CompSys 202 / MechEng 270
Object oriented design and programming

Lecture 2: *Redirection, pipelines, expansion*

Nasser Giacaman

Department of Electrical and Computer Engineering
The University of Auckland, New Zealand

Aims of this lecture

Today we will:

- Learn how to redirect command I/O to and from files
- Learn how to connect multiple commands with pipelines
- Learn about shell expansion (pathname, arithmetic, command, ...)
- Learn some of the history features available
- Introduce the **multi-user** aspect of Linux
    - Learn the basics of Linux file **permissions**
- Learn a little on C++ compiling

I/O redirection: In Linux, "everything is a file"

- What happens when we execute a command?
  - We usually provide it with some *input*,
  - Then it either executes successfully (and we might get some *output*), or it doesn't execute successfully (and we get some *error*).
- Most programs will:
  - Take input from **standard input** (stdin),
  - Send results to **standard output** (stdout),
  - Send messages to **standard error** (stderr).
- **stdout** and **stderr** are actually files linked to the screen, and not saved onto the disk.
  - stdout is file descriptor 1
  - stderr is file descriptor 2
- **stdin** is a file attached to the keyboard.
  - stdin is file descriptor 0
- I/O redirection allows us to change where output goes, and where input comes from.

Redirect standard output

- The following neat trick will create an empty file called “empty.txt”:
  - `> ` *[nothing here]* `> empty.txt`
  - `> ls -l empty.txt` *[empty.txt is zero bytes]*
- Let’s try again:
  - `> echo ‘‘Hello’’ > empty.txt`
  - `> ls -l empty.txt` *[empty.txt is no longer empty!]*
- Let’s try another command:
  - `> ls /bin > contents.txt` *[no output displayed on screen]*
  - `> ls -l contents.txt` *[non-empty file created]*
- Let’s try that again:
  - `> ls /binn > contents.txt` *[error displayed on screen]*
  - `> ls -l contents.txt` *[file overwritten, now empty]*
- Sometimes we want to append rather than overwrite:
  - `> ls /bin >> contents.txt`
  - `> ls /bin >> contents.txt`
  - `> ls -l contents.txt` *[contains both outputs]*

Redirect standard error

- Recall how here we still got something displayed to screen:
  - `> ls /binn > contents.txt` *[the same as "1>"]*
- ">" is shorthand for "1>", which is file descriptor 1 (stout)
- To redirect stream 2 (i.e. stderr) to a file:
  - `> ls /binn 2> contents.txt`
- But what if there was no error?
  - `> ls /bin 2> contents.txt` *[output displayed on screen]*
- We can redirect **both** stdout and stderr at the same time:
  - `> ls doesThisFolderExist &> contents.txt`
- Side note: traditionally, this was done using "2>&1", which is used more awkwardly and clearly not as elegant as "&>"

/dev/null

- `/dev/null` is the Linux *bit bucket*, a device file that accepts input and discards it – handy when we don't want to save a command's output.
- Geeky Linux humour:
  - "Please send complaints to `/dev/null`"
    - *Translation:* Don't bother sending complaints
  - "My mail got archived in `/dev/null`"
    - *Translation:* My mail was deleted
  - "Redirect to `/dev/null`"
    - *Translation:* Go away
- `> ls doesThisFolderExist 2> /dev/null` *[stderr discarded]*

Pipelines

- Redirection allowed us to:
  - Send output from a command to a file, and
  - Take a file's contents as input to a command
- **Pipelines** allow us to send output from one command as input to another command
- View contents of the following directories:
  - `> ls /bin /usr/bin` *[lists the contents in separate lists]*
- Use a pipeline to `sort` the output within same list:
  - `> ls /bin /usr/bin | sort` *[pipe output as stdin for sort]*
- Only show `uniq`ue lines:
  - `> ls /bin /usr/bin | sort | uniq`
- Only show `d`uplicated lines:
  - `> ls /bin /usr/bin | sort | uniq -d`

Pipelines

- Count:
  - `> ls /bin /usr/bin | sort | uniq | wc -l`
- Is sorting necessary?
  - `> ls /bin /usr/bin | uniq | wc -l`
  - `> man uniq` *[“Filter adjacent matching lines from ....”]*
- Print only lines that match a pattern:
  - `> ls /bin /usr/bin | sort | uniq | grep zip` *[anywhere]*
  - `> ls /bin /usr/bin | sort | uniq | grep ^zip` *[starting]*
  - `> ls /bin /usr/bin | sort | uniq | grep zip$` *[ending]*
  - `> ls /bin /usr/bin | sort | uniq | grep -v zip` *[invert]*
  - `> ls /bin /usr/bin | sort | uniq | grep -i ZIP` *[insensitive]*

Expansion

- What happens when you provide arguments to a command? Does it go straight to the command? Or is it first "intercepted" by the shell?
  - `> echo one two three four`
  - `> echo *` *[does the echo command or the shell interpret "*"?]*
  - `> *` *[seems that the shell is preprocessing "*" → **expansion**]*
- The shell expands any special characters that we enter on the command line, before the command is executed. Sometimes, it doesn't expand the characters if the expansion does not produce anything useful. So, it just passes those characters to the command.

Expansion

- **Pathname expansion**
  - `> echo D*`
  - `> echo *s`
  - `> echo *` *[no expansion if empty directory, so passes "*" to echo]*
- **Arithmetic expansion**
  - `> $((`*`expression`*`))` *[supports only integers]*
  - `> echo $((11/2))`
  - `> echo $((11%2))`
  - `> echo 11 divided by 2 = $((11/2)) remainder $((11%2))`
- **Brace expansion**
  - `> echo h{a,ea,o,,u}t` *[no spaces between commas]*
  - `> mkdir {jan,feb}-{1983..1993} {jan,feb}-{2011..2016}`
  - `> echo ABC backwards:  {Z..A}`
  - `> echo {10..1}, Lift off!` *[comma is touching brace]*

Multiple multi's!

- Back in the days, computers were large and expensive. The time of the computer was more valuable than human time!
- **Multi-programming**
  - Rather than run the program from start to end, it gets swapped out when it waits for I/O with another program that is ready to run
- **Multi-tasking / time-sharing**
  - Fairness for multiple programs, especially when they hogged the CPU
- **Multi-processor**
  - A system with multiple CPUs
- **Multi-core**
  - A CPU with multiple cores
- **Multi-threading**
  - The software is coded with (logically) multiple streams of instructions
- **Multi-user**
  - The operating system supports multiple simultaneous users

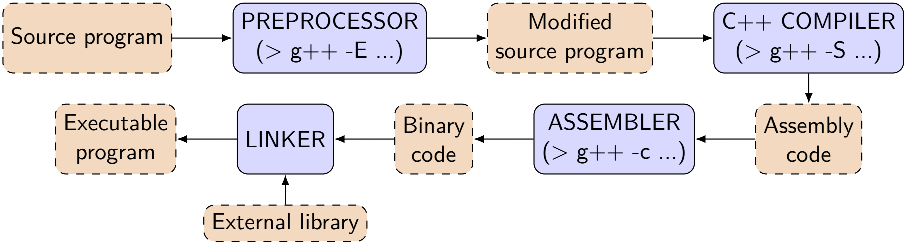
Permissions

- `> ls -l`
-
```
-rw-r--r-- 1  nas users 37977 Jun 12 15:41 handout.pdf
drwxr-xr-x 14 nas users 392   Mar 23  2003 oop_stuff
-rwxr-xr-x 1  nas users 582   Nov  5 17:10 run.sh
```
- File type: `-` (regular file), `d` (directory), `l` (symbolic link), ...
- File permissions: read, write, execute
  - Owner's permissions, group's permissions, other's permission

- **chmod**: change file mode *bits*
  - 0 (`---`), 1 (`--x`), 2 (`-w-`), 3 (`-wx`), 4 (`r--`), 5 (`r-x`), 6 (`rw-`), 7 (`rwx`)
  - Mode bits for run.sh: 755: 7=111(`rwx`), 5=101(`r-x`), 5=101(`r-x`)
  - `> chmod 744 run.sh` *[only owner can execute run.sh]*
  - `> chmod u+x file` *[add execute permission for owner]*
  - `> chmod o-r file` *[remove read permission for others]*
  - `> chmod g-wx file` *[remove write and execute permission for group]*
  - `> chmod +x file` *[or]* `a+x` *[add execute permission for all]*

The C++ compilation process

- The ***preprocessor*** performs an initial translation to prepare the source for more efficient processing (for example removing comments and white space, or expanding macros)
- The ***compiler*** translates the modified source into assembly code
- The ***assembler*** translates assembly code (e.g. add $8, $10, $9) into binary code (e.g. 00000001001010100100000000100000). But, this binary code is not executable yet.
- The ***linker*** creates an executable by linking the binary code with other libraries (pre-compiled code your program references).



The C++ compilation process

- Consider the following content inside a file called `source.cpp`

```
#include <iostream>
#define MAX 100
using namespace std;
int main() {
   cout << "The maximum number is ";
   cout << MAX << endl;
   return 0;
}
```

- Invoking the *preprocessor only*:
  - `> g++ -E source.cpp > modified_source.cpp`
- Stopping at the *assembly code:*
  - `> g++ -S source.cpp` *[creates* `source.s` *with assembly code]*
- Stopping at the *binary code*:
  - `> g++ -c source.cpp` *[creates* `source.o` *with binary code]*
- Full compilation:
  - `> g++ source.cpp` *[generates an executable called* `a.out`*]*

make and Makefiles

- Real software projects will involve 10s to 100s of source files, and the compilation process can quickly get more complicated.
- Sometimes, you want to build the project under different conditions (e.g. release vs debugging).
- `make` is a tool to automate the build process. It figures out which part of the build to execute, depending on what files have changed.
- The Makefile defines targets in the following format:

```
target: [dependencies...]
[tab]<first command>
[tab]<second command>
```

- If the *dependency* is a *changed file*, then the target is executed.
- If the *dependency* is another *target*, then the target is evaluated.
- Using `make`:
  - `> make` *[looks for file called* `Makefile`*, defaulting to the first target]*
  - `> make -f MyMakefile` *[specify a different file]*
  - `> make clean` *[execute a particular target, here called* `clean`*]*

An example Makefile

```
CC= g++     [define which compiler to use]
CCFLAGS= -Wall -c  [flags for the compiler]
FINAL= myprogram  [the final executable program]

all: main.o Wheel.o Car.o  [default target, 3 dependencies]
[tab]$(CC) -o $(FINAL) main.o Wheel.o Car.o

main.o: main.cpp  [main.o only depends on main.cpp]
[tab]$(CC) $(CCFLAGS) main.cpp

Car.o: Car.cpp Wheel.o [Depends on Car file & Wheel target]
[tab]$(CC) $(CCFLAGS) Car.cpp

Wheel.o: Wheel.cpp
[tab]$(CC) $(CCFLAGS) Wheel.cpp

clean:                [no dependencies]
[tab]rm -rf *.o $(FINAL)
```

Handy tips: completion and history

- **Auto-complete**
  - When you type part of a command or path, use the `Tab` key.
  - If multiple options are available, it auto-completes as much as possible.
  - If you press the `Tab` key twice, it displays all possible options.
- **Auto-stored history**
  - `> history | tail` *[display the last 10 lines of the history]*
  - `> history | grep alias` *[search for occurrence in history]*
  - `> !23` *["history expansion": re-execute the 23rd line in the history]*
  - Up/down arrow buttons to scroll history