule. It has no `prerequisites`, it is the only rule that fired.

```
target : # no prerequisites, this is a leaf in DAG
    @g++ -std=c++17 -o test test0.cpp test1.cpp test2.cpp test3.cpp -I. -I/home/n733607/proj -lpthread -lchrono
```

Secondly let's try multiple rules. Most commands are prefix with `@` to suppress the echo. The base thing with this `makefile` is that the list of cpp files and the list of object files seem to be duplicated.

```
target : link_step
    @rm *.o # remove intermediate object files

link_step : compile_step
    @echo "link step"
    @g++ -std=c++17 -o test test0.o test1.o test2.o test3.o -lpthread -lchrono

compile_step : preparation
    @echo "compile step"
    @g++ -std=c++17 -c test0.cpp test1.cpp test2.cpp test3.cpp -I. -I/home/n733607/proj

preparation : # no prerequisites, this is a leaf in DAG
    @clear
    @g++ --version
```

Thirdly let's try to remove the duplicated list of cpp files and object files by storing the list in variable `OBJECT`.

```
CC       = g++ -std=c++17
FLAG     = # reserved
CFLAG    = # reserved
SOURCE   = # reserved
OBJECT   = test0.o test1.o test2.o test3.o
INCLUDE  = -I. -I/home/n733607/proj
LINKPATH = # reserved
LINKLIB  = -lpthread -lchrono

test : $(OBJECT)
     $(CC) -o $@ $^ $(LINKLIB)

%.o : %.cpp
     $(CC) -c -o $@ $< $(INLCUDE)
```

The first rule states that the ultimate target is `test`, which depends on `test0.o test1.o test2.o test3.o`. The second rule states that the object files are dependent on the corresponding cpp files. By expanding the second rule, we have :

```
test0.o : test0.cpp         $(CC) -c -o test0.o test0.o $(INLCUDE)        where % = test0, $@ = test0.o, $< = test0.cpp
test1.o : test1.cpp         $(CC) -c -o test1.o test1.o $(INLCUDE)        where % = test1, $@ = test1.o, $< = test1.cpp
test2.o : test2.cpp         $(CC) -c -o test2.o test2.o $(INLCUDE)        where % = test2, $@ = test2.o, $< = test2.cpp
test3.o : test3.cpp         $(CC) -c -o test3.o test3.o $(INLCUDE)        where % = test3, $@ = test3.o, $< = test3.cpp
```

However we still need to enter either :

- a list of object files which is then mapped to cpp files by `%.o : %.cpp`
- a list of cpp files which is then mapped to object files by `%.cpp : %.o`

Finally we replace the file list by `SOURCE` variable, so that there is no need to modify `makefile` whenever we add new source code.

```
CC       = g++ -std=c++17
FLAG     = # reserved                # Flags for linking step
CFLAG    = # reserved                # Flags for compile step
SOURCE   = $(wildcard *.cpp)         # SOURCE = set{ for all files fulfilling *.cpp format }
OBJECT   = $(SOURCE : %.cpp = %.o)   # OBJECT = set{ for each item in SOURCE, map %.cpp to %.o }
INCLUDE  = -I. -I/home/n733607/proj
LINKPATH = # reserved
LINKLIB  = -lpthread -lchrono

test : $(OBJECT)
     $(CC) -o $@ $^ $(LINKLIB)

%.o : %.cpp
     $(CC) -c -o $@ $< $(INLCUDE)
```

Makefile for multiple projects

Now lets extend `makefile` to support multiple projects and linking shared libraries. We need to know :

- how to compile as static library `-shared -o libquantlib.a` static library should prefix with `lib`
- how to compile as shared library `-shared -o libquantlib.so` shared library should prefix with `lib`
- how to link a static library `-lquantlib -L/home/n733607/cpp/...` static library full path is needed
- how to link a shared library `-lquantlib -L/home/n733607/cpp/...` shared library full path is needed

How can compiler distinguish whether a linked library is static or shared?

- Suppose we link `-lquantlib`, then ...
- compiler searches for `libquantlib.a` in path specified in `-L`, if found, then it is static library, otherwise ...
- compiler searches for `libquantlib.so` in path specified in `LD_LIBRARY_PATH`, if found, it's done, otherwise compile error.
- static library must be named as `libxxxx.a`
- shared library must be named as `libxxxx.so`
- linked library is done by `-lxxxx`, excluding `lib`, `.a` or `.so`

Suppose we have two project folders, namely `proj` which is an executable and `quantlib` which is a shared library :

```
/home/n733607/cpp/proj/makefile          /home/n733607/cpp/quantlib/makefile
              .../proj/header.h                      .../quantlib/quantlib.h
              .../proj/test0.cpp                     .../quantlib/quant0.cpp
              .../proj/test1.cpp                     .../quantlib/quant1.cpp
              .../proj/test2.cpp                     .../quantlib/quant2.cpp
              .../proj/test3.cpp
```

The `quantlib/makefile` should look like the following :

```
CC       = g++ -std=c++17                 # -shared specifies output to be static/shared library
FLAG     = -fPIC -shared -g -Wall         # -g enables debug info, -Wall enables warning
CFLAG    =                                # reserved
SOURCE   = $(wildcard *.cpp)
OBJECT   = $(SOURCE : %.cpp = %.o)
INCLUDE  = -I.
LINKPATH =                                # reserved
LINKLIB  =                                # reserved

libquantlib.a : $(SOURCE)                 # for static library, use libquantlib.a
     $(CC) $(FLAG) -o $@ $^ $(INCLUDE)    # for shared library, use libquantlib.so
```

The `proj/makefile` should look like the following, don't forget to add `quantlib` in include path and link path :

```
CC       = g++ -std=c++17                 # -shared specifies output to be static/shared library
FLAG     =
CFLAG    = -g -Wall
SOURCE   = $(wildcard *.cpp)
OBJECT   = $(SOURCE : %.cpp = %.o)
INCLUDE  = -I. \
           -I/home/n733607/cpp/quantlib
LINKPATH = -L/home/n733607/cpp/quantlib
LINKLIB  = -lquantlib \
           -lpthread  \
           -lchrono

test : $(OBJECT)
     $(CC) $(FLAG) -o $@ $^ $(LINKPATH) $(LINKLIB)
     rm *.o

%.o : %.cpp
     $(CC) $(CFLAG) -c -o $@ $< $(INCLUDE)
```

Debugging information can be turned on/off in `proj` and `quantlib` independent.

Part 4B. CMake

CMake is a program that helps to create make file. Therefore we have the following production line :

```
                    cmake           make
CMakeLists.txt -----> Makefile -----> invokes g++ or clang compiler to generate .o .a .so and executable
```

Some possible combinations :

- `cmake > make > g++`
- `cmake > ninja > clang`

Config file `CMakeLists.txt` is put in the following file structure. Very often, we make a `build` directory, go inside and run `cmake` :

```
>> ls
-rwxrwxrwx 1 ktchow1 main_group 782 Aug 24 17:49 CMakeLists.txt
drwxrwxrwx 1 ktchow1 main_group 512 Aug 24 17:45 include
drwxrwxrwx 1 ktchow1 main_group 512 Aug 24 17:46 src
>> mkdir build;       cd build
>> mkdir debug;       cd debug;      cmake -DCMAKE_BUILD_TYPE=Debug    ../..;    make -j4;   cd ..
>> mkdir release;     cd release;    cmake -DCMAKE_BUILD_TYPE=Release  ../..;    make -j4;   cd ..
>> debug/Test arg     // run program in debug mode
>> release/Test arg   // run program in release mode
```

For example, if we want to build an executable with all source codes in folder `src` and headers in folder `include`, besides, we need to link shared libraries `libstdc++.so`, `libpthread.so` and `librt.so`, how can we write the config? Firstly let's find out where they are :

```
>> find / -name *libstdc++.so* 2>/dev/null              // libstdc++.so contains all runtime standard c++
/lib64/libstdc++.so

>> find / -name *libthread* 2>/dev/null
/lib/x86_64-linux-gnu/libthread_db-1.0.so
/lib/x86_64-linux-gnu/libthread_db.so.1
/lib32/libthread_db-1.0.so
/lib32/libthread_db.so.1

>> find / -name *librt* 2>/dev/null
/lib/x86_64-linux-gnu/librt-2.27.so
/lib/x86_64-linux-gnu/librt.so.1
/lib32/librt-2.27.so
/lib32/librt.so.1
```

We define project name by `project()`. Use `add_executable()` to build an executable, Use `add_library()`to build a static / shared library.

```
cmake_minimum_required(VERSION 3.10.2)
project(Oms)

### (0) flags ###
set(CMAKE_C_COMPILER   /usr/bin/gcc-10)                # hard code the exact location of compiler in this config
set(CMAKE_CXX_COMPILER /usr/bin/g++-10)                # hard code the exact location of compiler in this config
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
add_definitions(-std=c++20)
add_definitions(-g)                                    # for debug mode, not for release mode
add_definitions(-O2)                                   # for release mode, not for debug mode

### (1) include ###
include_directories(include  /mnt/d/dev/boost_1_72_0)

### (2) source ###
file(GLOB SOURCES "src/*.cpp"
                  "test/*.cpp"
                  "test/helper/*.cpp")                 # add all cpp files

add_executable(Test ${SOURCES})                        # for building executable
# add_library(Test STATIC ${SOURCES})                  # for building static lib .a
# add_library(Test SHARED ${SOURCES})                  # for building shared lib .so

### (3) link library and link path ###
target_link_libraries(Test -static-libstdc++)          # statically link to libstdc++, Test will be portable to other OS, but huge
target_link_libraries(Test -ldl)                       # dynamically link to dl lib, for boost::stacktrace's dladdr()
target_link_libraries(Test -lrt)                       # dynamically link to rt lib, for shm_open()
target_link_libraries(Test -lpthread)                  #
target_link_libraries(Test -lgtest)                    # -l specifies library to be linked
target_link_libraries(Test -L/lib/x86_64-linux-gnu)    # -L specifies library path

### (4) installation for shared lib ###
# install(TARGETS Test DESTINATION /usr/lib)
```

How can we apply different compilation flags for debug and release mode? We can set those flags with *IF* statement :

```
message("CMAKE_BUILD_TYPE        = ${CMAKE_BUILD_TYPE}")
message("CMAKE_C_FLAGS           = ${CMAKE_C_FLAGS}")
message("CMAKE_C_FLAGS_DEBUG     = ${CMAKE_C_FLAGS_DEBUG}")
message("CMAKE_C_FLAGS_RELEASE   = ${CMAKE_C_FLAGS_RELEASE}")
message("CMAKE_CXX_FLAGS         = ${CMAKE_CXX_FLAGS}")
message("CMAKE_CXX_FLAGS_DEBUG   = ${CMAKE_CXX_FLAGS_DEBUG}")
message("CMAKE_CXX_FLAGS_RELEASE = ${CMAKE_CXX_FLAGS_RELEASE}")

# Set flags conditionally ... here we use -g option for both debug and release
if ("${CMAKE_BUILD_TYPE}" STREQUAL "Debug")
    set(CMAKE_C_FLAGS   "-g -O0 -DLOGGING_TAG")                 # LOGGING_LAG is a directive for YLib only
    set(CMAKE_CXX_FLAGS "-g -O0 -DLOGGING_TAG")
elseif ("${CMAKE_BUILD_TYPE}" STREQUAL "Release")
    set(CMAKE_C_FLAGS   "-g -O3 -DNDEBUG -DLOGGING_TAG")        # NDEBUG is a directive for YLib only
    set(CMAKE_CXX_FLAGS "-g -O3 -DNDEBUG -DLOGGING_TAG")
elseif ("${CMAKE_BUILD_TYPE}" STREQUAL "Production")
    set(CMAKE_C_FLAGS   "-g -O3 -DNDEBUG")
    set(CMAKE_CXX_FLAGS "-g -O3 -DNDEBUG")
endif()

message("CMAKE_BUILD_TYPE        = ${CMAKE_BUILD_TYPE}")
message("CMAKE_C_FLAGS           = ${CMAKE_C_FLAGS}")
message("CMAKE_C_FLAGS_DEBUG     = ${CMAKE_C_FLAGS_DEBUG}")
message("CMAKE_C_FLAGS_RELEASE   = ${CMAKE_C_FLAGS_RELEASE}")
message("CMAKE_CXX_FLAGS         = ${CMAKE_CXX_FLAGS}")
message("CMAKE_CXX_FLAGS_DEBUG   = ${CMAKE_CXX_FLAGS_DEBUG}")
message("CMAKE_CXX_FLAGS_RELEASE = ${CMAKE_CXX_FLAGS_RELEASE}")
```

Function `message()` is for debugging. Most macros are for `cmake` executable only, whereas macros `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS` are used for compiler, which means we check those directives inside source code of `YLib` :

```
#ifdef NDEBUG
   ...
#endif
#ifdef LOGGING_TAG
   ...
#endif
```

Besides we can specify the values of directives on running `cmake` :

```
>> cmake -DCMAKE_C_COMPILER=$(which gcc)
         -DCMAKE_CXX_COMPILER=$(which g++)
         -DCMAKE_CXX_FLAGS="-DNDEBUG -DLOGGING_TAG -O3"
         -DCMAKE_BUILD_TYPE=Release
         -DIS_PRODUCTION=true  ../..
```

What is the difference between setting macros by `add_definitions(-g)` and by `set(CMAKE_CXX_FLAGS "-g -O2")`? *Please check ...*

- the former is cumulative (exists in old version `cmake`)
  ```
  add_definitions(-flag0)
  add_definitions(-flag1)
  add_definitions(-flag2)
  add_definitions(-Dmacro0)
  add_definitions(-Dmacro1=123)
  ```
- the latter sets all flags in single call, non-cumulative (exists in new version `cmake`)
  ```
  set(CMAKE_CXX_FLAGS "-flag0 -flag1 -flag2 -Dmacro0 -Dmacro1=123")
  ```

Option `CMAKE_EXPORT_COMPILE_COMMANDS` is also a commonly used `cmake` variable. When it is turned on it tells `cmake` to generate a database named `compile_commands.json` which indices all classes and functions for RTags. RTags is a software used both in terminal and in vim for fast jump to classes definitions / caller and callee. CTags is a similar, yet it does not rely on `compile_commands.json`.

```
>> cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ../..
>> cat build/debug/compile_commands.json
```

One more approach (but what are the differences?)

```
inside CmakeLists.txt :
target_compile_options(executable_name
    PRIVATE "-D_MY_MACRO_ABC=10U"
    PRIVATE "-D_MY_MACRO_DEF=10U"
)

inside *.h :
#ifdef _MY_MACRO_ABC
    static const std::uint32_t abc = _MY_MACRO_ABC;
#else
    static const std::uint32_t abc = 123; // default value if no compile option is defined
#endif
```

Part 5. Debugger `gdb` and `gdb-tui`

Ensure that `my_executable` is built with option `-g`. If we compile using `cmake`, we can set the following option :

```
>> mkdir build
>> cd build
>> cmake -DCMAKE_BUILD_TYPE=Debug ..  // cannot step-in in gdb if no symbol tables are generated
>> make -j4
>> objdump --syms Test/y | grep debug // check if whether symbol table actually exists
>> ldd Test/y                         // check dependency on other libraries
>> gdb Test/y
```

Debugger (and its text UI version) can be started in the following ways :

```
>> gdb -args my_executable args0 args1
(gdb) break algo.cpp:123
(gdb) run
```

```
>> gdb my_executable                 >> gdb -tui my_executable // for text-user-interface
(gdb) break algo.cpp:123
(gdb) run arg0 arg1
```

```
>> gdb                               >> gdb -tui // for text-user-interface
(gdb) file my_executable
(gdb) break algo.cpp:123
(gdb) run arg0 arg1
```

Commands can be abbreviated as the first character, `list` as `l`, `break` as `b`, `backtrace` as `bt` etc.

- breakpoints are set on particular location / line in code, execution paused on reaching that line
- watchpoints are set on particular variable, execution paused on modification of that variable

`(gdb) start`	equivalent to setting breakpoint on first line of `main` and then `run`
`(gdb) list fct`	list a session of code before and after function `fct`
`(gdb) list 123`	list a session of code before and after line number `123`
`(gdb) break fct`	set breakpoint in function `fct`
`(gdb) break 123`	set breakpoint in line `123`
`(gdb) break +10`	set breakpoint in `10th` next line
`(gdb) break classname::fct`	set breakpoint in function `fct` of class `classname`
`(gdb) break filename:fct`	set breakpoint in function `fct` of file `filename`
`(gdb) break filename:123`	set breakpoint in line `123` of file `filename`
`(gdb) break ... if var0 == 123`	set breakpoint conditional on `var0 == 123` where `...` denotes any choices above
`(gdb) watch var0`	set watchpoint for variable `var0` so that execution paused whenever it is modified set watchpoint can be done for in-scope variable only
`(gdb) next` or `n`	next line (it displays the line that it is going to run next)
`(gdb) until` or `u`	next line (it can slip for loop to avoid looping)
`(gdb) step` or `s`	step inside function
`(gdb) continue`	run until reaching next breakpoint
`(gdb) set var0 = 123`	set variable `var0` to `123`
`(gdb) print var0`	print variable `var0`
`(gdb) print var0.fct().mem`	print variable `var0.fct().mem`
`(gdb) info variables`	print all global variables and static variables
`(gdb) info locals`	print all local variables (i.e. stack variables)
`(gdb) info args`	print all arguments
`(gdb) info registers`	print all registers
`(gdb) info breakpoints`	print all breakpoints
`(gdb) disable 2`	disable breakpoint `#2` in the list

`(gdb) backtrace`
`#0 mult() at mult.cpp line 345`
`#1 algo() at algo.cpp line 200`
`#2 test() at test.cpp line 150`
`#3 main() at core.cpp line 40`
	print call stack, with stack-frame `#0` being the innermost layer in stack

`(gdb) ctrl-L`	clear `gdb` screen
`(gdb) kill`	stop running executable
`(gdb) quit`	quit `gdb`
`(gdb) info inferiors`	print all inferiors
`(gdb) info threads`	print all threads
`(gdb) info frame`	print whole callstack

`(gdb) inferior`	get current inferior ID
`(gdb) inferior n`	goto inferior `n`
`(gdb) thread`	get current thread ID
`(gdb) thread n`	goto thread `n`
`(gdb) frame`	get current frame ID
`(gdb) frame n`	goto frame `n`

How to access memory and disassemble?

If there are pointer variables in the code, we can use them to read a particular memory. Suppose we have :

```
char* c_ptr = "This is a string.";
```

then in `gdb` we can read the memory adjacent to `c_ptr` by command `x/` :

`(gdb) x/c c_ptr`	print 1 character starting from address `c_ptr`	output `'T'`
`(gdb) x/4c c_ptr`	print 4 characterw starting from address `c_ptr`	output `'T' 'h' 'i' 's'`
`(gdb) x/s c_ptr`	print 1 string starting from address `c_ptr`	output `"This is a string.\0"`
`(gdb) x/2x c_ptr`	print 2 heximals starting from address `c_ptr`	output `0x54686973 = "This"`
		output `0x20697320 = " is "`
`(gdb) disassemble fct`	disassemble function `fct`	

Backtrace

Backtrace, stacktrace, stack backtrace and dump stack all refer to the same thing. It is a snapshot of the callstack at any point during program execution, or the last snapshot when it crashes (recorded in the `coredump`).

Backtrace can be viewed inside `gdb` or invoked in a programmatic way :

- in `gdb`, the command is called `backtrace`, there are three ways to use `backtrace` in `gdb`
  - run the executable in `gdb` to any breakpoint, display the call stack by `backtrace`, you can resume execution afterwards
  - run the executable in `gdb` until it crashes, display the call stack by `backtrace`, you cannot resume execution, of course
  - run the executable in `bash` until it crashes, display the call stack by `gdb my_executable my_coredump`
- in `glibc`, the function is called `::backtrace`, which shows a snapshot of callstack
- in `boost`, the function is called `boost::stacktrace`, which shows a snapshot of callstack
  - both `::backtrace` and `boost::stacktrace` can be invoked anywhere in the program
  - both `::backtrace` and `boost::stacktrace` can be invoked inside callback of signal `SIGSEGV`, via `signal(SIGSEGV, my_callback);`

How generate coredump to in case of segmentation fault?

When program crash (segmentation fault), we need to find out `coredump` and apply `backtrace` on it. Here are the steps :

- compile executable using `-g` option, and extract the symbol-file from executable using some tools
- turn on core dump
- locate the core dump

Compilation with `-g` option means inserting symbol information into executable, which may introduce latency for production code. Thus for production code, in other to create a low-latency executable that supports core dump, what we do is to apply some kind of tools to decompose the executable into a *executable without symbol* plus *standalone symbol file*, this method offers the best speed while retaining the capability to backtrace.

Turn on core dump by :

```
>> ulimit -c unlimited
>> ./my_proj/my_executable
segmentation fault (core dumped)
```

Normally, core dump can be found locally :

```
>> gdb ./my_proj/my_executable -c core.1234
(gdb) backtrace
(gdb) frame 0
(gdb) list
```

Hint for debugging

Normally after completion of first version of class or algorithm, the first test usually involves segmentation fault, a useful trick is to run `gdb` and let it crashes, then `backtrace` to find out the lines of code that result in error. Very often these early stage bugs are easy to spot. Re-run the test after fixing the bugs, repeat the process, until it is stable and does not crash anymore. Yet there are still bugs as the test results are not the same as what expected, in this case, we need to print out something in order to study the discrepancy. If the discrepancy involves a few variables, we can observe them using `(gdb) print var`, otherwise we need to invoke some routines to print them into `std::cout`, we need to `(gdb) print function()`, where `print` is necessary.

```
(gdb) info locals
new_node = 0x7fffffffdf8
(gdb) print get_index(new_node)
$1 = 5
(gdb) print self_diagnosis()
```

Running `gdb` in `vim` / Running `gdb` with `tui`

To initiate `gdb` in `vim`

```
:packadd termdebug
:Termdebug
```

To switch windows

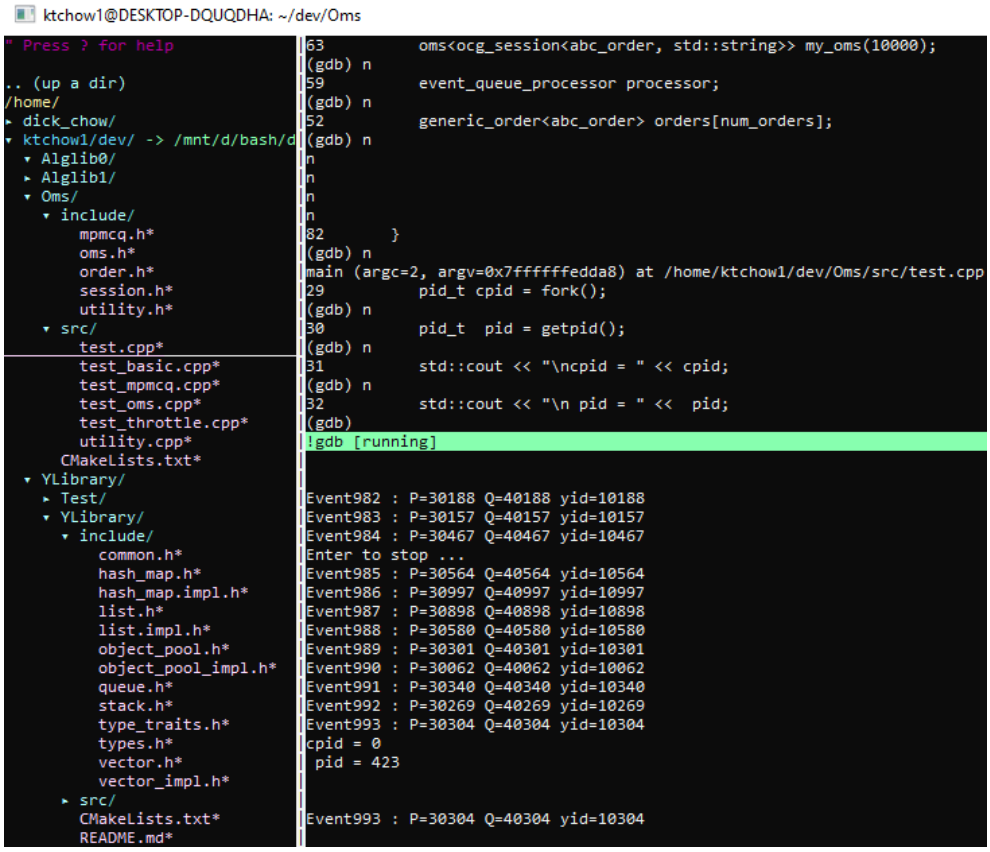
```
ctrl-w w
```

To make current windows bigger

```
ctrl-w L // note : must be capital L
```

We can also enter `tui` mode from normal mode by `layout src` command. There are many options for `layout` command, please check.

```
(gdb) layout src
(gdb) start
```



We can switch between the source window and command window by the following commands :

```
(gdb) focus src
(gdb) focus cmd
```

Running gdb with dashboard

Hidden file under home directory ~/.gdbinit is loaded everytime gdb is invoked. With appropriate python script like gdb-dashboard in GitHub (please search and install it), we can create fancy visualization for gdb.

Running gdb for multi-process

There are two occasions for running multiple processes in the same gdb session :

- we need to debug multiple processes involving interprocess-communication
- we need to debug a process that can fork children processes

In gdb state of a process is represented by an object called inferior, thus multiple inferiors are needed for multiple processes.

```
// This is occasion 1 : for debugging IPC
(gdb) add-inferior
(gdb) info inferiors          // we can see two inferiors (including original one)
(gdb) inferior 1              // switch to inferior 1
(gdb) file my_exe1            // load symbol from executable
(gdb) break file1.cpp:123
(gdb) run arg0 arg1
(gdb) inferior 2              // switch to inferior 2
(gdb) file my_exe2            // load symbol from executable
(gdb) break file2.cpp:234
(gdb) run arg0 arg1

// This is occasion 2 : executable that forks
(gdb) file my_exe
(gdb) break file1.cpp:123
(gdb) set detach-on-fork off  // when it forks, do not detach
(gdb) run arg0 arg1
(gdb) info inferiors          // we can see two inferiors, one for parent, one for child
```

Running gdb for multi-thread

There are various facilities for running multithread in gdb. When we run executable in gdb, on spawning new thread, it will notify :

```
(gdb) run
[New Thread 0x7ffff6e3a700 (LWP 20966)]
```

We can list all threads running in gdb by command info threads. For each thread, the following are shown :

- gdb-thread-id (usually in form of 1,2,3..)
- system-thread-id (usually a long HEX sequence)
- corresponding stackframe
- current thread

```
(gdb) info threads
  Id   Target Id                                Frame
  1    Thread 0x7ffff7fd9740 (LWP 20947) "y" 0x00007ffff7bbed2d in __GI___pthread_timedjoin_ex
* 2    Thread 0x7ffff6e3a700 (LWP 20966) "y" YLib::spinlock::lock (this=0x7fffffebba78)
  3    Thread 0x7ffff6639700 (LWP 21038) "y" std::atomic_flag::test_and_set
  4    Thread 0x7ffff5e38700 (LWP 21040) "y" std::atomic_flag::test_and_set
  5    Thread 0x7ffff5637700 (LWP 21059) "y" std::atomic_flag::test_and_set
```

The thread with asterisk is the current thread, gdb commands show information in current thread's perspective.

```
(gdb) thread 3                                  // switch current thread
(gdb) where                                     // show current local in callstack
(gdb) print queue.size()                        // show content of shared resource, which is a queue
```

We can set a break point in the code which :

- can stop all threads or
- can stop specific thread only
- can stop specific thread plus a condition

```
(gdb) break file1.cpp:123                       // all threads will stop on encountering line 123
(gdb) break file1.cpp:123 thread 3              // only thread 3 will stop on encountering line 123
(gdb) break file1.cpp:123 thread 3 if x > y     // only thread 3 will stop on encountering line 123 AND fulfilling condition
```

When we run executable in `gdb`, there are several modes :

- all-stop mode with scheduler-locking off (default) — stop all threads and resume all threads
- all-stop mode with scheduler-locking on — stop all threads and resume current thread only
- non-stop mode — stop current thread, while other threads keep running

What do these modes mean? The default mode is all-stop mode (in contrast to non-stop mode). All-stop mode means whenever the current thread is stopped, waiting to resume by `next` or `step`, other threads are stopped as well, thus we can observe the overall state of the process by switch thread and printing variables. When we press `next` again, all the threads will resume execution at their own speed and stop again either :

- when current thread complete `next` statement *OR*
- when other threads encounter a breakpoint

depending on whichever comes first, hence when execution stops again, we may already land on a different thread, like follows :

```
(gdb) n
[Switching to Thread 0x7ffff4e36700 (LWP 28094)]
Thread 6 "y" hit Breakpoint 3, /home/dick/dev/YLibrary/Test/src/Oms/test_mpmcq.cpp:124
(gdb) n
[Switching to Thread 0x7ffff4635700 (LWP 28095)]
Thread 7 "y" hit Breakpoint 2, /home/dick/dev/YLibrary/Test/src/Oms/test_mpmcq.cpp:116
(gdb) n
[Switching to Thread 0x7ffff3e34700 (LWP 28098)]
Thread 8 "y" hit Breakpoint 3, /home/dick/dev/YLibrary/Test/src/Oms/test_mpmcq.cpp:124
(gdb) n
[Switching to Thread 0x7ffff3633700 (LWP 28099)]
Thread 9 "y" hit Breakpoint 3, /home/dick/dev/YLibrary/Test/src/Oms/test_mpmcq.cpp:124
```

This option is called scheduler-locking off, which implies that we cannot `single-step` all threads together, as this is controlled by the scheduler (not by `gdb`). However, we can choose scheduler-locking on option, which can `single-step` the current thread only, while pausing all other threads, as a result, scheduler-locking on option forbids other threads from seizing the prompt.

```
(gdb) set scheduler-locking on
(gdb) set scheduler-locking off
(gdb) show scheduler-locking
```

Finally we have non-stop mode which allows other threads to run when current thread is paused, thus minimising intrusive effect on the whole system while debugging.

```
(gdb) set non-stop on
(gdb) set non-stop off
(gdb) show non-stop
```

Sometimes when debugging multi-thread program, both all-stop mode and non-stop mode do not help, it is still difficult to capture the moment when problem happens. Thus it is inevitable to use intrusive debugging technique, which involves adding debug code into the program for checking abnormal condition and list the states of program.

Running `gdb` to trace STL code

To trace STL container code, we can write a snipplet with STL container, build it with `-g` option, then run the program with `gdb`, set a breakpoint and step inside the container :

```
>> gdb ./my_executable
(gdb) b test.cpp:123
(gdb) run my_arg
(gdb) s
(gdb) s
...
(gdb) s
```

Finally we can see that STL codes can be found inside `/usr/include/c++/10/bits/vector.tcc`.

Attach `gdb` to a running executable

This is useful for debug of a never-ending loop process.

- once attach `gdb` to the process, all threads are paused (so we can jump across threads, jump across frame, call backtrace etc)
- once detach `gdb` from the process, all threads will resume (so we can attach again to see if each thread is making progress)

```
>> gdb ./my_executable -p pid
(gdb) type gdb commands here, snapshot of variables / callstack / threads is shown,
      however the real states keep changing due to the running process
(gdb) info threads
(gdb) thread 3
(gdb) bt
```

Running `gdb` for release version

There are 4 combinations when building a library :

`compile option : -O0 -g`	lowest speed with minimum optimization, contain debug symbols
`compile option : -O0`	lowest speed with minimum optimization, no debug symbols
`compile option : -O3 -g`	medium speed with maximum optimization, contain debug symbols
`compile option : -O3`	highest speed with maximum optimization, no debug symbols

Can we achieve highest speed, yet maintaining debug-ability? Yes, firstly build executable with both `-O3` and `-g` options, decompose the executable into (1) a thin `-O3` executable without debug symbol plus (2) a debug symbol file. Decomposition is done via 2 steps :

`>> objcopy --only-keep-debug my_exe my_symbol`	convert executable `my_exe` into a symbol file `my_symbol`, while `my_exe` is unchanged
`>> strip --strip-debug --strip-unneeded my_exe`	strip executable `my_exe` into a thin one by removing symbols

The size of `my_exe` is about 5% of the original, while the size of `my_symbol` is about 95% of the original. We can run `my_exe` as usual :

```
>> my_exe
>> gdb my_exe
no symbol loaded ...
```

In order to run `gdb`, we have to merge the stripped `my_exe` and `my_symbol` into a debug-able executable using the same function :

`>> objcopy --add-gnu-debuglink=my_symbol my_exe`	which adds `my_symbol` into the stripped `my_exe` (via links), becoming debug-able
`>> gdb -c coredump my_exe`	where `my_exe` is just slightly larger than the stripped version (since it is links)

Part 5A. Debug Infinity Loop

YTL in FPGA machine

Deploy HK options trading program to machine `fpga01` (10.250.6.86) running Centos. Lets start from scratch with generating `ssh` key in local machine `local` (10.250.6.80). Don't forget to set the appropriate access rights for different `ssh` related files :

```
>> chmod 755 ~/.ssh
>> cd ~/.ssh
>> ssh-keygen -t rsa -b 4096 -C "from local to fpga01"
>> scp fpga01.pub dev@10.250.6.86:/home/dev/.ssh
>> ssh dev@10.250.6.86
>> cat /home/dev/.ssh/fpga01.pub >> /home/dev/.ssh/authorized_keys
>> chmod 644 /home/dev/.ssh/authorized_keys
```

Usually, the standard C++ library `libstdc++` in Centos is not as updated as my local machine, hence I need to copy my local `libstdc++` to `fpga01`, and set the appropriate symbolic link. In my local machine :

```
>> find / -name libstdc++*
>> ll /lib64/libstdc++*
/lib64/libstdc++.so.6.0.28                                  real library
/lib64/libstdc++.so.6 --> /lib64/libstdc++.so.6.0.28        symbolic link
>> scp /lib64/libstdc++.so.6.0.28 dev@fpga01:/home/dev      why not copy to dev@fpga01:/lib64 directly? since it needs access right
>> ssh dev@fpga01
>> mv /home/dev/libstdc++.so.6.0.28 /lib64
>> ln -sf /lib64/libstdc++.so.6.0.28 /lib64/libstdc++.so.6  symbolic link setup
```

Now, let's run the multi-thread program.

```
>> cd /home/dev/hk-options
>> ./run.sh
```

All threads make no progress (seem to be stuck as some points), how to debug? First of all, install `htop` :

```
>> sudo yum install htop
>> htop                                                     press F4 to filter process-of-interest
                                                            press F5 to display process as tree
>> sudo htop                                                press F9 to kill highlighted process
```

Secondly identify threads inside the program :

```
smallest pid   =       main thread
100% cpu usage =    service thread spinning event queue
  0% cpu usage = subservice thread blocking on IO (socket, OAPI etc)
  0% cpu usage = threadpool waiting on mutex
```

If this is not clear enough, we can identify the threads by attaching `gdb`, and see what the threads are doing in backtrace `bt` :

```
>> gdb ytl -p main_pid
```

Once enter `gdb`, all threads are paused, we can switch between threads, backtrace it and jump across frames.

```
(gdb) info threads
(gdb) bt
(gdb) fr 3
(gdb) fr 4
(gdb) p my_variable
```

Try to get more hints to associate thread ID to our program threads. After that quit `gdb` and let the threads run. Attach `gdb` again, the stuck thread is the one that make no progress, which block other threads (a kind of deadlock). Usually the main cause is that stuck thread threw exception that no one catch and it terminated.

Part 6. Google test (Alternative C++ testing framework : `catch2`)

Google test is a library including header `gtest.h` and static library `libgtest.a`. Download it with `git` and install by following standard `cmake` procedures. Finally copy the header and static library to `/usr/local`.

```
>> git clone https://github.com/google/googletest
>> cd googletest/googletest
>> mkdir build
>> cd build
>> cmake ..
>> make -j4
>> cd ..
>> cp -r include/gtest /usr/local/include
>> cp -r build/lib*.a /usr/local/lib
```

Then we can write our own test. In the test, we need :

- include `gtest/gtest.h`
- link with `libgtest.a`
- two-liners `main` to start google test
- format 1 `TEST(test_suite_name, test_case_name)`
- format 2 `TEST_F(fixture_suite_name, test_case_name)` where `fixture` is a class derived from base class `testing::Test`

Various comparison macros are shown below. Finally call `my_google_test()` in `main()` :

```cpp
#include<gtest/gtest.h>

void my_google_test(int argc, char* argv[])
{
    testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS(); // This macro will search entire project for all TEST and TEST_F macros.
}

TEST(sample_suite0, case_boolean)
{
    std::uint32_t x = 123U;
    EXPECT_TRUE (x==123U);
    EXPECT_FALSE(x!=123U);
}

TEST(sample_suite0, case_integer)
{
    std::uint32_t x = 123;
    EXPECT_EQ(x, 123U);
    EXPECT_NE(x, 124U);
    EXPECT_LT(x, 124U);
    EXPECT_LE(x, 123U);
    EXPECT_GT(x, 122U);
    EXPECT_GE(x, 123U);
    EXPECT_EQ(x*x, 123U*123U);
}

TEST(sample_suite0, case_double)
{
    double x = 1.23;
    EXPECT_NEAR(x+0.0001, 1.23, 0.001);
    EXPECT_NEAR(x-0.0001, 1.23, 0.001);
    EXPECT_NEAR(x+0.0011, 1.23, 0.001);
    EXPECT_NEAR(x-0.0011, 1.23, 0.001);
}

TEST(sample_suite0, case_string)
{
    char x[] = "ABC-DEF-GHIJ";
    EXPECT_STREQ(x, "ABC-DEF-GHIJ");
    EXPECT_STRNE(x, "ABC-DEF-GHIX");
    EXPECT_STRCASEEQ(x, "abc-DEF-ghij");
    EXPECT_STRCASENE(x, "xbc-DEF-ghij");
}

class my_fixture : public ::testing::Test // beware the character case "testing::Test"
{
public:
    my_fixture()    { std::cout << "*** my_fixture() ";       }
   ~my_fixture()    { std::cout << ">> ~my_fixture() ***\n";  }
    void SetUp()    { std::cout << ">> SetUp() ";             }
    void TearDown() { std::cout << ">> TearDown()";           }
};

// It will invoke my_fixture::my_fixture() and my_fixture::SetUp() before each test case belonging to my_fixture.
TEST_F(my_fixture, sample_case0)
{
    std::uint32_t x = 123;
    EXPECT_EQ(x, 123U);
}
// It will invoke my_fixture::~my_fixture() and my_fixture::TearDown() after each test case belonging to my_fixture.
```

```
ktchow1@DESKTOP-DQUQDHA:~/dev/Test/build$ ./Test
[==========] Running 6 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 4 tests from sample_suite0
[ RUN      ] sample_suite0.case_boolean
[       OK ] sample_suite0.case_boolean (0 ms)
[ RUN      ] sample_suite0.case_integer
[       OK ] sample_suite0.case_integer (0 ms)
[ RUN      ] sample_suite0.case_double
[       OK ] sample_suite0.case_double (0 ms)
[ RUN      ] sample_suite0.case_string
[       OK ] sample_suite0.case_string (0 ms)
[----------] 4 tests from sample_suite0 (5 ms total)

[----------] 2 tests from my_fixture
[ RUN      ] my_fixture.sample_case0
[       OK ] my_fixture.sample_case0 (0 ms)
[ RUN      ] my_fixture.sample_case1
[       OK ] my_fixture.sample_case1 (0 ms)
[----------] 2 tests from my_fixture (5 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test cases ran. (16 ms total)
[  PASSED  ] 6 tests.
```

Part 7. Valgrind

There are many facilities in valgrind, here are some of them (I have tried all these for `YLib::sorted_list<T>`) :

- memcheck (default on) for detection of memory leak / duplicated memory free / uninit memory access
- helgrind for detection of thread synchronization error (such as deadlock)
- callgrind for measuring time spent in each function
- cachegrind for measuring instruction cache miss (branch prediction) and data cache miss in each function

They are invoked respectively as following. Since memcheck is default on, `--tool=memcheck` can be omitted.

```
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all --verbose my_path/my_executable
valgrind --tool=helgrind   my_path/my_executable
valgrind --tool=callgrind  my_path/my_executable
valgrind --tool=cachegrind my_path/my_executable
```

Extra reports are generated after running `callgrind` and `cachegrind`, they can be found in current directory, named after process ID :

```
>> ll callgrind.out.12345
>> ll cachegrind.out.34567
```

They are hugh text file, non human readable. Thus it is better to view them with a tools, called `kcachegrind`. Both use the same tools.

Memcheck

Memcheck detects if there is memory leakage. The number in between `==pid==` is process id.

```
valgrind my_path/my_executable
==15888== HEAP SUMMARY:
==15888==     in use at exit: 0 bytes in 0 blocks
==15888==   total heap usage: 504,261 allocs, 504,261 frees, 300,494,175 bytes allocated
==15888==
==15888== All heap blocks were freed -- no leaks are possible
==15888==
==15888== For counts of detected and suppressed errors, rerun with: -v
==15888== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memcheck notifies if there is leakage. For example, when I changed the stack memory of a fixed size ring buffer into heap memory in order to support runtime-adjustable buffer size, I forgot to deallocate memory in destructor :

```
==15776== HEAP SUMMARY:
==15776==     in use at exit: 283,140,096 bytes in 181 blocks
==15776==   total heap usage: 504,261 allocs, 504,080 frees, 300,494,175 bytes allocated
==15776==
==15776== LEAK SUMMARY:
==15776==    definitely lost: 270,557,184 bytes in 173 blocks
==15776==    indirectly lost: 0 bytes in 0 blocks
==15776==      possibly lost: 12,582,912 bytes in 8 blocks
==15776==    still reachable: 0 bytes in 0 blocks
==15776==         suppressed: 0 bytes in 0 blocks
==15776== Rerun with --leak-check=full to see details of leaked memory
==15776==
==15776== For counts of detected and suppressed errors, rerun with: -v
==15776== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memcheck notifies if memory is repeatedly deallocated. For example, when I implemented the move assignment for ring buffer but forget to reset members of `rhs` to `nullptr`, as a result the same piece of memory is deallocated once when `this` goes out of scope, once again when `rhs` goes out of scope.

```
==20936== Invalid free() / delete / delete[] / realloc()
==20936==    at 0x5A7873B: operator delete[](void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==20936==    by 0x14BAFD: YLib::sorted_list_manager<unsigned int, std::less<unsigned int> >::~sorted_list_manager()
==20936==    by 0x14A439: YLib::sorted_list<unsigned int, std::less<unsigned int> >::~sorted_list()
==20936==    by 0x14A2D3: test_resize() (in /home/dick/dev/YLibrary/build/Test/y)
==20936==    by 0x14A313: test_hitter() (in /home/dick/dev/YLibrary/build/Test/y)
==20936==    by 0x2694F4: main (in /home/dick/dev/YLibrary/build/Test/y)
==20936==  Address 0x6c54da0 is 0 bytes inside a block of size 960 free'd
```

Other common errors include "uninitialized bytes read". In the following, two `std::uint32_t` are read without initialization.

```
==2650==  Address 0x7f79a3c is in a rw- anonymous segment
==2650==  Address 0x7f79a40 is in a rw- anonymous segment
```

Helgrind

Helgrind detects possible synchronization error for threads sharing same address space, such as :

- detect deadlock
- unlock a mutex which is not locked
- unlock a mutex which is not owned by the thread
- recursive lock a non-recursive mutex ... etc

Here is what it prints when there is no problem :

```
>> valgrind  --tool=helgrind  my_path/my_executable
==7986== For counts of detected and suppressed errors, rerun with: -v
==7986== Use --history-level=approx or =none to gain increased speed, at
==7986== the cost of reduced accuracy of conflicting-access information
==7986== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Callgrind and Kcachegrind

Callgrind counts the number of calls and measures computation time *(not latency)* spent on each function of an executable.

```
>> valgrind  --tool=callgrind  my_folder/my_executable  arg
==13956== Callgrind, a call-graph generating cache profiler
==13956== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==13956== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13956== Command: my_folder/my_executable
==13956==
==13956== For interactive control, run 'callgrind_control -h'.
==13956==
==13956== Events    : Ir
==13956== Collected : 37521098423
==13956==
==13956== I refs : 37,521,098,423
```

It generates a report named `calgrind.out.13956` (the number is process *ID*) which can be displayed by `kcachegrind` :

```
>> kcachegrind calgrind.out.13956
```

There are 5 columns, showing 5 pieces of informations about functions called :

```
incl.    = %time used in a function (when the function is inside callstack)
self     = %time used in a function (when the function is on the top of callstack)
called   = number of invocation of the function
function = namespace and function name
location = library in which the function is defined (such as libstdc++.so.6.0.28 or my_algo.a)
```

There are some observations :

- if sort the functions by `incl.`, `main()` must be the highest one, with 100% computation load
- if sort the functions by `incl.`, the highest ranked functions usually rank the lowest in `self`, as they are usually testing function
- if sort the functions by `self`, and being called 447M times, there we should find out :
- whether this function is fast enough
- whether this function should be called so many times
- then we have to study the callers of this function, this can be done by double-clicking that row
- callers will then be shown on RHS windows, there is a `call-graph` tab at the bottom, which gives a DAG visualization

If we want to gauge a specific piece of code (rather than the whole program), we can use intrusive measurement, by adding macros surrounding the code of interest, like the following :

```
CALLGRIND_TOGGLE_COLLECT; // Don't forget semicolon.
// CALLGRIND_START_INSTRUMENTATION;
code_of_interest();
// CALLGRIND_STOP_INSTRUMENTATION;
CALLGRIND_DUMP_STATS; // Don't forget semicolon.
```

If we want to see the code in `kcachegrind`, make sure the program is compiled with `-g` option, which does also work with `-O3` option. Besides, add option `--collect-atstart=no` when running `callgrind`, which means no data collection at start of program.

```
>> valgrind  --tool=callgrind  --collect-atstart=no  ./Test  123
>> kcachegrind callgrind.out.13956
```

Cachegrind and Kcachegrind

Cachegrind counts the number of (instruction and data) cache miss for each function in an executable.

```
>> valgrind --tool=cachegrind my_path/my_executable
==29105== I   refs:      37,521,122,288
==29105== I1  misses:            41,523
==29105== LLi misses:             4,223
==29105== I1  miss rate:           0.00%
==29105== LLi miss rate:           0.00%
==29105==
==29105== D   refs:      23,445,382,828  (14,785,988,354 rd   + 8,659,394,474 wr)
==29105== D1  misses:       538,952,029  (   532,615,708 rd   +     6,336,321 wr)
==29105== LLd misses:            78,657  (        11,115 rd   +        67,542 wr)
==29105== D1  miss rate:            2.3% (           3.6%     +           0.1%  )
==29105== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==29105==
==29105== LL refs:          538,993,552  (   532,657,231 rd   +     6,336,321 wr)
==29105== LL misses:             82,880  (        15,338 rd   +        67,542 wr)
==29105== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

`I1`	first level instruction cache miss due to incorrect branch prediction
`LLi`	last level instruction cache miss due to incorrect branch prediction
`D1(rd)`	first level data cache miss when reading
`D1(wr)`	first level data cache miss when writing
`LLd(rd)`	last level data cache miss when reading
`LLd(wr)`	last level data cache miss when writing
`LL(rd)`	sum of above items (which?)
`LL(wr)`	sum of above items (which?)

There are some observations.

- The number of `LLd` miss is the number of `D1` miss
- The number of instruction miss is closed to zero for simple test function, because they usually do not have `if` nor `virtual`.
- How to make use of this tools? Run `cachegrind` on two implementations (one cache friendly, one not) to verify the former.

We can load its report `cachegrind.out.pid` using `kcachegrind`.

```
>> kcachegrind cachegrind.out.13956
```

On the LHS, we have a list of all functions, there are a few columns :

```
incl.    = number of time a function is called
self     = number of time a function is called (exactly the same as incl. in cachegrind)
function = namespace and function name
```

By double clicking the function of interest, it will breakdown the cache miss numbers for that function on RHS.

Options for Valgrind

The google test for `YLib` may run out of stack as we use a lot of stack memory for containers and object pool, we should assign more stackframe to the test through option `max-stackframe`. We can stop valgrind on first error with `exit-on-first-error`. Moreover for multi thread testing, valgrind will put all threads in the same core, overriding the OS's scheduler, it invokes its own scheduling algorithm, which may not be a fair one, leading to extremely slow valgrind time, its sometimes better to turn on `fair-sched`.

```
>> valgrind --max-stackframe=8000000
            --exit-on-first-error=yes
            --fair-sched=yes
              my_executable arg0 arg1
```

Part 9. Doxygen

Doxygen is started with `/*!` or `//!`. There are two types of comments : brief comment (one liner) and details (multiple lines).