

# 셸/앤서블 기본

Shell/Ansible

# 랩 구성

랩은 WSL2기반으로 진행 합니다. 만약, 윈도우 10/11 Pro버전이 아닌 경우, "VirtualBox"나 혹은 "VMware Player"를 사용하셔도 됩니다.

아래 명령어를 "관리자 권한 콘솔"으로 실행 합니다.

```
> dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart  
> wsl --set-default-version 2
```

<https://www.microsoft.com/store/apps/9MSVKQC78PK6>

WSL2에서 사용하는 리눅스 배포판은 "rocky", "ubuntu", "centos-8/9-stream"사용이 가능하다. 혹은, 저장소에 있는 가상 머신 기반의 랩 매뉴얼 "101-랩-리눅스 셸"를 참조하여도 된다.

# 랩 설계

셸 랩은 좀 더 효율성 있는 교육을 위해서 쿠버네티스와 함께 셸 스크립트 활용에 대해서 학습이 주요 목적이다.

1. 명령어와 배시 셸의 관계
2. TUI 도구 및 CLI도구 작성 방법
3. 앤서블에서 배시 셸 활용
4. 쿠버네티스 관리도구 구성(아마도...?)

# 셸 개념

셸 종류

컨셉트 및 간단한 기능 소개

# 셸 종류(ksh)

KSH은 유닉스에서 많이 사용하는 셸 중 하나. 보통 다음과 같은 유닉스 시스템에서 많이 사용한다.

- Unix 계열(HP, AIX, Solaris)
- Linux
- BSD계열

리눅스도 ksh를 지원하지만, 일반적으로 잘 사용하지는 않는다.

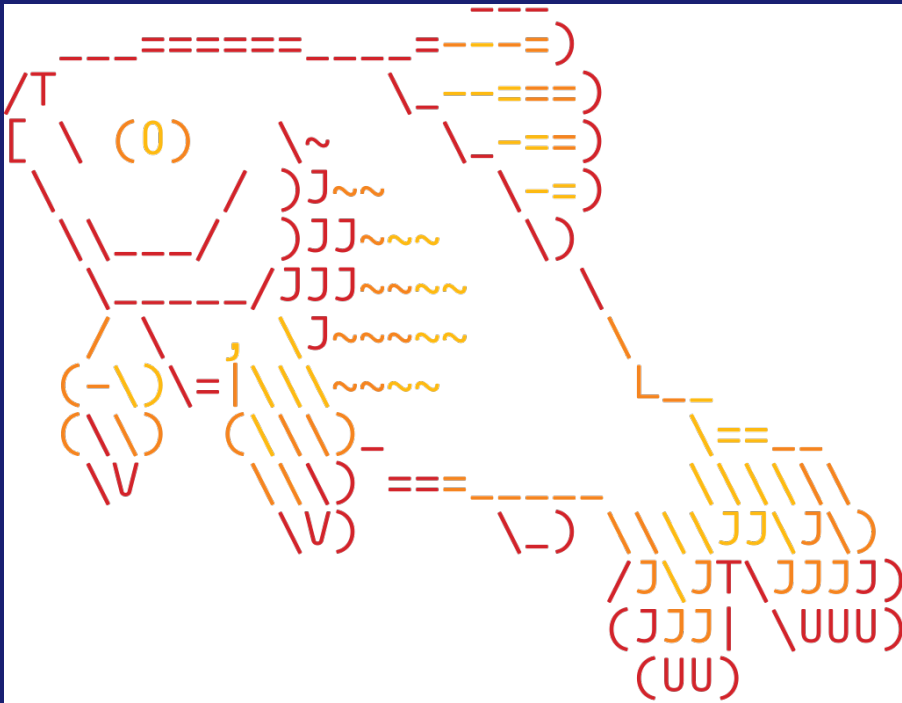
# 셸 종류(bash)

리눅스 배포판에서 많이 사용하는 셸. 일반적으로 모든 리눅스는 배시 셸 기반으로 구성되어 있다. 배시는 본래 Bourne Shell를 기반이며, 이를 기반으로 개선이 되었다. 이러한 이유로 배시는 /bin/sh와, /bin/bash로 링크 혹은 파일로 되어 있다.

배시는 POSIX표준을 따르고 있기 때문에, 대다수 셸과 호환성이 유지가 된다.



# 셸 종류(fish)



Fish셸은 그나마 리눅스 셸 중에서 최신 셸 중 하나이다. 사용자 친화적인 셸이지만, POSIX와 호환이 되지 않기 때문에 표준 시스템에서는 사용을 권장하지 않는다. 하지만, 표준적인 빌트인 명령어 export, set, alias와 같은 부분은 호환이 된다.

이 교육에서는 fish에 대해서는 다루지 않는다.

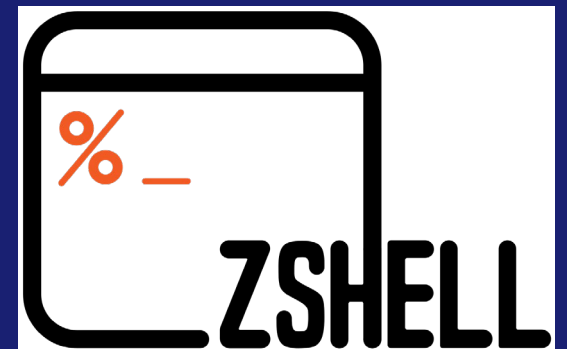
# 셸 종류(zsh)

zsh는 오랫동안 사용하던 셸이며, 1990년도에 만들어졌다. zsh는 ksh, tcsh에서 사용하는 명령어 자동완성 및 함수와 같은 부분이 호환이 된다. zsh은 마이크로소프트에서 윈도우 UnixUtils에 포함 시키면서 다시, 많은 사람들이 사용하기 시작하였고, 현재는 MacOS, Kail Linux에서는 기본셸로 사용하고 있다.

현재 확장 기능인 "Oh My Zsh"이라는 사용자 커뮤니티를 통해서 플러그인 및 테마와 같은 확장 기능을 추가적으로 설치 및 구성이 가능하다.

하지만, zsh에서 셸 스크립팅을 사용하는 경우, POSIX표준을 완벽하게 지원하지 않기 때문에, 스크립트 작성이 필요한 경우, bash를 권장한다.

**ZSH 스크립트:** <https://rwx.gg/advice/dont/zsh/>





# KSH/BASH 차이점

ksh와 bash의 큰 차이점은 실제로는 거의 없다.

하지만, [Conflicts between ISO/IEC 9945 \(POSIX\) and the Linux Standard Base](#)의 내용에 충돌이 되기 때문에 서로 호환이 되지 않는 부분이 있다. 예를 들어서 LSB에서는 `/usr/xpg4/bin`를 사용하지 않으며 또한 솔라리스도 이러한 POSIX규격을 따르지 않는다.

리눅스 및 솔라리스 경우 2004년도 기준으로 양쪽 진영은 표준 사양을 변경하였다. 이를 통해서 리눅스 및 솔라리스 같은 유닉스와 스크립트 격차를 많이 줄었다.

XPG4: <https://www.unix.com/solaris/164073-what-difference-between-xpg4-bin-usr-bin.html>

# SHELL POSIX

POSIX에서 명시한 echo, pwd, cd와 같은 빌트인 명령어는 불행하게도 UNIX System V, BSD, 그리고 **GNU**기반에 따라서 각기 다르게 동작한다. 예를 들어서 GNU echo명령어는 표준은 UNIX System V 혹은 BSD와 다르게 동작한다. (조금씩 결과가 다르게 나온다)

그러한 이유로, "echo"명령어 대신 호환성을 높이기 위해서 "printf"명령어 사용하는 것을 권장한다. 대다수 주요 셸은 POSIX.1-2017를 지원 및 제공하고 있다. 지원하지 않는 경우 셸 스크립트 호환성을 매우 낮아진다.

**POSIX.1-2017:** <https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/>

# BASH 무엇인가?

셸은 사용자가 커널 및 사용자 프로그램 사이에서 서로 대화 할 수 있도록 해주는 도구. 가끔, ssh와 셸을 혼동하는 경우가 있지만, 셸은 SSH와 같은 도구가 아님. 컴퓨터와 대화하기 위한 도구.

What is the shell: <https://www.ibm.com/docs/en/aix/7.2?topic=administration-operating-system-shells>

# BASH 무엇인가?

셸은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == SHELL라는 오해를 가지고 있음.

**PuTTY/iTerm/Xterm**와 같은 터미널 도구가 아님. 터미널에 접근하기 위한 에뮬레이터 도구임.

Bash는 일반적으로 사용하는 CLI 도구와는 다름. BASH는 COMMAND LINE INTERFACE와 비슷하기는 하지만, 대화형 형태로 사용한다. 그래서, BASH셸에서 사용할 명령어의 이름을 넣어주면, BASH는 사용자가 입력한 이름의 바이너리 파일을 찾아서 실행한다.

BASH사이트 다음과 같이 BASH를 설명하고 있다.

<https://www.gnu.org/software/bash/manual/bash.txt>

# THE TTY

TTY는 "Teletypewriter"의 약자. 초기 유닉스 시스템은 로컬만 지원하였으며, BSD의 유닉스 프로젝트를 통해서 TCP/IP를 도입 후, 원격에서 시스템 접근이 필요.

"A Teleprinter just needed a single operator to easily convey a message. While it did not have a modern-layout keyboard, its system was later evolved by Donald Murray in 1901 to include a typewriter-like keyboard."

사진에 있는 시스템은, 과거 텔레타이프 시스템이며, 현재 유닉스 및 리눅스에서 사용하고 있는 시스템도 위와 동일한 개념을 채용하고 있다. 이를 통해서 원격에서 리모트에 있는 시스템에 타이핑이 가능해짐.



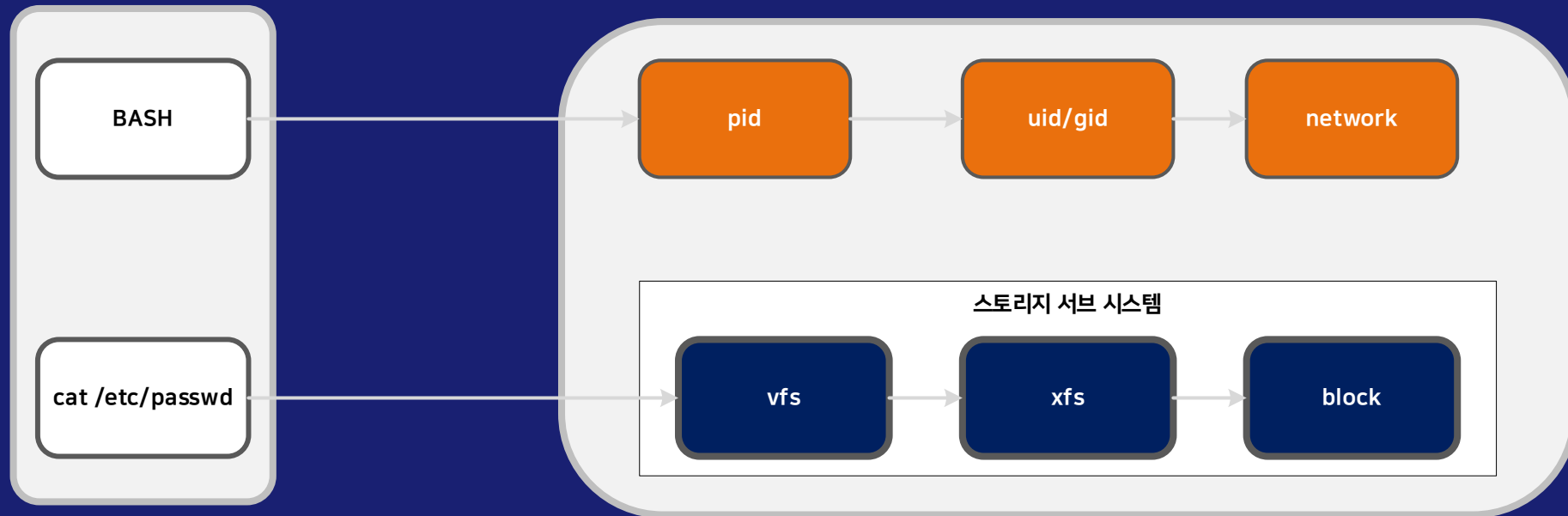
# BASH 무엇인가?

거의 대다수의 리눅스는 bash사용 및 설치되어 있음. GNU/LINUX에서는 bash가 표준 셸이라고 생각해도 됨.

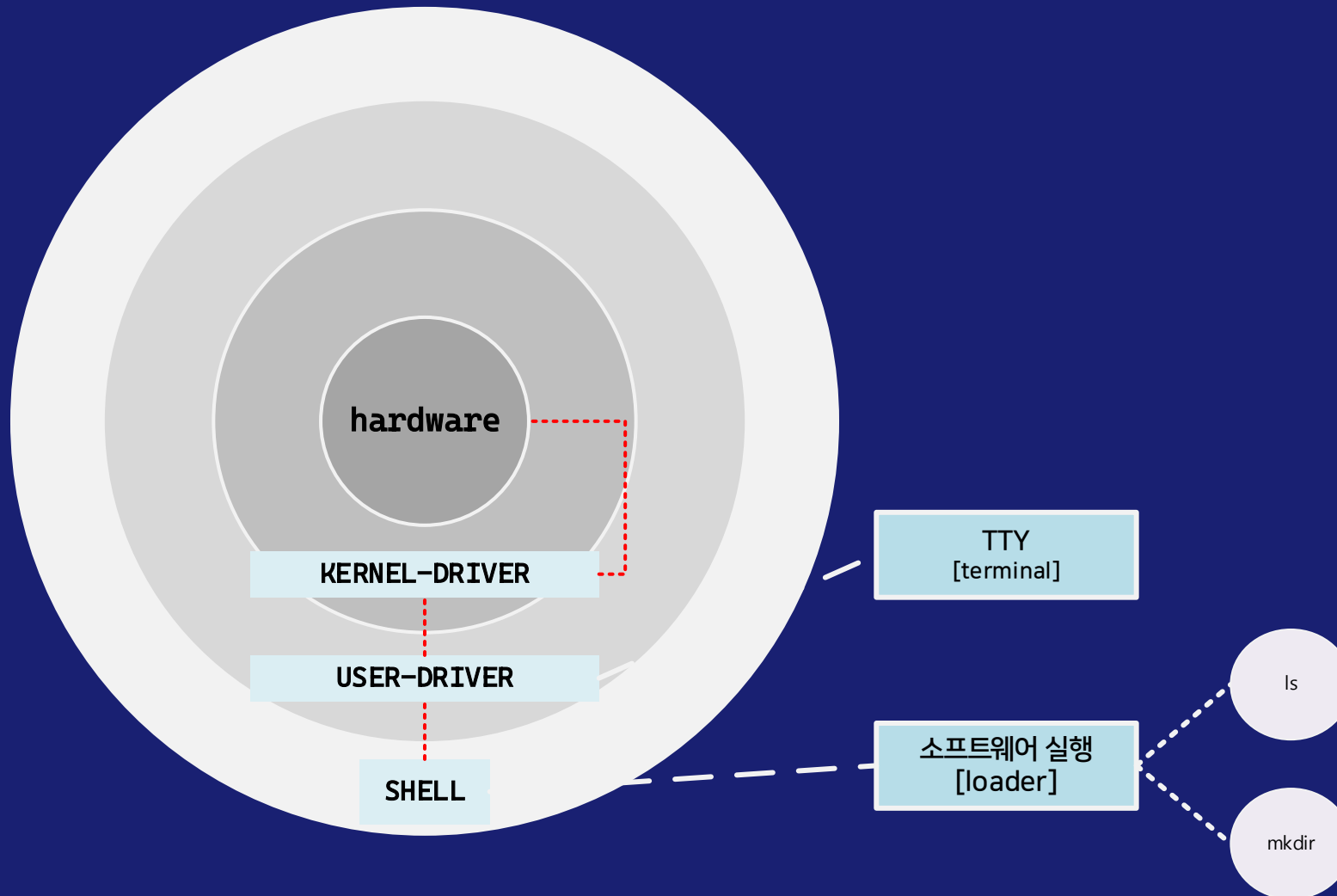
예를 들어서 **FreeBSD/NetBSD/OpenBSD**, 그 외 다른 유닉스 들에서는 bash가 아닌 csh, tsh를 사용하기도 한다. 하지만 bash사용이 가능하며, 기존에 사용하였던 csh, tsh에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

<https://www.gnu.org/software/bash/manual/bash.txt>

# BASH 무엇인가?

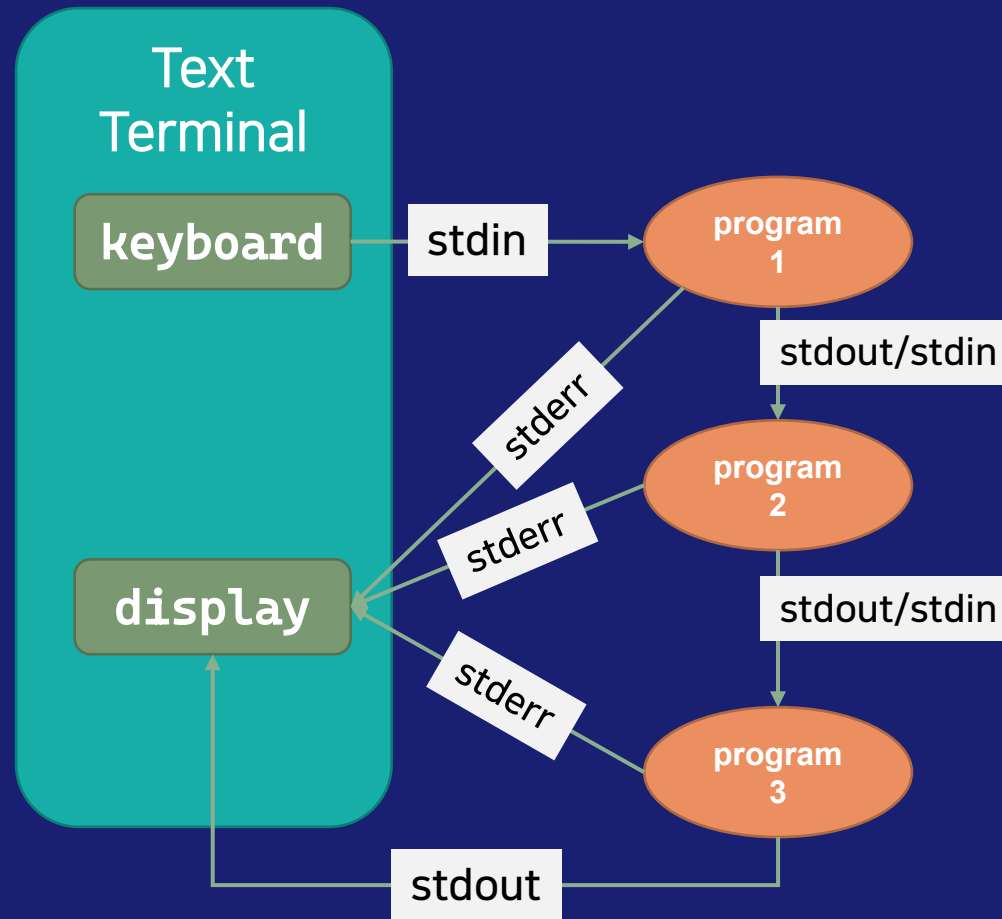


# BASH 무엇인가?

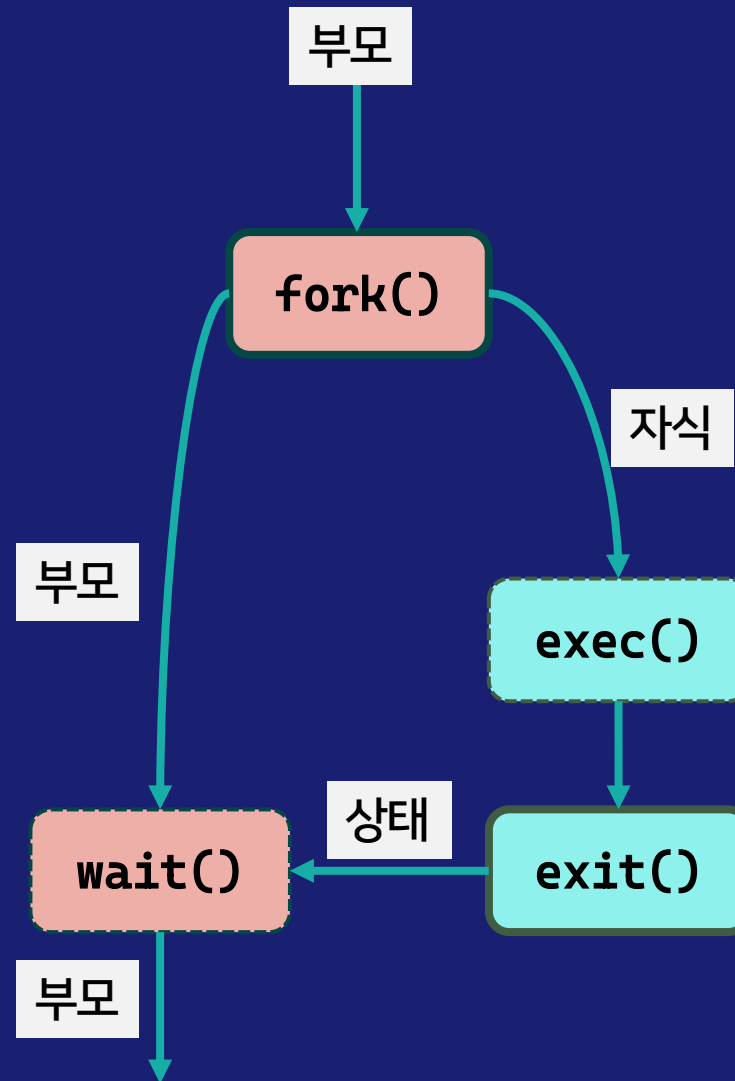




# BASH 무엇인가?

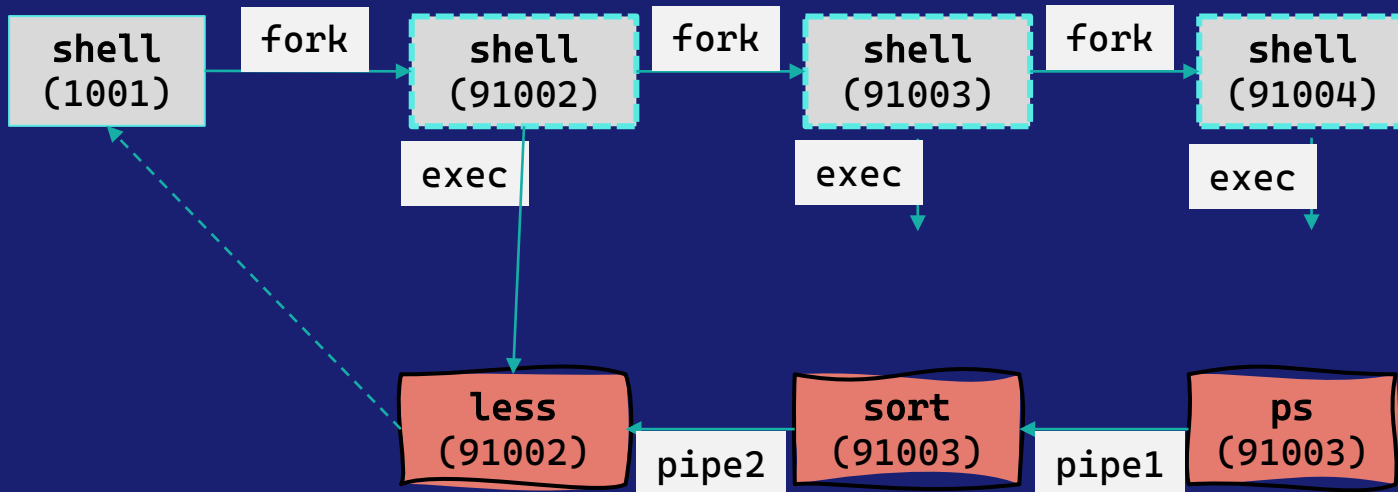


# BASH 무엇인가?



# BASH 무엇인가?

```
$ ps | sort | less
```



# 셸 기초

사용중인 리눅스에서 Bash 버전 확인하는 방법은 다음과 같다.

```
bash$ declare -p BASH  
BASH="/usr/local/bin/bash"
```

chsh에서 bash로 변경하는 경우 아래와 같이 변경이 가능하다. 모든 리눅스 배포판에서 사용가능한 셸 목록 및 변경은 아래와 같이 가능하다.

```
$ chsh -s /usr/local/bin/bash  
$ chsh -l  
$ cat /etc/shells
```

# 셸 기초

리눅스 시스템에서 사용하는 셸 목록은 아래와 같이 확인이 가능하다.

```
$ chsh -l
```

위의 명령어는 최소 설치한 경우, 일반적으로 설치가 안되어 있다. 아래 명령어로 설치가 가능하다. 다른 셸 설치를 원하는 경우, 아래 명령어로 추가적으로 패키지 검색이 가능하다.

```
# dnf install chsh -y
# dnf install epel-release
# dnf search [a-z][a-z]sh
# dnf search [a-z]sh
# dnf install zsh
# dnf install fish
```

# 프로토 타입 언어 도구

## 프로토 타입

bash기반으로 간단하게 스크립트 프로그래밍 가능하다. bash의 목표 중 하나가 "**hack together**"

## 대화형 관리 시스템

또한, BASH는 "**borne-style**" 셸 스타일로 리눅스 및 유닉스 에서 손쉽게 명령어를 사용할 수 있도록 해준다. 이를 통해서 리눅스 명령어 실행 그리고 시스템 관리가 가능하다. 기본적으로 대화형 범주 안에는 셸 스크립트 및 인터프리터가 포함이 된다.

# 셸 기능

배시에서 제공하는 셸 기능은 아래와 같다. 간단하게 기능을 정리하면 다음과 같다.

- **반복문:** 셸 명령어는 배시 반복문과 호환이 되며, 인라인 형태로 사용이 가능하다.
- **조건 문법:** 셸 명령어는 배시 조건문과 호환이 되며, 이라인 형태로 사용이 가능하다.
- **셸 함수:** 명령어를 함수 이름으로 묶어서 사용이 가능하다.
- **본 셸 내부 명령어:** 빌트인(내부 명령어)를 셸 에서 활용 및 사용이 가능하다.
- **본 셸 변수:** 셸의 변수를 명령어나 혹은 셸 내부에서 사용이 가능하다.
- **본 셸 추가 기능:** 본 셸에서는 추가 기능은 셸에서 사용하던 방식으로 그대로 사용이 가능하다. 이 부분은 배시 버전에서 달라진다.

[https://www.math.utah.edu/docs/info/features\\_1.html](https://www.math.utah.edu/docs/info/features_1.html)

# 오토메이션

자주 사용하는 명령어를 스크립트 기반으로 작성하여 간단하게 실행 및 수행이 가능한 방법. 이를 통해서 오토메이션 도구로 사용이 가능. 아래 내용에서 배시 셸 스크립트의 오토메이션에 대해서 더 자세한 내용 확인이 가능하다.

<https://opensource.com/article/19/12/automation-bash-scripts>

- 셸 스크립트 기반으로 자동화가 가능하다.
- C/C++언어처럼 사용이 가능하다. 적절하게 관리자에서 좋은 도구.
- 요구사항에 대해서 셸 스크립트로 작성하기 혹은 프로토타입 생성.
- CLI를 통해서 간단한 프로그램 작성이 가능.
- UID, GID를 통해서 스크립트 실행 권한 및 범위 제한.

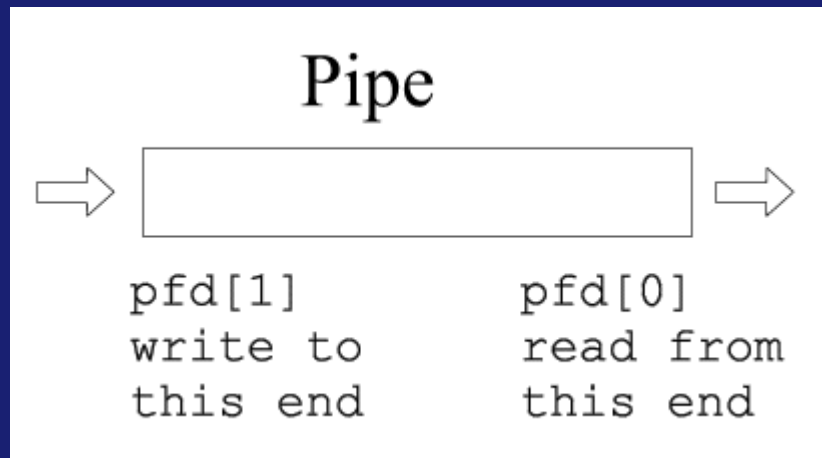


# 프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 한다. 파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다. 이를 보통 **파이핑(Piping)**라고 한다.

**필터링 그리고 입력 변환**, 어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다.

이러는 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다. 예를 들어서 '**awk**', '**sed**'같은 프로그램 들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.



# 베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다. 이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장 가능하다. 이 기능을 **와일드카드(wildcard)**라고 부른다.

## 기본 셸 와일드 카드

- Star or Asterisk (\*)
- Question mark (?)
- Square brackets ([ ])

# BASH 이동성

bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다. 앞서 이야기 하였지만, 배시는 POSIX 표준 사양을 따르며, 대다수 리눅스 및 유닉스 시스템, 심지어 윈도우 파워 셸도 POSIX 표준 사양을 따른다.

다만, 윈도우는 완벽하게 표준을 따르는 게 아니라, 호환 목적으로 부분적으로 따르고 있다. 참고로 윈도우 파워 셸은 IEEE POSIX 1003.2를 따르고 있다.

# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 변수	x=3	set x=3	echo \$x	3	3	
	y="\$x 4"	set y="\$x 4"	echo \$y	3 4	3 4	
환경변수	export z=5	setenv z 5	echo \$z	5	5	'setenv z = 5'
	export q="\$z 6"	setenv q "\$z 6"	echo \$q	5 6	5 6	

	BASH	TCSH	출력	BASH	TCSH	비고
셸 종류별 변수 선언	z=7	set z = 7	echo \$z; env grep z	7 z=7	7 z=5	tcsh는 다른 환경에서 같은 이름으로 다른 값을 가질 수 있다.

# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 경로 변수	export PATH =/a:/b	set path=(/ a /b)	echo \$path; echo \$PATH ; env grep pa th	- /a:/b -	/a /b /a:/b -	TCHS: path   PATH ?
별칭(별명)	alias ls="ls - l"	alias ls "ls -l "	ls	(same as ls -l)	(same as ls -l)	
명령 프롬프트	PS1=abc-	set prompt= abc-	[ENTER]	abc-	abc-	
재출력	prog > ofile 2> efile	(prog > ofile ) >& efile	[ENTER]	(stdout data in ofile; stde rr data in efi le)	(stdout data in ofile; stde rr data in efi le)	BASH는 재출력 파일 사용 가능

# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 로그인 시 사용하는 환경 파일	/etc/profile ~/.bash_profile ~/.bash_login ~/.profile	/etc/csh.cshrc /etc/csh.login ~/.tcshrc ~/.cshrc ~/.login				
셸 시작파일	~/.bashrc	/etc/csh.cshrc ~/.tcshrc ~/.cshrc				
로그아웃 파일	~/.bash_logout	/etc/csh.logout ~/.logout				
스크립팅						bash혹은 sh스크립트가 제일 사용하기 편함.

# 도움말

bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다.

```
bash$ help printf
```

혹은 명령어 타입확인이 필요한 경우, 다음과 같은 명령어로 확인이 가능하다.

```
bash$ type echo
```

# 연습문제

간단하게 bash에서 다음과 같은 동작을 수행해서 어떠한 결과가 나오는지 확인한다.

1. 'cp', 'ls', 'type', '[' 명령어가 내부 명령어인지 외부 명령어인지 'type'명령어로 확인한다.
2. 간단하게 "pipe"를 사용하여 'ps'명령어에서 현재 사용자가 사용중인 프로세스 목록을 출력한다.
3. 쉘 목록을 화면에 출력한다.
4. 쉘을 bash에서 fish -> zsh -> csh으로 변경한다.
5. 현재 사용중인 배포판에서 어떠한 쉘 패키지를 제공하는지 확인한다.
6. export, env명령어를 사용하여 시스템 변수를 확인한다.



# 셸 명령어

기본 명령어

# 셸 명령어

셸 명령어의 범주는 일반적으로 다음처럼 구별한다.

1. 셸 내부에서 제공하는 명령어
2. 셸 확장 명령어
3. 일반적인 관리 명령어

일반적인 관리 명령어는 셸 관리 명령어에 포함이 되지 않는다. 이 교육에서는 외부 명령어를 사용하지만, 셸 스크립팅 및 명령어 활용을 위해서 사용한다.

# 간단한 명령어

bash에서 명령어 사용 방법은 다음과 같다.

```
$ mkdir -p testdir/subdir
```

매우 간단하다! ☺

리눅스에서 사용하는 명령어는 두 가지 형태로 구성이 된다. 쉘에서 제공하는 **빌트인(built-in)명령어**, 시스템에서 제공하는 **시스템 명령어**, 굳이 따지면 사용자가 사용하는 **사용자 명령어**로 나누어 진다.

현재 리눅스 쉘 명령어는 LSB(Linux Standard Base)에 따라서, **"/bin/" ,"/sbin/"** 디렉터리 구별을 하지 않는다. 하지만, 여전히 **배포판에 따라서 조금씩 다르게 구성**이 되어있다.

참고로, **mkdir**명령어는 외부 명령어이다.

# 간단한 명령어

셸에서 입력된 문자열은 기본적으로 명령어로 인식이 된다.(몇몇 특수문자 제외) 특수 문자가 포함된 파일이나 혹은 실행파일(일반적으로 없겠지만)은 아래와 같은 명령어로 실행이 가능하다.

```
$ touch test file \@ \@world
$ ls -l
test
file
```

여기에 좀 더 복잡하게 특수문자를 사용하면 다음처럼 결과가 나온다.

# 예약 명령어

배시는 예약된 키워드 명령어가 있다. 해당 명령어는 다른 용도로 사용이 불가능하다.

if	then	elif	else	fi	time
for	in	until	while	do	done
case	esac	coproc	select	function	
{	}	[[	]]	!	

# 예약/내부/외부 명령어

리눅스 셸은 기본적으로 다음과 같은 형태로 명령어를 구별한다.

- 키워드 명령어
- 내부 명령어
- 외부 명령어
  1. 시스템 명령어
  2. 사용자 명령어

외부 명령어는 우리가 흔히 사용하는 ls, mkdir, ip와 같은 명령어이다. 이 명령어는 셸에서 제공하는 명령어가 아닌, 외부 도구를 통해서 제공되는 명령어이다.

# 내부 및 외부 명령어

현재 대다수 리눅스 배포판은 배시 기반으로 쉘 구성이 되어 있다. 이러한 이유로 모든 리눅스에서 기본 내장형 명령어는 호환이 된다. 하지만, 가끔 배포판끼리 "해당 명령어 존재하지 않음"이라는 메시지가 보인다.

이 이유는 내장 혹은 외장 명령어는 서로 호환이 되지만, 명령어가 존재하는 위치가 달라서 발생하는 경우다. 현재, 대다수 리눅스는 LSB기반으로 FHS를 구성하기 때문에(특히, 기업용) 이와 같은 문제는 이전에 비해서 많이 발생하지 않는다.

명령어 위치는 '**which**', '**where**'명령어를 통해서 확인이 가능하다. 또한, 몇몇 내부 명령어는 호환성을 위해서 외부 명령어로 구성된 경우가 있다. 예를 들어서 '**echo**', '**printf**'와 같은 명령어가 대표적이다.

```
# which echo
```

```
# where echo
```

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<code>:(a <u>colon</u>)</code>	네	네	종료 시 숫자 0으로 반환 합니다.
<code>.(a <u>period</u>)</code>	네	네	sourc명령어와 동일한 기능. ". <FILE>"로 표시가 되어 있으면 해당 파일을 셸 변수로 적용.
<code>[</code>	네		test명령어와 동일. 이를 통해서 참/거짓을 확인한다.
<code>alias</code>			단축 명령어 목록이나 혹은 선언.



# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<b>bg</b>			대기중인(suspend) 특정 프로세스를 백그라운드로 동작한다.
<u><b>bind</b></u>			화면에 바인딩(binding)된 키를 출력한다.
<b>break</b>	네	네	for, while같은 루프문을 종료한다.
<b>builtin</b>			셸 빌트인 명령어. 호환이 되는 셸 혹은 배시에서는 공통으로 사용이 가능하다.
<b>caller</b>			셸 스크립트 디버깅시, 많이 사용하는 명령어. 루틴 추적 및 실행 시 반환 값 확인 할 때 많이 사용한다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<b>cd</b>	네		작업 디렉토리를 변경합니다.
<b>command</b>			동일한 이름의 bash 함수가 있더라도 외부 또는 내장 명령을 실행합니다.
<b>compgen</b>			지정된 명령에 대한 자동 완성 일치를 생성합니다.
<b>complete</b>			인수를 자동 완성하는 방법을 지정합니다.
<b>compgpt</b>			자동 완성 옵션을 수정하거나 표시합니다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<b>continue</b>	네	네	for, while, until 또는 select 루프에서 다음 또는 다른 반복으로 건너뛵니다.
<u><b>declare</b></u>			조판과 동일합니다. 변수에 속성을 선언, 표시 또는 할당합니다.
<b>dirs</b>			기억된 디렉토리를 포함하는 셸의 디렉토리 스택을 표시합니다.
<b>disown</b>			현재 셸에서 지정된 작업을 제거합니다.
<b>echo</b>			표준 출력에 문자열을 씁니다.
<b>enable</b>			내장 셸 명령을 활성화 또는 비활성화합니다. 시스템의 다른 명령이 내장 명령 이름을 공유할 때 유용합니다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<u>eval</u>	네	네	인수를 평가하고 명령어로 실행합니다.
<u>exec</u>	네	네	새 프로세스를 만들지 않고 현재 셸을 대체하는 명령을 실행합니다.
<b>exit</b>	네	네	선택적 숫자 종료 상태로 셸을 종료합니다.
<u>export</u>	네	네	셸 변수 또는 함수가 하위 프로세스에 전달되어야 함을 지정하십시오.
<b>fc</b>			기록 목록에서 다양한 명령을 편집하거나 실행합니다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<b>fg</b>			백그라운드에서 동작중인 프로세스를 포그라운드로 전환한다.
<u>getopts</u>	네		셸 스크립트의 인자 값을 불러온다.
<u>hash</u>	네		history명령어처럼, 어떤 위치에서 명령어가 실행이 되었는지 출력한다.
<b>help</b>			내부 명령어에 대한 도움말 출력.
<b>history</b>			셸에서 사용한 명령어 목록 출력한다.
<b>jobs</b>			현재 동작중인 작업(프로세스) 목록을 출력한다.
<u>kill</u>			프로세스에 제어신호를 보낸다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<u>let</u>			셸에서 산술 연산을 처리 합니다.
local			지역 변수를 선언 시 사용합니다.
logout			셸에서 종료 합니다.
<u>mapfile</u>			파일이나 혹은 표준 출력에서 불러온 내용을 배열로 생성한다.
popd			셸 스택에서 이동한 디렉터리 순서대로 제거하면서 변경 합니다.
printf			화면에 표준 출력으로 문자열을 출력 합니다. echo와 비슷하지만, 포매팅 기능이 있습니다.
pushd			셸의 디렉터리 스택에 디렉터리를 추가 합니다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
pwd	네		현재 작업중인 디렉터리 위치 확인.
<u>read</u>			파일이나 혹은 표준 출력에서 입력을 받아서 단어 단위로 나눈 후, 변수로 저장한다.
readarray			mapfile과 동일: 파일 또는 stdin에서 행을 읽고 각 행을 배열 요소에 할당합니다.
readonly	네	네	함수 또는 변수를 읽기 전용으로 지정하여 변경할 수 없도록 합니다.
return	네		bash 함수를 종료하고 값을 반환합니다.
set	네	네	현재 셸의 옵션을 수정하여 동작을 변경합니다. 쉘 함수 및 변수의 이름과 값을 표시합니다.

# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<u>shift</u>	네	네	현재 명령의 위치 매개변수를 왼쪽으로 이동합니다.
<u>shopt</u>			특정 bash 셸 옵션을 수정합니다.
<u>source</u>			.(마침표)와 동일합니다. 파일에서 명령을 읽고 실행합니다.
<u>test</u>	네		파일 속성을 테스트하고 숫자 값을 비교합니다.
<b>times</b>	네		셸 프로세스의 사용자 및 시스템 CPU 시간을 표시합니다.
<b>trap</b>	네	네	셸이 특정 신호를 수신할 때 실행할 신호 트랩을 지정합니다.
<b>type</b>			셸이 이름을 해석하는 방법을 결정하십시오: 별칭, 함수, 내장, 파일 또는 키워드.



# 빌트인 명령어

빌트인 명령어	SH	POSIX	설명
<b>typeset</b>			선언과 동일합니다. 속성을 변수에 선언, 표시 또는 할당합니다.
<b>ulimit</b>			셸에서 시작한 프로세스에 사용할 수 있는 리소스에 제한을 할당합니다.
<b>umask</b>	네		셸에서 생성된 파일에 대한 파일 생성 마스크를 설정합니다.
<b>unalias</b>			별칭을 제거합니다.
<b>unset</b>	네	네	변수 또는 함수 이름을 제거하십시오.
<b>wait</b>	네		프로세스가 종료될 때까지 기다린 다음 반환 값을 표시합니다.

# 명령어 사용

내부 및 외부 명령어 구별 및 사용 방법은 다음과 같다.

```
$ touch $$$likeMoney.txt
$ ls
12234likeMoney.txt
```

키워드/내부/외부 명령어 구별 방법은 다음과 같다.

```
$ type touch
$ type [
$ type trap
```

# 혼돈되는 문자열 명령어

아래 명령어를 실행을 하면 올바르게 실행이 되지 않는다.

```
$ touch report<SRV1>.txt  
$ touch report;Final.txt  
$ touch He$$lo.txt
```

# 쿼트

이러한 문제를 해결 하기 위해서 쿼트(Quoting/Quote)을 사용해야 한다. 보통 다음과 같은 방법이 있다. 쉘에서는 쿼팅 방법에 따라서 문자열 혹은 변수로 취급한다.

```
'<string>'  
"<string>"  
\<escape>\<escape>
```

# ECHO

아래와 같이 쿼티를 테스트 해본다.

```
$ echo "hello world!"  
$ echo -e "hello world!"  
$ echo 'test "this is ECHO"'  
$ echo $HOSTNAME  
$ a=`echo "HELLO" | tr A-Z a-z`  
$ echo $a
```

# PRINTF

echo명령어로 문자열 출력이 가능하지만, 각기 다른 셸에서 echo명령어를 다루는 방법 및 렌더링이 조금씩 다르다. 이러한 이유로 printf명령어로 문자열 출력을 권장한다.

```
#!/bin/bash
printf "Enter your name:\n"
read name
printf "Enter your surname:\n"
read surname
printf 'Welcome: %s\n' "$name $surname"
```

# LET

'let'은 특정 변수 및 배열에서 값을 추출하여 연산을 수행한다. 사칙연산에서 보통 많이 사용한다. 루프 명령어는 없지만, PID크기만큼 작업을 반복적으로 수행한다.

```
#!/bin/bash
# spawn.sh
PIDS=$(pidof sh $0)
P_array=( $PIDS )
echo $PIDS
let "instances = ${#P_array[*]} - 1"
echo "$instances instance(s) of this script running."
echo "[Hit Ctl-C to exit.>"; echo
sleep 1
sh $0
exit 0
```

# EVAL

외부 명령어를 실행하여, 결과를 변수로 읽어올 때 많이 사용한다. 'eval'는 POSIX를 지원하지만, 보안상 이유로 사용을 최소화 한다. 이유를 최소화 하는 이유는 아래와 같다.

```
a='$b'
b='$c'
c=d

echo $a          # $b
                  # First level.
eval echo $a      # $c
                  # Second level.
eval eval echo $a # d
                  # Third level.
```



# 변수 확인

변수를 확인하는 방법은 아래와 같다.

## **printenv**

시스템에서 사용하는 변수목록을 화면에 출력한다. 함수는 제외하고 선언한 변수들만 화면에 출력이 된다.

```
$ printenv OLDPWD  
/tmp
```

## **env**

'printenv'처럼 시스템 변수 목록을 출력이 되며, 간단하게 변수의 값 변경도 가능하다.

```
$ env -i OLDPWD=/usr  
OLDPWD=/usr
```

# 변수 확인

## compgen

배시에서 제공하는 빌트인 명령어 혹은 옵션에 대해서 확인이 가능한 명령어.

- *-a*: aliases
- *-b*: builtins
- *-c*: all commands
- *-d*: directories
- *-e*: exported shell variables

# 변수 확인

- `-f`: files
- `-g`: groups
- `-j`: jobs
- `-k`: shell reserved words
- `-s`: services
- `-u`: user names
- `-v`: variables

# 변수 확인

아래 명령어는 다음과 같은 내용을 화면에 출력한다.

1. freedesktop.org에서 구현 및 배포되고 있는 라이브러리 확인
2. 배시에서 자동 완성으로 사용이 가능한 폰트 이름
3. 배시에서 자동 완성으로 사용이 가능한 디렉터리 이름

```
$ compgen -v XDG
$ compgen -f -G *.otf
$ compgen -d -P dir-
```

## 추가 설명



# 연습문제

시스템 변수 및 시스템 명령어를 사용하여 다음과 같은 내용을 출력한다.

1. 호스트 이름을 출력한다. 단, 호스트 이름은 셸 변수로 출력하다.
2. 셸 변수 목록 중에 **OLDPWD**를 찾아서 해당 내용을 출력한다.
3. 현재 사용자의 홈 정보를 가지고 있는 변수를 출력한다.
4. 사용중인 셸 프로그램 이름을 화면에 출력한다.
5. "This is <the linux service>"라는 문자열을 echo를 통해서 출력한다.
6. 같은 명령어가 몇 번 실행이 되었는지 확인한다.

# 배시

특별변수

핸들링

# 특수문자 해결 방법

가끔 특수문자를 셸에서 다루거나 혹은 직접 생성하는 경우가 있다. 아래와 같이 간단하게 테스트 한다.

```
$ touch report\<SRV1\>.txt  
$ touch report\;Final.txt  
$ touch \$\$\$likeMoney.txt
```



# Escape Reference

탈출문자열에 대한 설명은 다음과 같다.

A non-quoted backslash ‘\’ is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of newline. If a \newline pair appears, and the backslash itself is not quoted, the \newline is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

# 탈출문자

그냥 두 번 입력하면 된다. 첫 번째는 탈출문자(metacharacter), 두 번째는 문자로써 입력. 일반적으로 모든 셸에서 공통으로 사용이 가능하다.

```
$ echo \\backslash\\  
\\backslash\\
```

# 연속출력

연속출력을 하기 위해서 다음과 같이 백 슬래시 활용 및 사용이 가능하다. 혹은 다음과 같은 방법으로 연속 문자열을 생성이 가능하다.

```
$ echo backslash\  
> Hello  
> Slash  
backslashhelloygl  
$ cat <<EOF>> longest.txt  
> hello  
> this is  
> good world  
> EOF
```

# 특수문자

특수문자 중, 달라표시가 된 문자들은 어떠한 쿼티를 사용하느냐 따라서 출력결과 동작이 달라진다.

```
$ touch 'report document'  
$ touch 'report<SRV1>.txt'  
$ touch 'report;Final.txt'  
$ touch '$$$report.txt'
```

# 싱글 쿼트 줄 내림(Single Quote)

```
$ echo 'quotes
```

```
> foo
```

```
> bar'
```

```
quotes
```

```
foo
```

```
bar
```

# 싱글쿼트

아래 중 어떤 내용이 출력 되는지 확인한다.

```
$ echo 'it\'s today'  
$ echo 'it''s today'  
$ echo 'it'\''s today'
```

## 더블쿼트/백틱

```
$ echo "This is my shell: $SHELL"
```

```
$ echo 'This is my shell: $SHELL'
```

```
$ echo "The variable: \ $total"
```

```
$ echo "Back\\to\\back\\back\\slashes\\_world"
```

```
$ echo "hello $USER"'!!!'
```

```
$ echo "backticks: \\`\\`"
```

# 복합적 쿼트

```
$ echo Hello, \ "$USER"! Hell to the '"$HOSTNAME"'!  
$ $(echo echo hello)
```

위의 내용은 다음처럼 동작하게 된다.

첫번째 'echo'는 "echo hello"를 문자열로 출력하며, 이게 다시 콘솔(tty)에 전달하면서 'echo hello'라 명령어로 실행하면서 "hello"라는 메시지가 출력이 된다.



# 특수 변수

셸에서 지원하는 몇몇 특수변수가 있다. 이는 직접적으로 선언하지 않으며, 스크립트 작성 시 활용한다.

```
$ echo $?  
$ true ; echo $?  
$ false ; echo $?  
1: False  
0: True
```

# 특수 변수

명령어가 없는 경우 다음과 같이 메시지가 출력이 된다.

```
$ thereisnocommand ; echo $?
```

```
127
```

```
$ $_
```

```
$ $0
```

# 특수한 변수

변수 이름	설명
\$?	이전에 실행 되었던 명령어의 종료 값 출력
\$\$	셸이 사용하고 있는 PPID를 출력
\$!	백 그라운드 동작하고 있는 프로세스의 아이디를 출력
\$_	명령어 인수값이 있는 경우, 출력이 됩니다. 없는 경우에는 PID가 출력된다.
\$-	셸의 set옵션을 출력한다.

# 셸 종료 값

시그널 코드(이름)	시그널 값	효과
<b>SIGHUP</b>	1	Hangup
<b>SIGINT</b>	2	Interrupt from keyboard
<b>SIGKILL</b>	9	Kill signal
<b>SIGTERM</b>	15	Termination signal
<b>SIGSTOP</b>	17,19,23	Stop the process

# 배시 컨트롤 조합

배시 프로세스 관리	
Ctrl + A	CLI 맨 처음으로 커서를 이동 합니다.
Ctrl + E	CLI 맨 끝으로 커서를 이동 합니다.
Ctrl + F	글자 하나씩 앞으로 이동 합니다.
Ctrl + B	글자 하나씩 뒤로 이동 합니다.
Ctrl + XX	커서를 CLI의 맨 처음과 끝으로 위치를 바꾸면서 이동 합니다.
Alt + F / Esc + F	단어 단위로 이동 합니다. 이동 시 기준은 빈 공간 혹은 세미콜론이 기준이 됩니다.
Alt + B / Esc + B	반대로 단어 단위로 이동 합니다.

# 배시 컨트롤 조합

배시 프로세스 관리	
Ctrl + L	화면에 출력된 내용을 전부 지웁니다.
Ctrl + S	화면에 출력을 중지 합니다.
Ctrl + Z	현재 실행중인 프로세스를 일시적으로 백그라운드로 중지 합니다.
Ctrl + Q	일시적으로 중지된 프로세서를 재개 합니다.
Ctrl + C	프로세스에 SIGI시그널을 보내서 현재 실행중인 프로세스 중지 합니다.
Ctrl + D	현재 사용중인 터미널을 종료 합니다.

# 배시 컨트롤 조합

배시 프로세스 관리	
Ctrl + R	배시 명령어 기록을 역순으로 검색 합니다.
Alt + P	콜론이 출력이 되면, 실행하였던 명령어 번호 혹은 명령어 일부분을 입력하면 됩니다.
Ctrl + P / Up arrow	과거에 사용했던 명령어를 조회 합니다.
Ctrl + N / Down arrow	과거 기준으로 사용하였던 명령어 조회 합니다.
Ctrl + S	화면을 잠깐 중지 합니다. 취소는 Ctrl + Q를 누르면 됩니다.

# 배시 컨트롤 조합

배시 프로세스 관리	
Ctrl + O	Ctrl + R로 명령어 선택 후 실행 합니다.
Ctrl + G	Ctrl + G로 Ctrl + R를 종료 합니다.
!!	마지막에 실행한 명령어를 수행 합니다.
!*	이전에 수행한 명령어를 실행 합니다.



# 배시 컨트롤 조합

배시 프로세스 관리	
Ctrl + U	커서 뒤로 있는 모든 문자열을 제거 합니다.
Ctrl + K	커서 뒤로 있는 단어 혹은 문자를 제거 합니다.
Ctrl + W	명령어 혹은 인자 값 커서 기준으로 뒤로 제거 합니다.
Ctrl + D	한 글자씩 제거 합니다.
Ctrl + H	커서가 있는 위치에 문자를 제거 합니다.

# 배시 컨트롤 조합

배시 프로세스 관리	
Alt + D	커서 뒤로 있는 문자들을 제거 합니다.
Alt + Backspace	단어별로 삭제 합니다.
Alt + . / Esc+.	마지막에 실행한 명령어 인자 값도 같이 불러 옵니다.
Alt + <	히스토리에 맨 처음에 있는 기록을 출력 합니다.
Alt + >	마지막에 실행한 명령어를 출력한다.
Esc + T	두 단어의 위치를 커서 기준으로 변경 합니다.
Alt + T	현재 단어 위치를 앞 단어와 변경 합니다.

# 배시 컨트롤 조합

배시 프로세스 관리	
<b>TAB</b>	명령어 혹은 자원에 대해서 자동완성을 합니다.
<b>~TAB TAB</b>	현재 등록중인 모든 사용자를 출력 합니다.
<b>Ctrl + I</b>	단어로 시작된 명령어를 화면에 출력 합니다.
<b>Alt + *</b>	현재 디렉터리에서 파일 혹은 디렉터리에 맞는 값들을 화면에 출력 합니다.

# 연습문제

다음 특수 문자를 화면에 올바르게 출력해보세요.

- 백틱 이용해서 사용하면 됩니다.
- 반드시 특수문자로 출력이 되어야 합니다.
- 배시 키 바인딩을 통해서 이전에 사용한 명령어를 다시 검색해보세요.
- bash, zsh에서 키 바인딩이 동일한지 확인해보세요.
- sleep명령어를 실행 후 프로세스의 PID를 출력한다.
- 명령어 수행 후, 리턴 값을 같이 출력한다.

```
This World sponsor @Bash.  
I love to save $$USD$$ in my bank account!!!  
The sleep process ID is 23123 and 0/1
```

# 재-출력

필드 필터링

재-출력 번호

# 입출력 전환

표준 입출력 오류는 각각 숫자를 가지고 있다.

1. 출력은 "stdout(standard output)", "표준출력"이라고 말한다.
2. 오류는 "stderr(standard error)" 그리고 "표준오류"라고 부른다.
3. 3번은 사용하지 않으며, 사용자가 별도로 지정해서 활용 및 사용이 가능하다.

배시 스크립트에서 매우 일반적으로 사용한다. 다른 호칭으로 **파일 디스크립터(File Descriptor)**라고 부른다. **파일 디스크립터**는 총 3개를 가지고 있다. 실제로 사용이 가능한 디스크립터는 총 2개이다.

# Input, Output, Redirection

```
$ printf "Hello World!\n"
```

```
Hello World!
```

```
$ printf "Hello World!\n" > myfile
```

```
$ cat myfile
```

# Input, Output, Redirection

디렉터리 위치로 재 전달 혹은 출력하는 경우 다음처럼 한다.

```
$ printf '' > empty.txt
```

```
$ wc -c empty.txt
```

```
0
```

```
$ cd
```

```
$ printf 'Hello World!\n' > ~/tmp/test.txt
```

```
$ printf 'Hello World!\n' > /tmp/test.txt
```



# Input, Output, Redirection

```
$ printf 'Hello World!\n' > "$HOME"/text.txt
```

```
$ printf 'Hello, root! \n' > ~root/tmp/text.txt
```

# 이어 붙이기

기존 텍스트 내용에 추가를 하거나 혹은 이어 붙이기 하는 경우 다음처럼 명령어를 실행한다.

```
$ printf 'First Command\n' > /tmp/test_doc.txt
$ printf 'Second Command\n' >> /tmp/test_doc.txt
$ cat test_doc.txt
First Command
Second Command
```

# 이어 붙이기

스크립트 기반으로 이어 붙이기. 실제로 이러한 방식으로 사용할 필요가 없지만, 아래와 같이 구성 및 사용이 가능하다.

```
$ vi append.sh
#!/bin/bash
printf 'Starting script\n' >> log
printf 'Creating test directory\n' >> log
mkdir test || exit
printf 'Changing into test directory\n' >> log
cd test || exit
printf 'Writing current date\n' >> log
date > date || exit
```

# 이어 붙이기

"||" 만약 해당 디렉터리가 이미 생성이 되어 있는 경우 exit명령어를 실행한다. "||" 연산자는 논리적 연산자 "or"역할을 한다.

```
$ mkdir test || exit
```

# cut

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다. 'cut'명령어 경우에는 "-d"옵션을 통해서 필드 구별자를 명시한다.

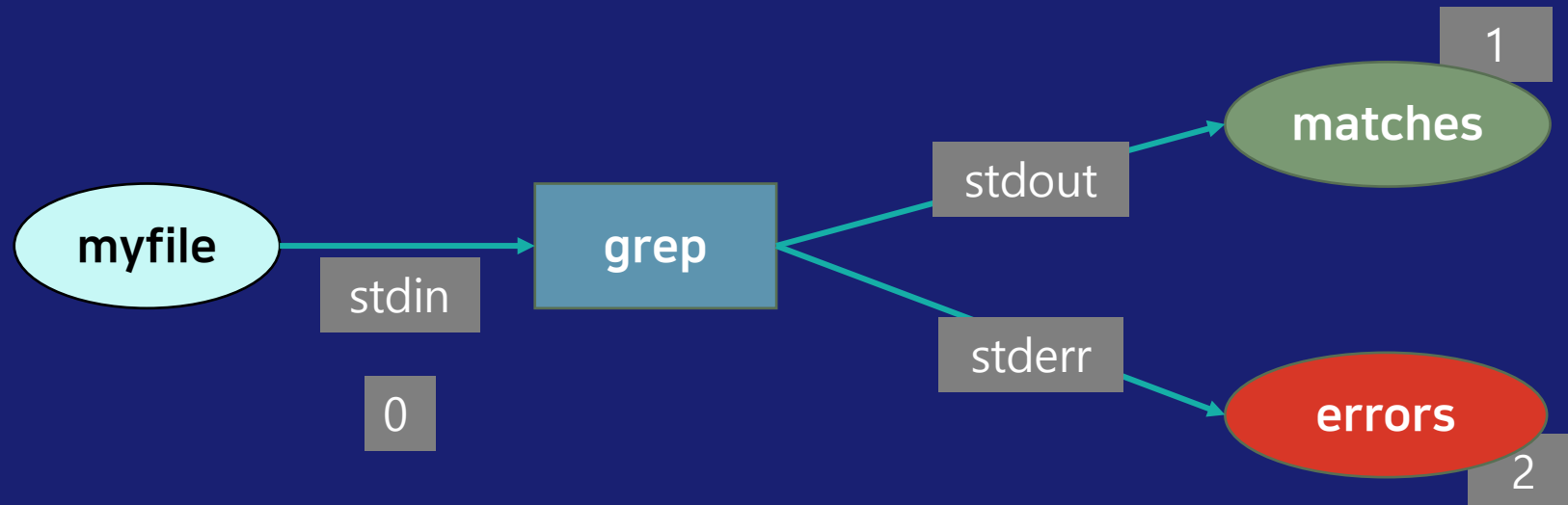
```
$ cut -d: -f1,6 /etc/passwd  
root:/root  
bin:/bin  
daemon:/sbin  
adm:/var/adm  
lp:/var/spool/lpd  
sync:/sbin
```

# awk

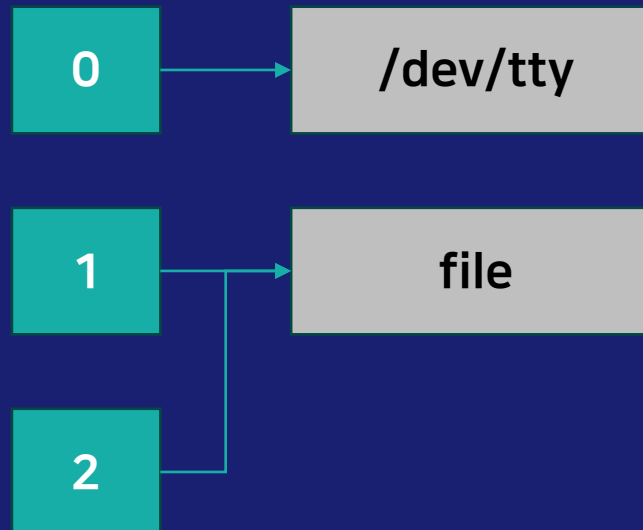
'awk'는 'cut'과 비슷하지만, 필드 출력 결과물을 제 가공이 가능하다. 아래 예제처럼 사용 및 구현이 가능하다.

```
$ awk -F: '{ print "\033[32mUSER: " $1, "\033[31mHOME: "$6, "\033[33mSHELL: "$7 }' /etc/passwd
root:/root
bin:/bin
daemon:/sbin
adm:/var/adm
lp:/var/spool/lpd
sync:/sbin
```

# 입출력 전환



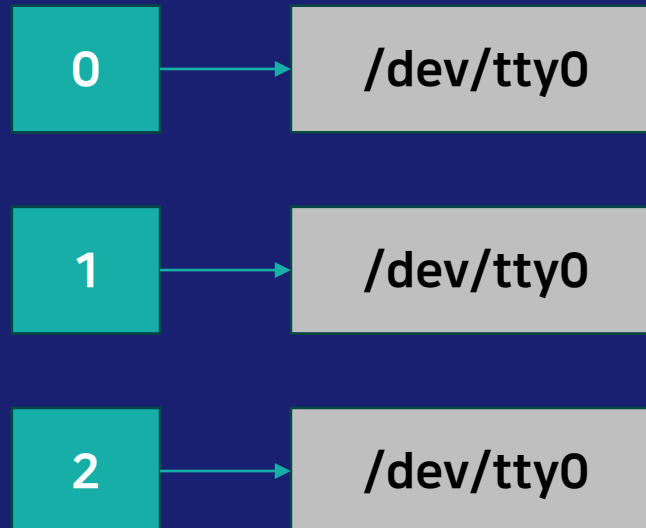
# 입출력 전환



```
$ ls >file 2>&1
```

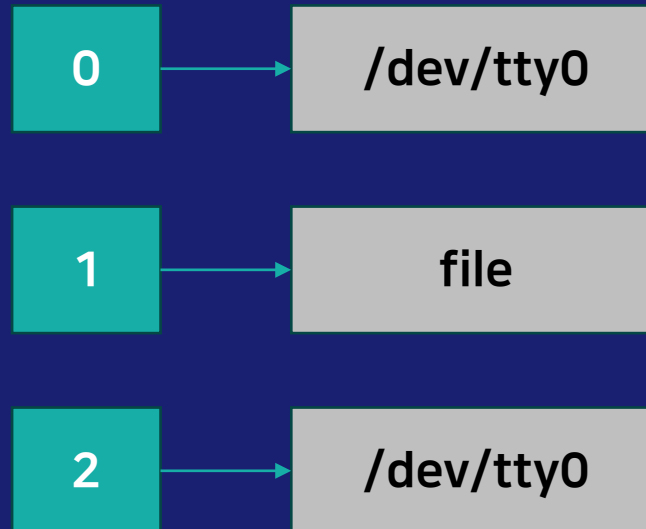


# 입출력 전환



```
$ ls
```

# 입출력 전환



```
$ ls > file > 1
```

# grep전환

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

```
$ grep pattern testfile.txt /thereisnowhere > matches 2> matches_error.txt
```

위에 처럼 사용하는 경우 현재 오류 메시지만 matches\_error.txt파일에 기록 정상적인 메시지는 matches에 저장되어 있다. 그래서 같이 출력을 하기 위해서는 **&(ampersand)**를 통해서 1에 대한 출력을 명시 해주어야 한다.

# grep전환

아래 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다. 주의 할 부분은 동일한 작업을 수행하기 위해서는 ">"앞에다가 **2>&1**을 넣을 수 없다. 리디렉션은 왼쪽에서 오른쪽으로 항상 이루어진다.

```
$ grep pattern testfile.txt /thereisnowhere > matches 2>&1
```

# /dev/null

가끔 발생한 모든 오류를 화면에 출력할 필요는 없다. 그럴 때 사용하는 유용한 장치가 있다. 바로....

```
"/dev/null"
```

위의 장치로 오류 내용을 내보내면 된다. 전달하는 방법은 매우 간단하다.

```
$ grep pattern testfile /thereisnowhere > matches 2> /dev/null
```

# comm

블록 오류를 처리하기 위해서 comm명령어를 사용하여 테스트 한다. comm명령어는 두 개 파일의 내용은 줄 별로 비교 확인한다.

```
$ comm test1.txt test2.txt
$ comm test1.txt test2.txt 2> /dev/null
$ echo $?
```

# tee

여러 파일로 내용을 재 전달 하기 위해서는 tee를 사용하여 다음과 같이 처리한다. 하지만 꼭 tee명령어를 사용할 필요는 없다.

```
$ printf ' Copy Copy rom rom\n' | tee my_copy_rom.txt  
Copy Copy rom rom
```

# tee

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

```
$ printf 'Copy Copy rom rom\n' | tee myfile1 myfile2 myfile3
```

```
Copy Copy rom rom
```

```
$ ls
```

```
myfile1  myfile2  myfile3
```



# tr

리눅스 명령어 중에 입력 재전달의 대표적인 명령어가 있다. tr명령어는 입력 받은 내용을 대문자 혹은 소문자로 다시 출력한다.

```
$ tr a-z A-Z  
Hello, world!  
HELLO, WORLD!
```

# tr

위의 명령어를 가지고 재 전달 프로세스를 다루면 다음처럼 다룰 수 있다.

```
$ cat mysample | tr a-z A-Z
```

혹은 미리 작성된 파일을 대소문자로 변경하는 경우에는 다음과 같은 명령어로 실행이 가능하다.

```
$ tr a-z A-Z < mysample
```

# tr

재 전달 오퍼레이터를 손쉽게 다음처럼 사용이 가능하다. 출력을 위한 cat같은 명령어를 사용할 필요가 없다.

```
$ cat /etc/passwd | tr a-z A-Z
```

혹은 아래처럼 복잡하게 입출력을 제어할 수 있다. 아래 명령어는 표준오류만 출력하는 예제이다.

```
$ tr a-z A-Z < mylines > mylines.capitalized 2> mylines.error
```

# 텍스트 파일 합치기

여러 개의 텍스트 파일을 하나로 통합할 때 다음처럼 스트리밍 처리한다.

```
$ cat testfile1 testfile2 testfile3 > testfiles.combined  
$ date > output  
$ hostname >> output  
$ cat output
```

# 변수 변경 및 중괄호

그룹 명령어는 중괄호를 사용하여 처리한다. 아래처럼 실행하면 모든 출력이 대문자로 출력이 된다. 또한 변수로 처리가 가능하다.

```
$ { LANG=C date ; hostname ; } | tr a-z A-Z  
WED MAR 20 23:43:35 KST 2024  
VMW.EXAMPLE.COM
```

위의 명령어는 아래처럼 사용이 가능하다.

```
$ (date; hostname)  
2024. 03. 20. (수) 23:44:01 KST  
vmw.example.com  
$ showdate=$(date; hostname)  
$ echo $showdate  
2024. 03. 20. (수) 23:44:17 KST vmw.example.com
```

# 파일 출력 및 입력

```
$ { date ; printf '%s\n' apple carrot banana ; } > testwords
```

```
Mon Sep 19 17:17:19 KST 2022
```

```
'applen' 'carrotn' 'bananan'
```

```
$ { read date ; printf '%s\n' "$date" ; sort ; } < testwords
```

```
Mon Sep 19 17:17:19 KST 2022
```

```
'applen' 'carrotn' 'bananan'
```

# sed

에디터가 아닌 표준 입력을 통해서 편집하는 방법. 보통 스크립트 처리시 아래와 같은 방법으로 작업한다. 자주 사용하는 방법은 아니지만, 가끔 스크립트 작성 시 사용한다.

```
$ cat /usr/share/doc/lshw/README.md
$ sed '' /usr/share/doc/lshw/README.md | head -2
```

```
lshw: HardWare LiSter for Linux
=====
```

```
$ sed '1,2d' /usr/share/doc/lshw/README.md | head -2
```

lshw is a small tool to provide detailed information on the hardware configuration of the machine. It can report exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed, etc. on DMI-capable x86 or EFI (IA-64) systems and on some ARM and PowerPC machines (PowerMac G4 is known to work).

# sed

간단하게 편집을 테스트 해보려면 다음과 같이 명령어를 실행한다. 첫번째는 "CONFIG\_SYSFS\_DEPRECATED" 문자열이 있으면 제거한다.

```
$ sed -i '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/systemd/README
```

두 번째는, 제거 후 해당 문자열이 있는지 명령어로 확인한다.

```
$ sed -i '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/systemd/README | grep  
CONFIG_SYSFS_DEPRECATED
```

마지막 세번째는 "found"를 "not\_found"로 바꾼다.

```
$ sed -i 's/found/not_found/' /usr/share/doc/systemd/README
```



# 연습문제

화면에 출력된 내용을 텍스트 파일로 저장한다.

- `/usr/share/doc/`에 있는 디렉터리 내용을 출력해서 `result-doc.txt`라고 저장한다.
  - `ls` 명령어로 `doc/` 밑의 모든 내용을 출력 후, `docs-file-list.txt`라는 파일로 저장한다.
  - 표준 출력/오류 상관 없이 `docs-file-list.txt`에 저장한다.
- `'sed'` 명령어를 사용해서 다음과 같이 내용을 변경한다.
  - `"httpd"` 패키지를 설치한다.
  - `"grep ServerName /etc/httpd/conf/httpd.conf", "www.example.com"` 이 내용을 `"nodea.example.com`으로 변경한다.
  - `'sed'` 명령어를 통해서 위의 내용을 수정한다.
- 호스트 이름을 파일로 저장한다.
  - 호스트의 이름은 시스템 변수에서 가져온다.
  - 호스트 이름을 `"nodea.example.com"`으로 변경.
  - 해당 이름을 `"result-hostname.txt"`으로 저장한다.
- `'find / -type f -print'` 실행 하였을 때 발생하는 오류 메시지를 `"/dev/null"`로 전달한다.
  - 단, 표준 출력은 그대로 화면에 출력한다.
  - `adduser test1` 사용자 생성 후 위의 작업을 수행.
  - `"su - test1"` `find` 명령어 실행

# 1일 평가문제

간단한 오전 프로젝트. 첫날 학습한 내용을 가지고 아래와 같은 작업을 스크립트로 자동화 한다.

1. servera에서 선 작업 및 테스트 진행

2. serverb에서 실 작업 진행

- `mutt`
  - `dnf install mutt -y`
- `mailx`
  - `mail -s event -a /etc/hostname root@localhost`
- `postfix`
  - `dnf install postfix && systemctl enable --now postfix`

# 1일 평가문제

첫째 날 진행하였던 내용을 가지고 아래 내용을 쉘 스크립트로 구현한다. 다만, 'if', 'case', 'test'와 같은 부분을 학습하지 않았기 때문에, 최대한 단순한 방식으로 스크립트를 작성한다. 예약작업이 필요한 경우, cron, timer 둘 중 아무거나 사용하여도 된다. 뒤에 "systemd-timer"의 예제를 추가 해두었다.

1. "setup-httpd.sh", "create-user.sh" "start-httpd.sh", "setup-host.sh"를 생성한다.
2. "setup-httpd.sh"는 웹 서버를 스크립트 형태로 설치 한다.
3. "create-user.sh"는 사용자를 스크립트 형태로 생성 및 구성한다.
4. "service-httpd.sh"는 설치된 웹 서비스를 외부에서 접근이 가능하도록 한다.
5. "setup-host.sh"는 서버의 아이피 주소/호스트 이름/NTP서버 설정을 수행한다.
6. "collect-logs.sh"는 서버에서 발생한 오류를 메일로 발송한다.

# 1일 평가문제

서비스 파일 생성은 아래와 같이 한다.

```
# vi /etc/systemd/system/foo.service
[Unit]
Description=this is foo service

[Service]
ExecStart=/bin/bash <LOCATION_SCRIPT>

[Install]
WantedBy=multi-user.target
```

# 1일 평가문제

".timer"자원 설정은 다음과 같다.

```
# systemctl list-timers
# vi /etc/systemd/system/foo.timer
[Unit]
Description=Run foo weekly and on boot
[Timer]
OnBootSec=15min
OnUnitActiveSec=1w
OnCalendar=*-*-* 2:00:00
Unit=foo.service

[Install]
WantedBy=timers.target
```

# 1일 평가문제

명령어 힌트는 다음과 같다.

```
sed -i 's/ /'  
systemctl  
journalctl  
hostnamectl  
mkpasswd  
adduser  
dnf  
timedatectl  
systemd-run
```

변수

BASH

# 변수

셸에서 변수는 다음과 같이 선언한다.

```
tangshell='This is mine shell'
```

"tangshell"를 변수 이름이 되며, 그 뒤에 홑 따옴표에 있는 내용은 셸 변수에서 사용하는 자료(data)가 된다. BASH에서는 이 부분을 "string content"라고 한다.



# 변수

bash는 C 스타일을 사용하기 때문에 변수 이름에 절대 빈 공간이 있으면 안된다. 보통 다음과 같이 많이 선언한다.

```
tangshell = 'fish'  
tangvar='GNU'\''s Not Unix!'  
tangprompt="$USER$HOST"
```

변수를 호출 할 때 C언어와 마찬가지로 "\$"붙여서 접근이 가능하며, **읽기전용(read only)**상태이다.

# 변수 목록

변수 목록 및 출력을 하기 위해서는 다음과 같은 방법을 사용해서 호출한다.

```
$ today="2022-06-26"
```

```
$ printf '%s\n' "$today"
```

```
2022-06-26
```

```
$ declare -p today
```

```
declare -- today='2022-06-26'
```

# 변수 목록

'declare'에 표시되는 "--"는 특별한 속성이 아니라는 표시이다. 'declare -p'명령어를 실행하면 다음과 같은 내용을 볼 수 있다.

```
$ declare -p
declare -- BASH="/bin/bash"
declare -- PS1="\u@\h:\w\$"
```

# 변수 네이밍

변수 네이밍에 대한 규칙은 따로 없지만 일반적으로 다음과 같은 규칙을 가진다.

- test\_var
- TestVar
- testvar
- Testvar
- \_testvar
- testVar
- TestVar\_

# 변수 네이밍

하지만 다음과 같은 이름은 허용하지 않는다.

- `1testVar`
- `test-var`
- `test.var`
- `test:var`

# 변수 설정

또한 변수 설정 시 기존 설정된 이름과 충돌을 주의해야 한다. 매우 자주 있는 케이스는 보통 다음과 같다.

```
#!/bin/bash  
PATH=/home/tang/sbin:/home/tang/bin  
grep -- teststring "$PATH"
```

결과는 다음과 같이 출력이 된다.

```
grep: command not found
```

# 변수 네이밍

이유는 간단하다. \$PATH는 시스템 변수이며, 기존에 설정된 경로 값에 덮어쓰기가 되어버린 상태다. 그래서 PATH라는 이름을 사용한다면 다음처럼 선언을 보통 한다.

```
#!/bin/bash
path=/home/tang/bin:/home/tang/sbin
grep -- teststring "$path"
```

# 시스템 변수

혹은 기존 **\$PATH**를 사용하면서 내용을 추가하는 경우, 다음처럼 처리가 가능하다.

```
$ PATH="$PATH:/home/tang/bin/:/home/tang/sbin"  
$ MYHOME=/home/tang  
$ export MYHOME
```



# 변수 취소

변수 초기화 하는 방법은 다음과 같다.

```
$ system_var =
```

그리고 초기화 된 내용을 확인 시 다음과 같은 방법으로 가능하다.

```
$ declare -p system  
declare -- system=""
```

# 변수 확인

POSIX 기준으로 변수에 값이 설정이 되어 있는지 확인 시, 다음과 같이 if문으로 확인한다.

```
#!/bin/bash
test_system=linux
system=windows
if [[ -n $test_system ]] ; then
    printf 'the 'test_system' value is: %s\n' "$system"
fi
```

# 변수 취소

변수를 완전히 제거하려면 unset명령어를 통해서 할 수 있다.

```
$ unset -v test_system
```

```
$ declare -p system
```

```
bash: declare: system: not found
```

# 시스템 변수

```
$ echo $$  
$ bash  
$ echo $$  
$ declare -p BASHPID  
declare -ir BASHPID= "331219"  
$ declare -p shellusername  
declare: shellusername: not found
```

# 시스템 변수

다시 이전 환경으로 돌아가기 위해서는 'exit'명령어 실행한다. 시스템 변수는 일반적으로 대문자로 명시가 되어 있으며, 대문자 BASH로 시작하는 경우가 많다.

```
$ exit  
$ declare -p BASHPID  
$ declare -p ZSH  
$ declare -p shellusername
```

# 시스템 변수

만약 다른 환경에도 값을 전달하고 싶은 경우 POSIX명령어 export로 선언한다.

```
$ USERNAME="choi"  
$ export USERNAME  
$ declare -p USERNAME  
declare -x USERNAME="choi"
```

# 시스템 변수

앞에서 이야기 하였지만, 변수 이름 설정 시 전부 대문자로 설정한다. 'export'로 설정하는 경우 'declare'명령어로 변수를 확인하면 다음처럼 변수 이름이 출력이 된다.

```
$ bash
$ declare -p USERNAME
declare -x USERNAME="choi"
$ exit
```

# 시스템 및 애플리케이션 변수

환경변수는 bash에서만 사용하는 게 아니다.

프로그램 및 운영체제 환경에서 적용이 되며, 다른 프로그램 언어 Python, PHP, Awk같은 명령어에서도 적용이 된다. 예를 들어서 'sort', 'date' 혹은 'ls'같은 명령어는 몇몇 시스템 변수에 적용이 받는데 그 중 하나가 "**LANG**", "**LC\_ALL**", "**LC\_COLLATE**" 변수이다.



# 시스템 변수

```
$ printf '%s\n' 'helloworld' 'HelloTang' > words
```

```
$ LC_COLLATE=en_US.UTF-8
```

```
$ export LC_COLLATE
```

```
$ sort words
```

```
helloworld
```

```
HelloTang
```

# 시스템 변수

```
$ LC_COLLATE=C
```

```
$ sort words
```

```
HelloTang
```

```
helloworld
```

```
$ LC_COLLATE=C sort words
```

```
HelloTang
```

```
helloworld
```

# 일반변수

셸에서 변수를 확장하기 위해서 **\$1**를 사용한다. 일반적으로 **\$0**은 자신이 사용하는 명령어, **\$1**부터는 확장 파라미터라고 한다.

```
$ yourname='tang'
$ printf '%s\n' "Hello, $yourname."
Hello, Tang.
```

# 명령어 인자 값

입력 값 처리	설명
\$0	스크립트 자체 값. 보통 실행하는 명령어가 들어간다.
\$1	첫 번째 변수 값. 보통 명령어 및 스크립트의 첫 번째
\${10}	만약, 10개 이상의 변수가 들어오거나 혹은 한 자리 이상의 변수 순서 선언이 필요한 경우, 괄호로 표시한다.
\$#	인수에 필요한 개수를 설정 합니다.
\$*	모든 인수를 참조합니다.

# 일반변수

변수 이름을 좀 더 명확하게 보기 위해서 중괄호로 표시를 할 수 있다.

```
$ yourname='tang'
$ printf '%s\n' "__Hello, ${yourname}__."
__Hello, Tang__.
```

중괄호를 사용하지 않아도 되지만, 배시 셸 버전 4.0이후부터는 중괄호 기반으로 변수 선언으로 권장.

# 일반변수

역시 변수에 값이 선언이 되어 있지 않는 경우에는 다음처럼 출력이 된다.

```
$ printf '%s\n' "Hello, $notest."  
Hello, .
```

# 일반변수

만약에 선언이 되지 않는 변수에 대해서 오류 메시지를 출력하고 싶으면 다음처럼 선언한다.

```
$ set -u
$ printf '%s\n' "Hello, $notest."
bash: notest: unbound variable
```

# 일반변수

변수를 외부에서 받아 오기 혹은 읽어오기 할 때 내장함수인 **read**를 사용해서 사용한다. 이때 같이 사용하는 옵션은 **"-r"**를 같이 사용해서 변수에 할당한다.

```
$ read -r yourname  
$ declare -p yourname  
declare -- yourname='tang'
```



# 명령어 결과를 변수로

종종 사용하는 방법으로, 실행된 명령어의 결과를 변수에 저장한다. 이유는 여러가지가 있지만 일반적으로 재가공 혹은 출력 용도로 사용한다. 다음과 같은 예제를 본다.

```
$ whom=$(whoami)
$ printf '%s\n' "$whom"
tang
```

# 변수 계산

변수의 값을 계산하는 방법은 다양하게 있다. 아래 예제들은 변수 값 계산 방법에 대해서 나열 한다. 아래 방법중에서 편한 방법을 사용하면 된다. 계산 범주가 크거나 혹은 소수점이 발생하는 경우, 외부 명령어를 통해서 계산을 권장한다.

```
$ tmp_a=3
$ tmp_b=2
$ printf 'The sum of the a + b is %u\n' $((tmp_a + tmp_b))"
The sum of the a + b is: 5
```

# 계산

```
$ diff=$((tmp_a - tmp_b))  
$ printf 'The difference of the two numbers is: %u\n' "$diff"  
The difference of the two number is: 1  
  
$ printf '%u\n' "$(3**2)"  
$ printf '%u\n' "$(180/60)"  
$ printf '%u\n' "$(2>1)"  
$ printf '%u\n' "$(30>40)"
```

# 계산

```
$ printf '%.2f\n' "$((3/2))"  
1.00  
$ bc <<'EOF'  
scale=2  
3/2  
EOF  
1.50  
$ awk 'BEGIN { printf "%.2f\n", 3/2 }'  
1.50
```

# 명령어 결과를 변수로

두 개 이상의 명령어를 동시에 사용해야 되는 다음처럼 명령어를 구성한다.

```
$ yourpasswd="$(getent passwd $(whoami))"
```

# 기본값 설정

기본값 설정을 종종 필요하다.

예를 들어서 시스템에서 올바르게 기본값이 설정이 안되어 있으면 올바르게 동작이 안될 수 있다. 그래서 쉘에서는 `"${default_var:-default}"` 같은 형식을 제공한다.

```
$ cd -- "${MYDATA_DIR:-/usr/local/data}"
```

혹은 다음과 같이 재선언이 가능하다.

```
$ cd -- "${MYDATA_DIR:=/usr/local/data}"
```

# 문자열 입력

사용자가 입력한 문자열을 변수로 입력 받는 방법은 다음과 같다.

```
$ read username  
$ read number
```

위의 명령어가 실행이 되면, username 변수에 사용자가 입력한 값이 저장된다. 문자열로 출력하기 위해서 아래와 같이 실행한다.

```
$ printf "Your name is %s\n" "$username"
```

숫자 출력은 아래와 같이 한다.

```
$ printf "%5.2f\n" "number"
```

# 연습문제

다음과 같이 간단하게 변수를 선언 및 구성한다.

1. 변수 a에 1234값을 할당한다.
2. 변수 b에 5678값을 할당한다.
  - 변수 a, b의 합을 변수 total에 저장한다. 저장된 total변수를 화면에 출력한다.
3. 사용자의 이름을 user\_name이라는 변수에 저장한다.
  - 입력 받은 사용자 이름을 화면에 출력한다.
4. 호스트의 이름을 user\_hostname으로 저장한다.
  - 호스트 이름은 hostname명령어로 불러온다.
  - 호스트 이름을 user\_hostname으로 출력한다.



배열

bash

# 배열

셸에서 배열을 사용하기 위해서는 **경계기호(delimiter)**를 사용해서 변수와 배열을 구별한다.

```
$ devices=('iphone', 'android', 'winphone')
```

위의 배열은 매우 기본적인 배열 선언이다. 배열은 쉼표(,)로 분리한다. 순서는 0번부터 시작이 된다. 맨 끝에는 보통 "End of Array"라는 "EOA"라는 값이 붙어있다.

# 배열

```
$ printf '%s\n' "${devices[0]}"
```

```
iphone
```

```
$ printf '%s\n' "${divces[2]}"
```

```
winphone
```

# 배열

```
$ printf '%s\n' "${devices[-1]}
```

```
winphone
```

```
$ printf '%s\n' "${devices[@]}"
```

```
iphone
```

```
android
```

```
winphone
```

# 배열

```
$ printf '%s\n' "${devices[@]^}"
```

IPHONE

ANDROID

WINPHONE

# 배열

```
$ homefiles=("$HOME"/*)
$ printf '%s\n' "${homefiles[@]}"
/home/tang/1.txt
/home/tang/2.txt
/home/tang/3.txt
/home/tang/4.txt
```

# 배열

**bash4.0** 이후에 새로 할당된 배열 할당이 있다. 파이선처럼 기능을 제공한다.

```
$ declare -A colors  
$ colors=([tomato]="red" [orange]="yellow" [carrot]="orange")  
$ printf '%s\n' "${colors[orange]}"  
yellow
```

# 사전(table)

배시 4.0부터 사전(dictionary)를 지원한다. 하지만, 이는 리눅스 및 MacOS(OS X)에서만 지원한다. 사전 형식의 배열을 사용하기 위해서는 'declare'로 선언한다. 배시 3.0에서는 네이티브로 지원하지 않지만, "index array"로 구현이 가능하다. 이 과정에서는 배시 3.0에서 구현은 다루지 않는다.

```
$ declare -A animals
$ animals=(["moo"]="cow" ["woof"]="dog" ["meaw"]="cat")
```

혹은 다음과 같은 방법으로 선언이 가능하다.

```
$ declare -A animals=(["moo"]="cow" ["woof"]="dog" ["meaw"]="cat")
```

이를 호출하기 위해서 'echo'명령어나 혹은 'for'키워드로 루프를 돌려서 출력이 가능하다.

```
$ echo "${animals[moo]}"
$ for sound in "${!animals[@]}" ; do echo "$sound - ${animals[$sound]}"; done
```



# DICT VS MAP

- a **dictionary** is a data structure representing a set of elements, with insertion, deletion, and tests for membership; the elements may be, but are not necessarily, composed of distinct *key* and *value* parts
- a **map** is an *associative* data structure able to store a set of *keys*, each associated with one (or sometimes more than one - e.g. C++ multimap) *value*, with the ability to *access* and *erase* existing entries given only the key.
- "*Map*" is used by Java, C++
- "*Dictionary*" is used by .Net, Python
- "*Associative array*" is used by PHP

<https://stackoverflow.com/questions/2884068/what-is-the-difference-between-a-map-and-a-dictionary>

# DICT VS MAP

## MAP VS. DICTIONARY

### Map ADT

- Collection of **Entries**
  - ▣ *key* – searched for
  - ▣ *value* – cared about
- Implemented with:
  - ▣ List w/ **Entries** in order they were added
  - ▣ List w/ **Entries** in increasing order of *keys*
  - ▣ Hash table

### Dictionary ADT

- Collection of **Entries**
  - ▣ *key* – searched for
  - ▣ *value* – cared about
- Implemented with:
  - ▣ List w/ **Entries** in order they were added
  - ▣ List w/ **Entries** in increasing order of *keys*
  - ▣ Hash table

# ARRAY VS LIST



# 연습문제

과일 관련 배열을 다음과 같은 조건으로 생성한다.

1. "fruits"라는 리스트 이름을 생성한다.
  - "apple", "pear", "mango", "mulbeery", "peach"이와 같이 리스트 값을 구성한다.
2. "APT"라는 사전 이름을 생성한다.
  - "101: Hans", "102: Tom", "103: Yun", "104: Kim"으로 사전 값을 구성한다.
3. 위의 값들을 화면에 출력한다.

# 종합문제

아래와 같이 변수를 생성 및 구성한다.

1. 사용자의 영문 이름을 변수로 생성한다.
  - 생성 시, 사용자 이름을 셸에서 입력 받는다.
  - 쉘 스크립트로 구성하여도 상관 없다.
2. 생성된 사용자 이름을 화면에 출력한다
  - 'echo'명령어로 사용자 이름을 출력한다.
  - 'printf'명령어로 사용자 이름을 출력한다.
    - 출력 시, 문자열 타입으로 출력 한다.
  - 사용자 비밀번호를 "helloworld"으로 설정한다.
  - 생성된 사용자를 'getent', 'grep'를 통해서 올바르게 생성이 되었는지 확인한다.

# 글로빙

globbing

# 글로브(\*) 사용하기

글로브, 별표 혹은 눈이라고 말하는 문자이다. 셸에서는 글로브 즉 별표를 사용하면 기본적으로 "모든 것(all of match)"라고 표현한다. 매치가 되는 자원은 거의 모든 자원에 적용이 된다.

```
$ ls -a
.      ..      Documents  Pictures  Downloads
$ printf '%s\n' *
hello
world
Documents
Downloads
```

# 글로브(\*) 사용하기

```
$ printf '%s\n' ???uments
```

Documents

```
$ printf '%s\n' *[ts]
```

test1.txt

test2.txt



# 글로브(\*) 사용하기

앞에 느낌표가 붙어 있으면, 해당 단어는 제외하고 출력한다.

```
$ printf '%s\n' ???uments
```

```
Documents
```

```
$ printf '%s\n' *(!ts]
```

```
jsues.txt
```

```
sjwue.txt
```

# 글로브(\*) 사용하기

'shopt'명령어는 bash의 빌트인 명령어이며, 쉘의 추가적인 값(optional behavior)을 변경한다. "-s"에서 사용이 가능한 대표적인 옵션은 다음과 같다. 더 많은 옵션은 아래에서 확인한다.

1. autocd: 'cd'명령어를 사용하지 않고 자동적으로 디렉터리를 이동한다.
2. cdspell: 'cd'명령어 사용 시, 오타가 있는 경우, 자동으로 수정 및 이동한다.

```
$ shopt -s autocd
$ shopt -s cdspell
$ /usr/share/doc
$ shopt -u autocd
$ cat -- c*
cat: 'c*' : No such file or directory
```

```
$ shopt -s nullglob
$ printf '%s\n' c*
```

# 글로브(\*) 사용하기

대소문자 구별없이 글로빙하기 위해서 다음과 같이 셸 옵션 변경이 가능하다.

```
$ shopt -s nocaseglob  
$ printf '%s\n' A*  
apple  
Aux
```

# 글로빙 확장하기

bash는 부족한 글로빙 기능을 확장하기 위해서 정규식(**regular express**)기반으로 고급기능(혹은 더 복잡하게) 사용이 가능하다. 혹은 쉘의 내부 기능을 활용하여 좀 더 확장하는 방법도 있다.

다만, 범주가 너무 넓기 때문에 사용자가 편한 방식으로 사용을 권장한다. 아래는 일반적으로 사용하는 글로빙 방식이다.

- `?(pattern)`: 하나라도 패턴에 맞으면 출력
- `+(pattern)`: 최소 한 개로도 맞으면 출력
- `*(pattern)`: 숫자 하나라도 맞으면 출력
- `@(pattern)`: 명시한 조건만 맞으면 출력
- `!(pattern)`: 조건에 맞지 않는 것만 출력

# 글로빙 활용하기

예를 들어서 자주 사용하는 **"/var/log"**를 예를 들면 다음처럼 사용한다.

**/var/log**에서 .gz로 끝나지 않는것만 출력

```
$ ls -l /var/log/!(*.log)
```

haw혹은 mess단어가 맞으면 화면에 출력

```
$ ls -l /var/log/@(haw|mess)*
```

# 기본 명령어

일반 명령어

# ps

1. UNIX options, which may be grouped and must be preceded by a dash.
2. BSD options, which may be grouped and must not be used with a dash.
3. GNU long options, which are preceded by two dashes.

<https://unix.stackexchange.com/questions/78691/unix-bsd-gnu-options-in-linuxs-ps-command-where-are-they-from>

# ps

많이 ps 옵션은 보통 아래와 같다.

```
# ps aux or ps au
```

- "a" 옵션은 BSD 스타일의 "only yourself" 옵션
- "u" 옵션은 EUID(effective user ID)
- "x" 옵션은 x BSD 스타일의 "must have a tty" 옵션

```
[root@localhost ~]# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1   0.0   0.7 128144  6828 ?        Ss   Sep11   0:02 /usr/lib/systemd/systemd
root         2   0.0   0.0      0     0 ?        S    Sep11   0:00 [kthreadd]
root         4   0.0   0.0      0     0 ?        S<   Sep11   0:00 [kworker/0:0H]
root         5   0.0   0.0      0     0 ?        S    Sep11   0:00 [kworker/u32:0]
root         6   0.0   0.0      0     0 ?        S    Sep11   0:00 [ksoftirqd/0]
root         7   0.0   0.0      0     0 ?        S    Sep11   0:00 [migration/0]
root         8   0.0   0.0      0     0 ?        S    Sep11   0:00 [rcu_bh]
root         9   0.0   0.0      0     0 ?        R    Sep11   0:36 [rcu_sched]
root        10   0.0   0.0      0     0 ?        S<   Sep11   0:00 [lru-add-drain]
root        11   0.0   0.0      0     0 ?        S    Sep11   0:00 [watchdog/0]
```



# ps

```
# ps -ef
```

- "-f" 옵션은 "full-formatting list"
- "-e" 옵션은 "all processes"

위의 명령어는 Unix(AIX)스타일의 명령어

```
[root@localhost ~]# ps -ef
UID          PID  PPID  C  STIME TTY          TIME CMD
root           1     0  0  Sep11 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --
root           2     0  0  Sep11 ?        00:00:00 [kthreadd]
root           4     2  0  Sep11 ?        00:00:00 [kworker/0:0H]
root           5     2  0  Sep11 ?        00:00:00 [kworker/u32:0]
root           6     2  0  Sep11 ?        00:00:00 [ksoftirqd/0]
root           7     2  0  Sep11 ?        00:00:00 [migration/0]
root           8     2  0  Sep11 ?        00:00:00 [rcu_bh]
root           9     2  0  Sep11 ?        00:00:36 [rcu_sched]
root          10     2  0  Sep11 ?        00:00:00 [lru-add-drain]
root          11     2  0  Sep11 ?        00:00:00 [lru-add-drain]
```

# ps

# ps -x

```
[tang@www ~]$ ps -x
```

PID	TTY	STAT	TIME	COMMAND
69492	?	Ss	0:00	/usr/lib/systemd/systemd --user
69496	?	S	0:00	(sd-pam)
79289	?	Ss	0:05	tmux
79290	pts/15	Ss	0:00	-sh
79320	pts/21	Ss	0:00	-sh
79350	pts/22	Ss	0:00	-sh
80825	pts/15	S+	0:00	ssh root@192.168.90.171
80828	pts/15	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
80861	pts/21	S+	0:00	ssh root@192.168.90.3
80864	pts/21	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
80865	pts/22	S+	0:00	ssh root@192.168.90.168
80868	pts/22	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.168

# ps

```
# ps -fu or -fU <UID> <UNAME>
```

```
# ps -fU tang
```

```
[tang@www ~]$ ps -fU tang
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
tang	69492	1	0	Sep06	?	00:00:00	/usr/lib/systemd/systemd --user
tang	69496	69492	0	Sep06	?	00:00:00	(sd-pam)
tang	79289	1	0	Sep06	?	00:00:05	tmux
tang	79290	79289	0	Sep06	pts/15	00:00:00	-sh
tang	79320	79289	0	Sep06	pts/21	00:00:00	-sh
tang	79350	79289	0	Sep06	pts/22	00:00:00	-sh
tang	80825	79290	0	Sep07	pts/15	00:00:00	ssh root@192.168.90.171
tang	80828	80825	0	Sep07	pts/15	00:00:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
tang	80861	79320	0	Sep07	pts/21	00:00:00	ssh root@192.168.90.3
tang	80864	80861	0	Sep07	pts/21	00:00:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3

# ps

```
# ps -U root -u root
```

"-u" 영향 받는 사용자 아이디(RUID)

"-U" 실제 사용자 아이디(EUID)

```
[tang@www ~]$ ps -u tang -U tang
  PID TTY          TIME CMD
 69492 ?           00:00:00 systemd
 69496 ?           00:00:00 (sd-pam)
 79289 ?           00:00:05 tmux: server
 79290 pts/15      00:00:00 sh
 79320 pts/21      00:00:00 sh
 79350 pts/22      00:00:00 sh
 80825 pts/15      00:00:00 ssh
 80828 pts/15      00:00:00 sss_ssh_knownho
```

# ps

```
# ps -Fg tang
# ps -fG 1000
```

```
[tang@www ~]$ ps -Fg tang
UID      PID     PPID  C   SZ   RSS  PSR STIME TTY
tang    69492       1   0 22388 9528   4 Sep06 ?
tang    69496   69492   0 81902 3320   0 Sep06 ?
tang    79289       1   0  7246 4528   5 Sep06 ?
tang    79290   79289   0  6929 5140   3 Sep06 pts/15
tang    79320   79289   0  6929 5012   5 Sep06 pts/21
tang    79350   79289   0  6929 4988   3 Sep06 pts/22
tang    80825   79290   0 15448 6812   6 Sep07 pts/15
tang    80828   80825   0 24797 5588   6 Sep07 pts/15
tang    80861   79320   0 15448 6920   3 Sep07 pts/21
```

# ps

```
# ps -fp
```

```
[tang@www ~]$ sudo ps -fp 1
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 Sep03 ?        00:01:30 /usr/li
```

# ps

```
# ps -f --ppid 1
```

```
[tang@www ~]$ sudo ps -f --ppid 1
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	751	1	0	Sep03	?	00:00:11	/usr/lib/systemd/systemd-journald
root	799	1	0	Sep03	?	00:00:06	/usr/lib/systemd/systemd-udevd
rpc	899	1	0	Sep03	?	00:00:03	/usr/bin/rpcbind -w -f
root	908	1	0	Sep03	?	00:00:04	/sbin/auditd

# ps

'ps'명령어는 제일 기본적으로 사용하는 프로세스 확인 명령어. 제일 많이 사용하는 명령어는 다음과 같다. 이 방식은 BSD나 혹은 GNU/Linux에서 많이 사용한다.

```
# ps -ef
```

위의 명령어는 AIX(Unix)에서 사용하던 유닉스 스타일 명령어

```
# ps aux
```

위의 명령어는 GNU 스타일의 명령어. 일반적으로 리눅스 엔지니어들이 많이 사용하는 스타일은 "**dash(-)**"스타일이다.



# ps

ps명령어는 쉘과 함께 사용할 때 다음처럼 활용한다.

```
$ ps -ax -ocmd,pid,gid,ppid,rss
```

혹은 다음과 같이 파이프를 통해서 활용이 가능하다.

```
$ ps -uapache -ocmd,pid,gid,ppid,rss | uniq | awk '{ print $1 }'
```

위의 내용을 변수로 처리하기 위해서는 다음과 같이 작업을 수행한다.

```
$ tmp=$(ps -uapache -ocmd,pid,gid,ppid,rss | uniq | awk '{ print $1 }' | wc  
-l)  
$ echo $tmp
```

# grep

표준 입출력에서 문자열을 찾는다. grep 사용방법은 보통 두 가지가 있다. 이 명령어는 ps명령어와 같이 많이 사용한다.

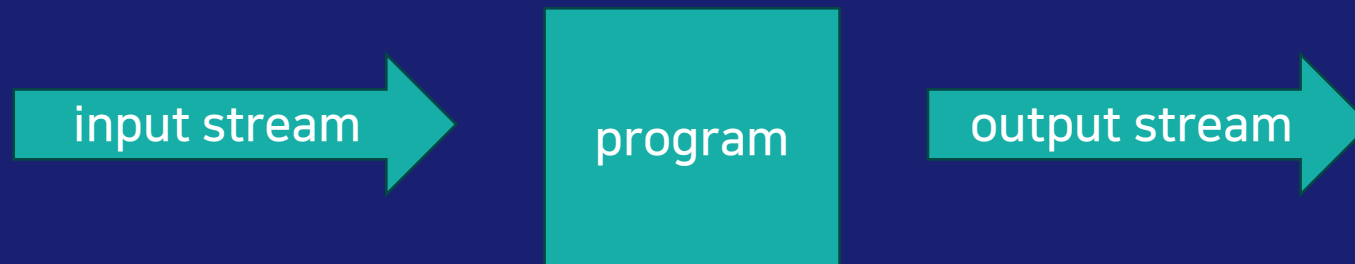
1. 특정 대상으로 문자열 찾는 방법
2. 스트림(stream)기반으로 출력 문자열 찾는 방법

아래와 같은 방법을 문자열 검색이 가능하다.

```
$ grep -r GNU /usr/share/doc/  
$ cat /etc/share/doc/procps-ng/README.md | grep GNU  
  
$ sudo dnf install cowsay  
$ sudo dnf install fortune-mod
```

# 표준 스트리밍

리눅스 및 유닉스에서는 공통적으로 스트림 문자열 처리를 지원 및 제공한다. 대다수 명령어는 표준 라이브러리에서 전부 호환 및 사용이 가능하다.



# grep

grep명령어는 실제로, egrep, pgrep 그리고, 일반적으로 우리가 사용하는 grep이 있다. grep명령어를 활용하면 다음처럼 활용이 가능하다.

```
$ grep -e httpd -e gnome -e gnu -Ri /usr/share/docs
```

위의 명령어는 한번에 여러 단어를 표준 출력에서 찾는 명령어이다. 맨 뒤의 "-Ri"는 디렉터리 검색 및 대소문자 구별 없음을 명시한다. 만약, 프로세스의 아이디를 확인하고 싶으면 다음과 같은 방법으로 확인 및 응용이 가능하다.

```
$ sudo dnf install httpd -y
$ systemctl start httpd
$ proc_httpd=$(pgrep httpd)
626931
632261
632263
632264
632267
$ kill $proc_httpd
$ systemctl is-active httpd
```

# kill/killall/pkill

프로세스 종료 시그널은 아래와 같이 많이 사용한다.

- 1: 프로세스를 재시작 한다. 모든 프로세스에서 적용되는 신호는 아니다.
- 9: 현재 사용중인 프로세스를 강제로 메모리에서 종료한다. 이는 보호가 되지 않기 때문에, 프로세스는 종료하면서 모든 데이터를 소멸한다.
- 15: 정상적으로 프로세스를 종료한다.

x86에서는 프로세스 종료 시, 무조건 15번 시그널 사용을 권장한다. 보통 무심코, 9번 시그널을 종료 용도로 사용하는데, 메모리에 있는 데이터를 디스크에 쓰기를 하지 못하고 종료가 될 가능성이 매우 높다.

# pgrep/pkill

'pgrep'은 프로세스의 이름 혹은 다른 속성을 확인하여 시그널 전달. 아래 셸 변수 형태로 프로세스를 핸들링 한다.

```
$ pkill <signal> <process_name>
$ pgrep -u root sshd
$ pgrep -u tang,root,daemon
$ kill $(pgrep -u tang)
```

# pgrep

```
$ pgrep -c ssh
```

```
3
```

```
$ pgrep -d ssh
```

```
778
```

```
726343
```

```
726828
```

```
$ pgrep -d "-" ssh
```

```
778-d726343-d726828
```

# pgrep

```
$ pgrep -u tang ssh
```

```
778
```

```
726343
```

```
726828
```



# pgrep

```
$ pgrep -u tang ssh -c
```

```
12
```

```
$ pgrep -l ssh
```

```
1363 sssd_ssh
```

```
30317 sshd
```

```
79439 ssh
```

```
79442 sss_ssh_knownho
```

```
79459 ssh
```

```
79462 sss_ssh_knownho
```

# pgrep

```
$ pgrep -a ssh
```

```
79439 ssh root@192.168.90.178
```

```
79442 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.178
```

```
79459 ssh root@192.168.90.187
```

```
79462 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.187
```

```
80023 ssh root@192.168.90.183
```

```
80026 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.183
```

```
131706 sshd: tang@pts/0
```

# pgrep

ssh와 일치하지 않는 프로세스만 출력.

```
$ pgrep -v ssh
```

정확하게 sshd이름과 일치하는 프로세스만 출력.

```
$ pgrep -o ssh
```

최근에 생성된 ssh process만 출력.

```
$ pgrep -n ssh
```

이전에 생성된 ssh process만 출력.

```
$ pgrep -x sshd
```

# pgill

'ssh'로 시작하는 프로세스를 종료한다.

```
$ pgill '^ssh$'
```

'ping 8.8.8.8'으로 시작하는 명령어를 종료한다.

```
$ ping 8.8.8.8 &  
$ pgill -9 -f "ping 8.8.8.8"
```

사용자 'mark'를 제거 합니다.

```
$ pgill -u mark
```

# kill

사용자 "mark", "gnome"를 시그널 "-9"으로 종료한다.

```
$ kill -u mark gnome -9
```

최근에 실행한 프로세스 'screen'를 종료 한다.

```
$ kill -9 -n screen
```

# pstree

이 명령어는 프로세스 종속성(혹은 의존성)을 트리형태로 화면에 출력한다.

```
$ pstree -Ap
```

**-A:** 아스키 문자 기반으로 화면에 출력 합니다. 일반적으로 'pstree'사용시 아스키 문자로 출력한다.

**-P:** 프로세스의 PID를 화면에 같이 출력한다.

위의 명령에 대응하는 명령어는 'ps'명령어 이다. 'ps'에서 다음과 같이 트리 형태를 구현이 가능하다.

```
$ ps -ejH
```

# kill/killall

## kill

PID로 프로세스 종료. 기본값은 **15번(TERM)**으로 되어 있다.

## killall

'kill'과 동일하게 사용하는 명령어. 차이점은 **PID**가 아닌 **Process Name**으로 관리한다. 위에서 많이 사용하였기 때문에, 별도로 예제를 구현하지 않는다. 다만, 'kill'명령어는 CPU 아키텍처마다 코드번호가 다르기 때문에, 아키텍처가 다를 때는 꼭 확인 후 사용한다.

# 명령어 연속 실행 시 오류 중지

여러 명령어를 동시에 수행하는 방법은 두 가지가 있다.

```
$ cd ; rm -r /testdir ; ls
```

```
$ cd / && rm -r /testdir ; ls
```

첫 번째 방법은 오류가 발생하여도 다음명령어가 실행이 되며(or), 두 번째 방법은 첫 번째 명령어가 실행 중 문제가 발생하면 중지(and). 위의 명령어들은 명령어 실행 후 셸에서 반환되는 "Return Count(RC)"를 확인한다.



# 포그라운드/백그라운드

명령어 수행 시 포그라운드/백그라운드 실행은 다음과 같은 명령어로 실행이 가능하다.

```
$ sleep 10  
$ sleep 10 &  
$ echo $_  
$ echo $!
```

# 포그라운드/백그라운드

백 그라운드로 실행이 된 명령어를 확인 하려면 아래와 같은 명령어로 확인이 가능하다.

```
$ bg
```

# awk

"awk"는 스트리밍 데이터를 핸들링하기 위해서 만들어진 도구. 리눅스에서 사용하는 대다수 awk는 GNU AWK기반으로 사용하고 있기 때문에, 기존 유닉스 버전의 awk와 동작 방법이 다르다. awk는 일반적으로 필드 필터링 같은 용도로 많이 사용한다.

아래 예제는 "tmux"으로 시작하는 문자를 'ps'명령어 결과에서 찾아서 화면에 출력한다.

```
$ ps -ef | awk '/tmux/ { print }'
```

tang	67	12	0	13:30	pts/0	00:00:00	tmux
tang	69	11	0	13:30	?	00:00:00	tmux
tang	394	70	0	15:07	pts/1	00:00:00	awk /tmux/ { print }

# awk

혹은 원하는 필드를 하나 혹은 여러 개를 선택하여 출력이 가능하다.

```
$ ps -ef | awk '{ print $1 }'
```

```
UID
```

```
root
```

```
root
```

```
root
```

```
root
```

```
tang
```

```
tang
```

# awk

```
# ps -ef | awk '{ print $1, $3, $NF }'
```

```
UID PPID CMD
```

```
root 0 /init
```

```
root 1 --log-truncate
```

```
root 1 /init
```

```
root 10 /init
```

```
tang 11 -bash
```

# ls

만약 'ls' 명령어 없이 파일 이름 및 목록을 얻으려면 어떻게 해야 할까?

```
$ ls -l
```

```
$ grep pattern -- *
```

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

```
$ find . -type f -exec grep <PATTERN> -- {} \;
```

```
$ for file in * ; do grep -F <PATTERN> -- "$file" ; done
```

# mv

'mv'명령어는 디렉터리 및 파일을 이동하거나 혹은 이름을 바꾸는 명령어. 스크립트에서는 파일 혹은 디렉터리 이동 시 사용한다.

```
$ mv file path/to/directory
$ mv file1 file2
$ mv file1 file2 dir4 path/to/directory
$ mv testdir/ goodir/

$ touch world.txt
$ mv world.txt /tmp
$ touch world.txt
$ mv --backup=t world.txt /tmp
```

# mv

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

```
$ mv -i olduser/ newuser/
$ nano mv1.sh
#!/bin/bash
if [[ -e old_user ]] ; then
    printf 'the old user exists.\n'
    mv old_user new_user
fi
mv new_user user
```



# cp

cp 명령어는 파일이나 혹은 파일을 복사 시 사용한다. "-r", "-R"의 차이점은 기본 옵션은 GNU에서는 -r를 사용한다. 하지만, 전통적인 유닉스에서는 -R옵션을 사용한다.

```
$ cp doc1 doc2 doc3 dir/
$ ls
dir
$ cp dir/ back_dir/
$ cp -R
$ cp -r
$ cp -R back_dir/ rel_dir/
```

# rm, rmdir

'rm', 'rmdir'은 비슷한 동작을 하지만 약간의 차이가 있다.

```
$ rm testfile
```

```
$ rm -rf testdir/
```

```
$ rmdir testdir/
```

```
Directory not empty
```

# grep

'grep' 명령어는 특정 문자열을 표준 출력에서 검색 및 출력한다.

```
$ grep 'ssh' /etc/services
ssh          22/tcp          # The Secure Shell (SSH)
Protocol
ssh          22/udp          # The Secure Shell (SSH)
Protocol
x11-ssh-offset 6010/tcp        # SSH X11 forwarding offset
ssh          22/sctp         # SSH
sshell       614/tcp         # SSLshell
sshell       614/udp         #      SSLshell
```

# grep

```
$ grep '^ssh' /etc/services
```

ssh	22/tcp	# The Secure Shell (SSH) Protocol
ssh	22/udp	# The Secure Shell (SSH) Protocol
ssh	22/sctp	# SSH
sshell	614/tcp	# SSLshell
sshell	614/udp	# SSLshell
ssh-mgmt	17235/tcp	# SSH Tectia Manager
ssh-mgmt	17235/udp	# SSH Tectia Manager

# grep

```
$ grep -i 'ssh$' /etc/services
```

```
ssh                22/sctp           # SSH
netconf-ssh        830/tcp           # NETCONF over SSH
netconf-ssh        830/udp           # NETCONF over SSH
sdo-ssh            3897/tcp          # Simple Distributed Objects over SSH
sdo-ssh            3897/udp          # Simple Distributed Objects over SSH
tl1-ssh            6252/tcp          # TL1 over SSH
```

# grep

중복된 출력 결과를 다음과 같은 명령어로 정리가 가능하다.

```
$ uniq  
$ sort
```

혹은 'grep'에서 '-q' 옵션을 사용하여 조건에 맞는 문자열이 있는지 확인이 가능하다.

```
$ grep -q telnet /etc/service && echo 'Matched'  
$ echo $?  
$ grep -q foobar /etc/service && echo 'Matched'  
$ echo $?
```

# grep

혹은 몇 개의 라인이 일치하는지 확인 하려면 다음과 같은 방법이 있다.

```
$ grep -c telnet /etc/service
```

```
5
```

```
$ grep telnet /etc/service | wc -l
```

복합적으로 문자열을 확인이 필요한 경우는 다음과 같은 명령어로 확인이 가능하다.

```
$ grep -e ssh -e telnet /etc/services
```

# grep

'-F'옵션을 통해서 문자열 검색이 가능하다. 매뉴얼 설명은 다음처럼 나와있다. 특수 문자열을 일반 문자열처럼 다루기 위해서 아래와 같이 옵션을 사용한다.

```
$ grep -F '$' ikea
```

만약 매치가 되지 않는 문자열만 출력하고 싶은 경우 -v 문자열을 사용한다. '^#'옵션은 주석 문자인 #를 제외하고 화면에 출력.

```
$ grep -v '^#' /etc/services
```



# grep

조금 더 복잡하게 특정 문자만 카운팅이 필요한 경우 다음과 같이 실행한다

```
$ grep -cFv -e bash -e ' .' /etc/shells
10
```

보통 아래와 같은 옵션을 제일 많이 사용한다.

- c: 카운팅
- F: 정규식 기호 문자열 취급
- v: 예외
- e: 복수 조건(정규식)

# grep

'grep'명령어는 "OR"조건은 지원하지만, "AND"조건은 지원하지 않는다. AND조건으로 문자열을 찾기 위해서는 다음과 같은 방법을 권장한다. 이 부분은 위에서 한번 사용했던 방법이다.

```
$ grep -e <STRING> -e <STRING>  
$ grep <STRING> | grep <STRING>
```

# cut

'cut'명령어는 말 그래도 자르는 명령어. 특정 필드나 문자열을 자르고 싶은 경우 cut명령어를 통해서 사용이 가능하다.

자주 사용하는 옵션은 보통 '-d', '-f'이며, '-d'는 필드 분리 문자 '-f'는 필드이다.

```
$ cut -d: -f1 /etc/passwd
```

```
root
```

```
bin
```

```
daemon
```

```
adm
```

```
lp
```

```
sync
```

```
shutdown
```

# cut

만약, 맨 끝 필드를 출력하고 싶은 경우 아래처럼 실행한다.

```
$ cut -d: -f6- /etc/passwd  
/root:/bin/bash  
/bin:/sbin/nologin  
/sbin:/sbin/nologin
```

## WC

단어 혹은 줄 개수 그리고 바이트를 출력한다. 기본 값은 전부를 출력한다.

```
$ wc .bash_history
200  437 3373 .bash_history
```

위의 출력된 내용은 200줄, 437줄 그리고 총 3373바이트. 여기에서 특정 값만 출력을 원하는 경우 '-c', '-w', '-l'같은 옵션을 사용해서 출력이 가능하다.

'-l'옵션은 엄연히 따지고 보면 newline개수만 출력한다.

## WC

"-c"옵션 경우에는 바이트를 출력한다.

```
$ wc -c .bash_history
3373 .bash_history
```

"-m"총 문자 개수를 카운팅 하기 위해서는 "-m"옵션을 사용한다.

```
$ wc -m .bash_history
3373 .bash_history
```

한국어를 UTF-8로 입력하면 영문과 다르게 바이트 및 문자가 카운트가 된다. 그 이유는 인코딩 및 2바이트 문자열로 인하여 개수와 크기가 다르게 출력이 된다.

```
$ wc -m -c Korean
35 93 test
```

# du

'wc'명령어로 파일의 크기를 알 수 있다.

```
$ wc -c /bin/bash
```

하지만 전체적인 크기에 대해서는 확인이 불가능 하기 때문에 'du'명령어를 사용하면 된다.

```
$ du -k /bin/bash
1124    /bin/bash
```

디렉터리 크기를 알고 싶은 경우 아래처럼 실행한다. 모든 결과는 **POSIX 1kibibyte(1024)**로 출력이 된다.

```
$ du -k /bin
```

# find

'find'명령어는 리눅스에서 많이 사용하는 명령어 중 하나이다. 이 명령어는 디렉터리 및 파일 검색이 가능하다.  
기본적인 동작 방식은 다음과 같다.

```
$ find ~/
```

출력만 하고 싶은 경우 다음처럼 옵션을 지정한다.

```
$ find /usr/share -print
```



# find

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다. 아래 명령어는 "email.txt"파일에서 "**docs**", "**googles**"에서 찾는 명령어 이다.

```
$ find ~/docs ~/googles email.txt
```

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다. 아래 명령어는 결과를 sort명령어로 정렬한다.

```
$ find ~/docs | sort
```

# find

test라는 문자열을 가지고 있는 파일 및 디렉터리.

```
$ find ~/docs -name '*test*' -print
```

fish라는 문자열 가지고 있는 파일만 출력.

```
$ find ~/docs -name '*fish*' -type f -print
```

수정한지 3일이 지난 파일.

```
$ find ~/docs -mtime +3 -print
```

5일전에 수정한 파일.

```
$ find ~/docs -mtime -5 -print
```

# find

이름에 beef라는 문자열이 없는 것만 검색

```
$ find ~/docs ! -name '*beef*'
```

만약 특정 문자열에 대해서 무시를 하고 싶은 경우, 아래처럼 명령어를 실행한다.

```
$ find . -name '.git*' -prune -o -print
```

# find

find를 사용하면, 각각 결과마다 명령어를 같이 실행이 가능하다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- {} \;
```

1. -exec: 명령어 실행, 여기서는 'grep -F search --' 명령어
2. {}: 이 부분에 find에 찾을 출력 결과물로 대체
3. \;;: 명령어 종료 알림
4. --: 인자 값으로 전달 되는 값들을 옵션으로 처리 하지 않는다.(맨 페이지에서는 시작점(start point)를 ./ 혹은 /으로 처리 권장)

# find

위 명령어를 나누면 다음과 같이 두개 구조로 나누어진다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- '{}' \;
```

인자 끝  
(명령어)

치환

```
$ find /usr/share/doc -type f -name '*.vim'
```

/usr/share/doc/vim/map.vim

```
$ grep -F search -- vim/map.vim  
$ grep -F search -- vim/test.vim
```

# find

만약 'find'의 '-exec'에 더 많은 인자 값이 필요한 경우 +기호를 사용하여 좀 더 많은 인자 값 옵션 사용이 가능하다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- '{}'
```

```
$ grep -F search -- vim/map.vim vim/test.vim
```

## find -exec +

"+"와 "\";" 기능을 비교하면 다음과 같다. 명령어에 전달할 인자 값이 없는 경우 "--"를 사용하지 않아도 된다. 아래 예제는 'echo'를 사용하기에 "--"사용하지 않았다.

```
$ find /usr/share/doc -name "*.md" -type f -exec echo {} +
```

위의 명령어로 실행하는 경우 줄 매번 echo로 출력해주는 내용이라서 newline없이 화면에 출력이 된다.

```
$ find /usr/share/doc -name "*.md" -exec echo {} \;
```

## find with cp

'find' 명령어로 파일 복사 시, 다음과 같이 활용한다.

```
# mkdir -p /tmp/md_files/  
# find /usr/share/doc -name "*.md" -type f -print -exec cp '{}'  
/tmp/md_files \;
```



# xargs

이 명령어는 find -exec와 비슷하지만 조금 다른 동작 구조를 가지고 있다.

```
$ find /usr/share/doc -type f -name '*.md' | xargs grep -F search --
```



데이터 스트림

위의 명령어는 find에서 실행한 결과를 오른쪽 'xargs' 명령어 전달, 'grep' 명령어가 작업을 수행한다.

# xargs

만약, 출력되는 결과에 null-byte로 종료되는 경우가 있다. 올바르게 동작이 안될 수 있기 때문에 find에서 xargs를 사용시 -print0이라는 옵션을 사용한다.

*'find -print0' 혹은 'xargs -0'*

```
$ find /usr/share -type f -name '*.vim' -print0 | xargs -0 grep -F search --
```

# xargs

만약, 출력되는 결과에 null-byte로 종료되는 경우가 있다. 올바르게 동작이 안될 수 있기 때문에 find에서 xargs를 사용시 -print0이라는 옵션을 사용한다.


'find -print0' 혹은 'xargs -0'

```
$ find /usr/share -type f -name '*.vim' -print0 | xargs -0 grep -F search --
```

## find + cp + xargs

```
$ find / -type f -name LICENSE -exec cp {} -- /tmp/LICENSE-$(od -An -N2 -i /dev/random | grep -v ^$ | tr -d [:blank:]) \;
```

```
$ find / -type f -name LICENSE | while IFS= read -r f; do cp -v "$f"  
"/tmp/${f//\//\_}"; done
```



표준 출력으로 전달 받은 입력 내용중 디렉터리 파싱

## find+

추가적으로 'find'의 -exec 및 '{} +'에 대해서 추가적인 설명. 앞에서 말한 옵션 "-exec {} \", "-exec {} +", "| xargs" 총 3가지로 사용이 가능하다. 기본적으로 'find'에서 검색된 내용은 "-exec"에서 이벤트가 발생한다. 만약 삭제라고 한다면 'find'에서 찾은 내용은 'rm'명령어가 삭제를 한다.

# find+

아래는 같은 find방식이지만, 처리 하는 방법 혹은 방식은 동일하다.

1. `find . -name "*.txt" -exec rm {} \;`
2. `find . -name "*.txt" -exec rm {} +`
3. `find . -name "*.txt" | xargs rm`

# find+

1번 경우에는 실행이 되는 1:1 명령어로 수행이 된다. 예를 들어서 다음과 같이 동작한다. 삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} \; → rm -f test1.txt  
$ find . -name "*.txt" -exec rm {} \; → rm -f test2.txt
```

## find+

2번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다. 삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음 처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} + → rm -f test1.txt test2.txt
```



# find+

3번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다. 삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음 처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} + → rm -f test1.txt test2.txt
```

하지만 3번 경우에는 문제가 있다.

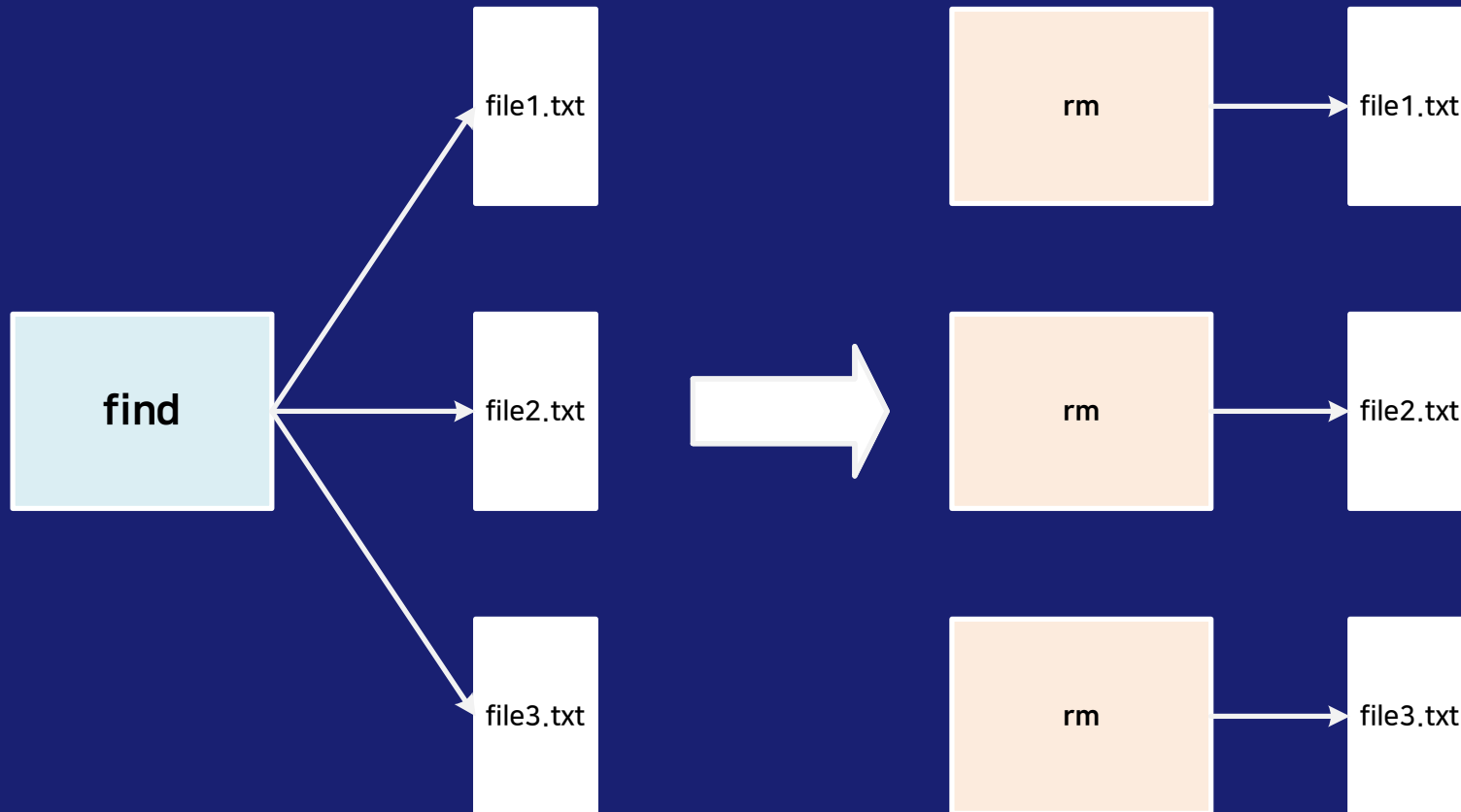
## find+

3번은 "**null string**"이 들어오는 경우 올바르게 처리는 하지 못하는 문제가 있다.

여기서 말하는 "null string"은 말 그래도 비어 있는 문자이다. 예를 들어서 파일명이 "**test1.txt**", "**test2.txt**", "**test document.txt**"되어 있는 경우 "test document.txt"는 비어 있는 공간이 있기 때문에 다른 파일로 인식이 된다.

```
$ rm -f test1.txt test2.txt test document.txt
```

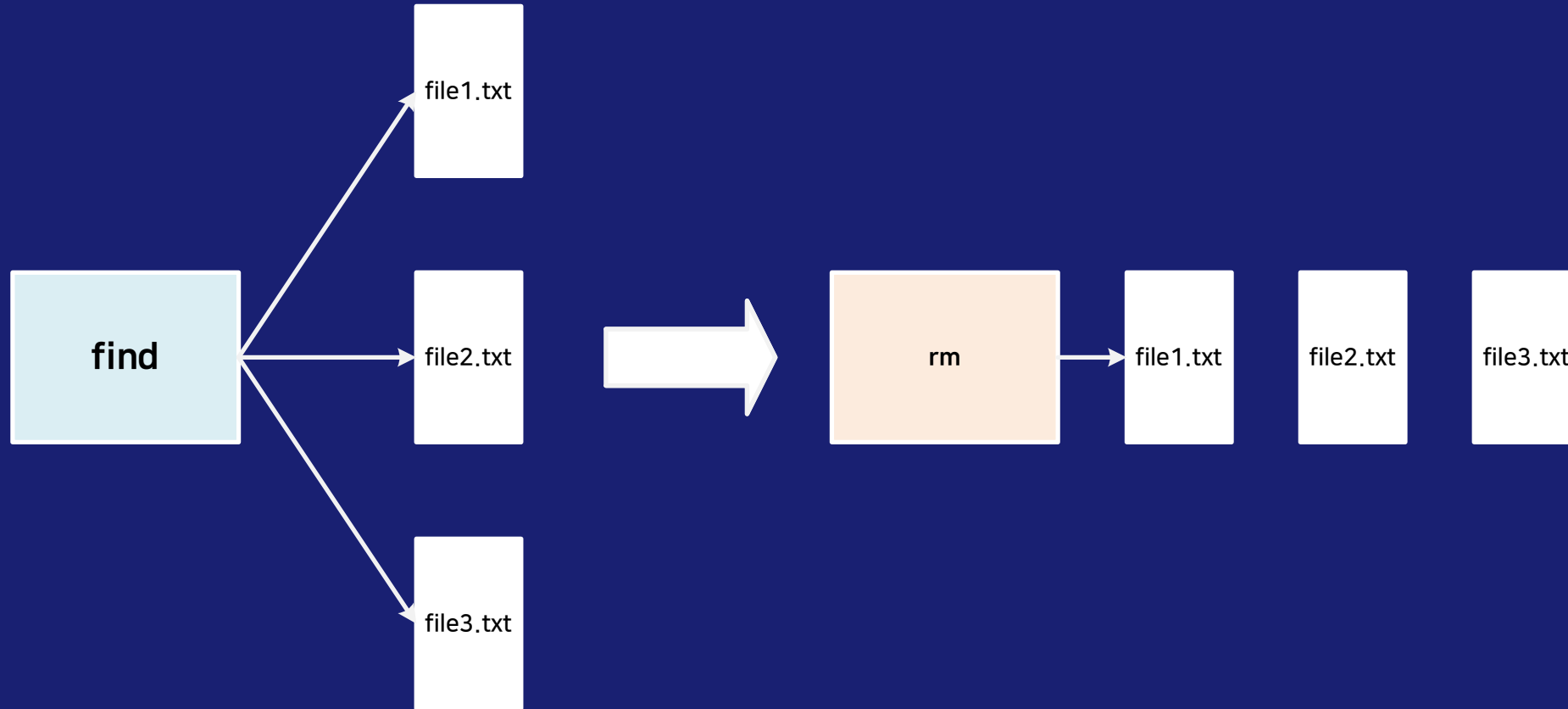
# find+



3개의 파일을 찾음.

find에서 대상을 찾을 때마다 1번씩 파일 삭제 명령어 수행, 총 3번 실행한다.

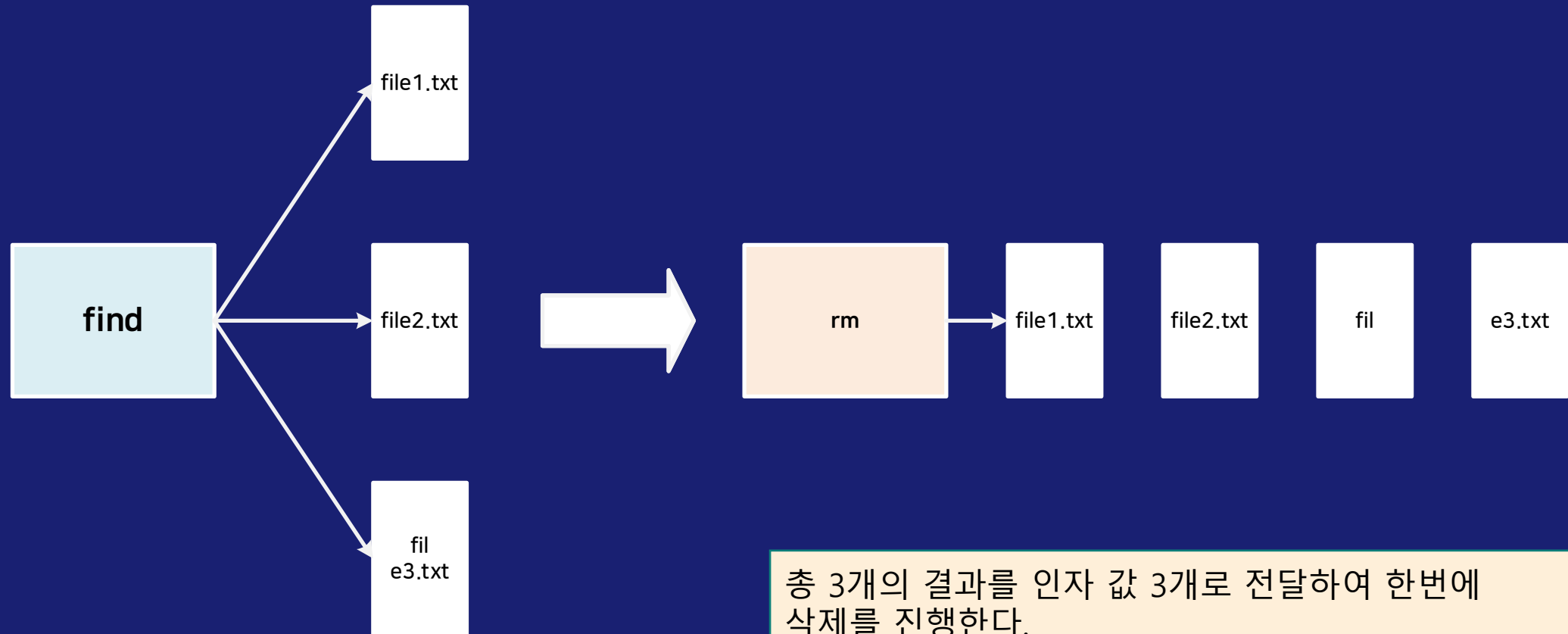
# find+



3개의 파일을 찾음.

총 3개의 결과를 인자 값 3개로 전달하여  
한번에 삭제

# find+



3개의 파일을 찾음.

총 3개의 결과를 인자 값 3개로 전달하여 한번에 삭제를 진행한다.  
하지만 이전과 다르게 "null string"이 있는 경우 다른 인자 값으로 판단하여 총 4개의 인자 값이 전달.

# find+

기본값: `find / -name test1.txt -type f -print`

위의 test1.txt에 "null string"이 존재.

널스트링: `find / -name test1.txt -type f -print 0`

`./test1.txt\0./test2.txt\0`



`./test1.txt`  
`./test2.txt`

빈 공간이 있는 경우, 이 부분을 문자열로 취급하여, 다른 결과로 인식 및 인지한다.

# find+

-print0 옵션을 사용하기 전.

find



file1.txt

fi      le2.txt

-print0 옵션을 사용하는 경우, 빈 공간이 있으면 이 부분 "null string"으로 처리한다.

find



./file1.txt\0./fi   le2.txt\0

# Null String vs Empty String

positive value



1



0



negative value



Infinity



NaN



null



undefined





# sort

'sort'는 출력된 결과를 알파벳 순서로 정렬 시 사용하는 문자이다.

```
$ sort /etc/shells
$ sort ~/file1 ~/file2
$ printf '%s\n' 'line 2' 'line 1' | sort
```

또한 같이 사용하는 옵션은 다음과 같다.

1. **-n**: 숫자가 알파벳 순서보다 먼저 출력이 된다.
2. **-r**: 역 정렬, 맨 마지막 결과가 맨 처음에 출력

```
$ cat ipaddress.txt
$ sort ipaddress.txt | uniq -c | sort -k1, 1nr
```

# 연습문제

여러 문자열을 화면에 시스템 변수와 함께 사용해서 처리한다. 출력 시, 사용하는 명령어는 아무거나 사용하여도 상관이 없다.

1. 호스트의 정보를 쉘 스크립트로 작성 후, 실행하면 다음과 같은 정보가 텍스트 파일 "result-system-info.txt"를 생성한다.
  - "This server name is <서버이름>. Also, This sever have IP Address <IP\_ADDRESSS>.
2. 호스트에서 사용중인 "sshd"프로세스를 찾아서 result-sshd-list.txt파일을 생성한다. 위치는 "/tmp/"디렉터리에 생성한다.
  - 'ps'명령어를 사용하여도 된다.
  - 'pgrep'를 통해서 찾아도 된다.
3. 'ls -al /usr/share/docs | grep -i readme | error'실행 시 발생하는 오류를 무시하고 함수 "listdocs"를 실행하도록 'listdocs.sh'파일을 생성한다.
4. ps명령어를 통해서 "cmd, pid, uid, gid"만 출력한다.
  - "-o"옵션을 통해서 출력하여도 된다.
  - 'awk'명령어를 통해서 출력하여도 된다.

# 연습문제

1. 'sleep'명령어를 약 1000초 동안 백그라운드로 동작하게 한다.
2. 가동중인 백 그라운드 프로세스를 bg로 확인한다.
3. 백 그라운드로 동작중인 프로세스를 다시 포그라운드로 올린다.
4. 사용자 procuser를 추가한 다음에 procuser로 로그인 한다.
5. procuser에서 'sleep 1000'명령어를 실행한다.
6. 'ps'명령어로 procuser의 프로세스 실행을 확인한다.
7. 확인이 완료가 되면 root에서 'pkill'명령어로 procuser를 종료한다.

# 기본 명령어

내부 명령어

# bash 기본 명령어

셸에서 기본적으로 제공하는 내부 명령어(internal command)가 있다. 이 명령어를 통해서 좀 더 편하게 셸 환경을 꾸밀 수 있다.

빌트인 명령어는 다음과 같으며, 실제로 몇몇 명령어는 외부 파일로 구현이 되어 있기도 하다. 그러니 외부에(명령어가) 있어도 너무 신경 쓰지 말자.

# 빌트인 명령어

빌드인 명령어는 아래와 같다. 아래 빌트인 명령어는 전부 암기할 필요는 전혀 없다. 또한, 빌트인 명령어는 구성에 따라서 외부 명령어로 제공이 되는 빌트인 명령어가 있다.

```
$ man -k built-in
```

```
alias, bg, bind, break, builtin, case, cd, command, compgen, complete,  
continue, declare, dirs, disown, echo, enable, eval, exec, exit, export,  
fc, fg, getopts, hash, help, history, if, jobs, kill, let, local, logout,  
popd, printf, pushd, pwd, read, readonly, return, set, shift, shopt,  
source, suspend, test, times, trap, type, typeset, ulimit, umask,  
unalias, unset, until, wait, while.
```

# 빌트인 명령어

내부 명령어는 외부에 별도의 명령어 없이 실행이 가능한 내장형 명령어. 대표적인 명령어는 다음과 같다.

**echo, type, source**

아래서 자세한 내용은 뒤에서 설명.

# 런타임 명령어

런타임 명령어(runtime), 기본적으로 셸 스크립트(bash)명령어 구성이 되어 있다.

여기 말하는 셸 스크립트 명령어는 'type', 'alias'같은 명령어 이며, 특정 시스템 명령어 'ls', 'cp'같은 명령어는 'alias'로 재구성이 되어 있기도 하다.

특정 명령어(혹은 확장기능)는(은) "**shell function()**"로 구성이 되어 자체적으로 실행은 불가능하며, 셸을 통한 조합 실행이 가능하다.



# 시스템 명령어

시스템 명령어는 파일 시스템에 독립적으로 구성이 되어 있는 프로그램. 어떠한 패키지가 설치가 되어 있느냐에 명령어가 조금 다를 수 있다. 아래는 대표적인 GNU/LINUX명령어 이다.

**grep, ping, rm, ls**

# 대표 리눅스 명령어

**type:** 명령어 형식 확인. 쉘에 alias로 구성이 되어 있는 경우 alias를 출력한다.

**echo:** 문자열 출력

**printf:** echo와 비슷하지만 포매팅 형식으로 출력

**pwd:** 쉘의 현재 위치 출력

**cd:** 쉘 위치 변경

**set:** 현재 쉘에 설정된 값을 출력

**declare:** 변수나 함수(function)를 관리하는 명령어

**test, [, [[:** 정규식 구성, 대괄호로 구성이 된다.

# 대표 리눅스 명령어

**ls**: 디렉터리 및 파일 출력

**mv**: 파일 이동 및 이름 변경

**cp**: 파일 복사

**rm**: 파일 및 디렉터리 제거, 다른 명령어로 rmdir이 있다.

**grep**: 문자열 검색하는 정규식 도구

**cut**: 행 단위로 자르는 명령어

**wc**: 줄 및 문자열 카운팅 도구

**find**: 파일이나 혹은 디렉터리를 특정 조건으로 찾는 도구

**sort, uniq**: 정렬 및 중복제거 하는 도구

# type

bash에서 명령어 사용 방법이 궁금한 경우 아래와 같이 사용이 가능하다.

```
bash$ help type
```

type명령어를 사용하면 다음과 같이 출력한다.

```
bash $ type cp
```

```
cp is /usr/bin/cp
```

```
bash $ type ls
```

```
ls is aliased to `ls --color=auto`
```

```
bash $ help cp
```

```
-sh: help: no help topics match `cp`. Try `help help` or `man -k cp` or `info cp`.
```

```
bash $ type for
```

```
for is a shell keyword
```

# type

function()를 type으로 확인하기. type명령어는 명령어가 바이너리인지 혹은 함수인지 확실하지 않으면, 호출하여 확인이 가능하다.

```
bash$ myfunc() { : ; }          ## myfunc() 선언
bash$ type myfunc               ## myfunc 확인
myfunc is a function
myfunc()
{
    :
}

bash$ alias myalias=:          ## alias 선언
bash$ type myalias              ## 선언된 alias내용 확인
myalias is aliased to `:`
```

# type

간단하게 명령어가 내부/외부 명령어 확인이 필요할 때, 아래처럼 실행한다.

```
bash$ type -a true
true is a shell builtin
true is /bin/true
```

# echo

간단하게 메시지 출력을 'echo' 명령어로 한다.

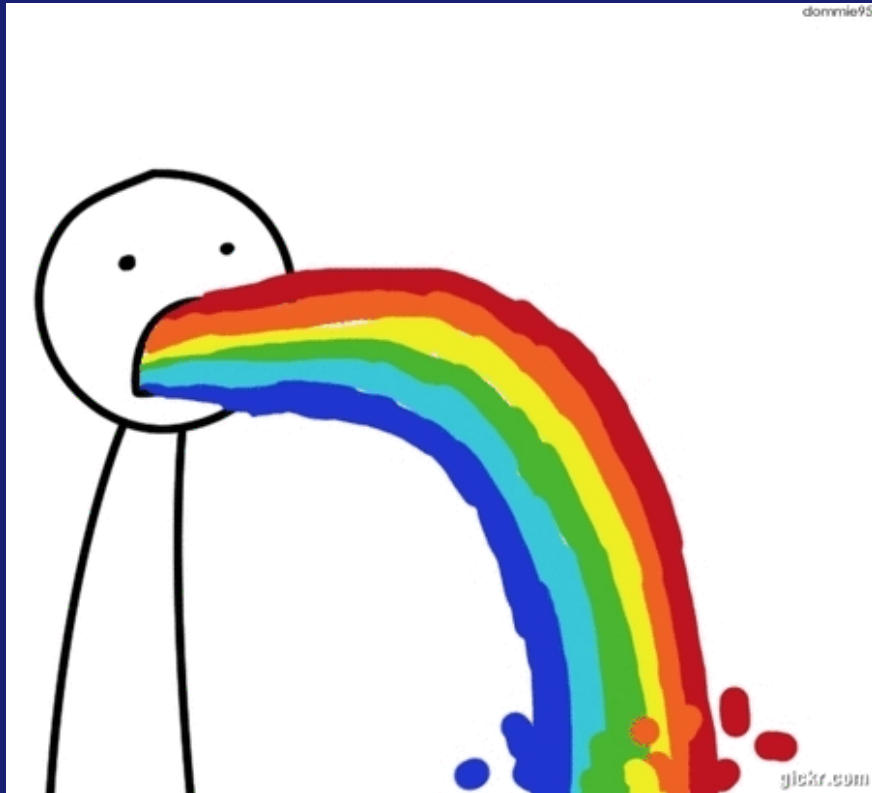
```
bash$ echo hello
```

```
hello
```

```
bash$ echo 'Hello, "$USER"'
```

```
Hello, bashuser!
```

# 기본 명령어



빌트인?

기본 명령어?

키워드 명령어?

????



# printf

'echo'명령어 대신 'printf'명령어 사용이 가능하다.

```
bash $ printf '%s\n' 'Hello!'  
Hello!
```

이 명령어는 C의 "printf"와 흡사한 포매팅 구조를 가지고 있다.

# echo vs printf

'echo', 'printf'명령어 둘 다 셸에서 사용이 가능하다. 셸 스크립트가 여러 유닉스 혹은 리눅스 계열에서 동작하는 경우, 'echo'로 출력되는 형식은 조금 다를 수 있다. 이를 해결하기 위해서 'printf'를 통해서 출력하는 경우, POSIX를 따르는 시스템에서는 일괄적으로 내용이 출력이 된다.

참고로 현재는 'echo'명령어도 표준을 따르나, 가급적이면 'printf'를 사용하도록 한다.

## echo

실행 시 "0"이외 리턴을 하지 못한다. 그러기에 출력에서 오류가 발생 하였는지 확인이 어렵다.

## printf

"0"이외 리턴이 가능하기 때문에 출력 오류에 대해서 확인이 가능하다. 또한 기본적으로 'printf'는 'echo'명령어보다 수행속도가 빠르다.

# echo vs printf

하지만, 쉘 버전이 업데이트 및 업그레이드가 되면서 현재는 'echo', 'printf'의 실행 속도에 큰 차이가 없지만 일반적으로 'printf' 명령어가 더 빠르다. 성능 테스트 하기 위해서 다음과 같이 명령어를 실행한다.

```
$ time for i in {1..999999}; do echo "$i" >/dev/null; done
real    0m20.095s
user    0m10.509s
sys     0m9.566s
```

# echo vs printf

```
$ time for i in {1..999999}; do printf "$i" >/dev/null; done  
real    0m18.851s  
user    0m10.115s  
sys     0m8.715s
```

# echo vs printf

'printf'명령어가 'echo'와 비슷하게 동작하기 위해서는 다음과 같이 실행한다.

```
$ time for i in {1..999999}; do printf "$i\n" >/dev/null; done  
real    0m19.614s  
user    0m10.678s  
sys     0m8.881s
```

# 빌트인?

몇몇 명령어는 시스템에 존재하지만, 실제로는 쉘의 빌트인 명령어. 다음과 같은 명령어로 확인이 가능하다.

```
# type echo
echo is a shell builtin
# whereis alias
alias: /usr/bin/alias /usr/share/man/man1/alias.1.gz
```

# 빌트인?

```
$ file /usr/bin/alias
```

```
/usr/bin/alias: a /usr/bin/sh script, ASCII text executable
```

```
$ cat /usr/bin/alias
```

```
#!/usr/bin/sh
```

```
builtin alias "$@"
```

# printf

대표적인 빌트인 명령어 사용 예제는 다음과 같다.

```
bash$ printf '%s\n' -n
-n
bash$ string=-n
bash$ printf '%s\n' "$string"
-n
```



# printf

```
$ printf '%s\n' foo bar baz
```

```
foo
```

```
bar
```

```
baz
```

# pwd

현재 작업하는 위치를 확인 시 사용하는 명령어.

```
$ pwd
```

```
/home/bash/working
```

```
$ ls secret/report.txt
```

```
$ ls report_review.txt
```

```
$ ls nonexistent
```

# tildes(~)

경로 관련된 "~"(틸드, tild)특수 문자. 이 문자를 통해서 홈 디렉터리로 이동이 가능하다. 틸드 path이외 다른 특수 문자 :, !!등 있지만, 보통 홈 디렉터리 이동 시 틸드를 더 선호 한다.

```
$ echo ~  
$ echo ~/secret  
# echo ~  
# echo ~root/.ssh
```

# tildes

만약 존재하지 않는 경우 그냥 문자열로 출력이 된다.

```
$ echo ~nouser  
~nouser
```

# tildes

특수 문자로 출력하고 싶은 경우 아래처럼 실행한다.

```
$ echo ~bash
```

```
/home/bash
```

```
$ echo \~bash ‘~bash’ “~bash”
```

```
~bash ~bash ~bash
```

# cd

cd는 change directory의 약자, 이를 통해서 디렉터리 이동을 한다. 이와 관련된 셸 변수는 \$PWD 혹은 'pwd' 명령어가 있다.

```
$ pwd
/home/bash
$ cd /tmp
$ pwd
/tmp
```

# cd

앞에서 언급하였던 .. 혹은 .를 통해서 디렉터리 이동이 가능함. 이미 많이 사용하는 명령어이니 그냥 넘어감. 😊

```
$ cd .
```

```
$ cd ..
```

```
$ pwd
```

# cd

조금 더 자세히 ".", ".."에 대해서 알고 싶으면 'stat'명령어로 확인해보면 디렉터리 관계 확인이 가능하다. 이들은 메타 디렉터리이며, 상위 inode와 연결 및 구성이 되어있다. 삭제는 불가능하며, 관리 용도로 사용한다.

```
$ stat .  
$ stat ..
```



## pushd/popd

'cd' 명령어와 같이 'pushd', 'popd' 명령어로 다음처럼 사용이 가능하다. 이 명령어를 사용하는 경우, 배시 셸 값에 이동 위치에 대한 기록이 남는다. 현재는 거의 사용하지 않으며, 스크립트 작성 시 기록 이동 용도로 가끔 사용한다.

```
$ sudo yum install httpd -y
$ pushd /var/www
$ pushd /opt/
$ pushd -n /usr/local/
```

# pushd/popd

"-n" 옵션이 붙어 있으면 실제로 디렉터리 이동하지는 않음. 내역을 확인하기 위해서는 다음과 같은 명령어로 확인 가능.

```
$ push -n /usr/local/  
$ dirs -l -v  
$ pushd +2  
$ pushd -1. == $ cd -
```

# pushd/popd

'popd'명령어로 이전에 사용하였던 하나씩 이동하면서 스택(**stack**)에서 하나씩 제거한다.

```
$ pushd /usr/local
$ pushd /opt/
$ pushd /var/www
$ popd
```

# alias

'alias'는 긴 명령어 혹은 명령어 "+" 옵션을 짧게 줄여서 사용하기 위해서 사용한다. 설정 되어있는 'alias'를 확인하기 위해서 다음 명령어로 확인이 가능하다.

```
$ alias move='mv -I'
$ alias Documents='/tmp/documents/'
```

'alias'설정은 다음처럼 한다.

```
$ alias
$ alias -p
```

해제하기 위해서는 다음 명령어로 'alias'해제가 가능하다.

```
$ unalias -a
$ unalias move
```

# function/alias

```
bash$ FirstFunction() { : ; }  
bash$ type FirstFunction  
FirstFunction is a function  
FirstFunction ()  
{  
    :  
}  
bash$ alias FirstAlias=:  
FirstAlias is aliased to `:`
```

## set 설명

'set' 명령어는 빌트인 명령어. 'set'은 현재 사용중인 셸에 설정되어 있는 값을 화면에 변수 이름과 함께 출력한다. 기존에 사용하던 값을 'set' 명령어를 통해서 설정 및 변경도 가능하다.

알파벳 순서대로 정렬해서 화면에 출력한다. 여기에 출력된 변수들은 전부 셸에서 명시된 값이다.

# set

옵션	설명
<b>declare</b>	변수의 속성을 정의하는 명령어.
<b>set</b>	배시 빌트인 명령어. 이 명령어를 통해서 변수 생성 및 제거를 한다.
<b>env</b>	배시의 변수 정보 확인.
<b>export</b>	특정 변수를 현재 세션이외에서도 사용이 가능하도록 설정한다.

# set

실제로는 'set'명령어보다 'declare'명령어 사용을 더 권장한다. 다만, 'declare'명령어는 변수 초기화는 지원하지 않기 때문에, 변수 선언은 'set'명령어로 선언해야 한다.

또한, 'declare'명령어는 'set'으로 선언된 변수 값은 변경할 수 없다. 이러한 차이점으로 변수 선언 및 접근 시 주의해야 한다. 'set'명령어는 "-v"옵션을 통해서 설정된 값 확인이 가능하다. 확인 하는 방법은 'printenv', 'env', 'compgen' 같은 명령어 사용을 권장한다.

```
$ declare -i Var1
$ declare -a Var2=("a", "b", "c")
$ declare -p Var1
$ declare -p Var2
```



## set -e(errexit)

"-e" 옵션은 "error exit"이다. 쉘 스크립트는 문제가 발생하면 0이 아닌 값을 반환하면서 종료한다. 하지만, "-e" 옵션을 사용하면, 오류가 발생하여도 무시하고 다음 명령어 혹은 함수를 실행한다.

하지만 이 방법을 스크립트에서 사용하는 경우 거의 대다수가 올바르지 않는 값을 반환한다. 그 이유는 "-e" 문제는 실행 결과값 예측이 어렵기 때문이다.

그래서 대다수 스크립트는 "set -e"를 사용하지 않고 자체적으로 오류 값 핸들링을 더 선호한다. 참고로 함수에서 반환은 'return' 종료는 'exit'로 처리한다.

## set -e(errexit)

```
$ nano set-e.sh
#!/bin/bash
set -e
function hello()
{
    ls -lR /usr/share/doc | grep readme
}
hello
$ sh -x set-e.sh
```

## set -e(errexit)

```
$ nano set-e.sh
#!/bin/bash
function hello()
{
    set -e
    ls -lR /usr/share/doc | greppp readme | grep readme
}
hello
$ sh -x set-e.sh
```

[https://www.gnu.org/software/bash/manual/html\\_node/The-Set-Builtin.html](https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html)

## set -e(errexit)

보통 아래와 같이 'trap'명령어와 같이 사용하는 것을 권장한다.

```
$ nano set-e.sh
#!/bin/bash
function hello()
{
    set -e
    echo "error occur at line $1"
}
trap 'hello $LINENO' ERR
errrrr
```

## set -n(noexec)

"-e"가 올바르게 동작하면 아래처럼 어느 라인에 문제가 있는지 메시지를 출력한다. 하지만, 해당 오류는 정확한 정보가 아닌 경우가 더 많다. 'trap'를 사용하지 않는 경우, 어느 부분에서 문제가 발생하였는지 확인이 어렵다.

```
$ sh set-e.sh
set1.sh: 줄 8: errrrrr: 명령어를 찾을 수 없음
error occur at line 8
```

## set -n(noexec)

명령어를 읽어 오지만 실행하지 않는다. 보통 셸 스크립트 문법 확인 시 사용한다.

```
$ nano set4.sh
#!/bin/bash
set -n
os="linux"
echo $os
echo $unix
```

## set -u(unset)

변수가 선언이 되었지만, 값이 비어 있는 경우, 경고 메시지 출력.

```
$ nano set5.sh
#!/bin/bash
set -u
os="linux"
echo $os
echo $unix
```

## set -f(noglob)

글러빙 끄기. 글러빙 기능을 비활성화 한다.

```
$ set -f
$ ls /usr/share/docs
$ set +f
$ ls /usr/share/docs
```



## set -a(allexport)

변수를 외부에서도 사용할 수 있도록 한다. 보통 셸은 다른 환경 혹은 PID가 다른 경우 이전 셸에서 사용하던 변수에 접근이 불가능 하다.

```
$ echo $$  
$ set -a  
$ number1=23  
$ number2=34  
$ bash  
$ echo $$  
$ echo number1
```

## set -v(verbose)

읽어오는 값을 그대로 화면에 출력한다. 디버깅시 매우 유용하지만, **-x**와는 다른 정보를 출력한다. 실행 시, 발생한 메시지를 자세히 출력한다.

```
$ bash -v set1.sh
```

## set -x(xtrace)

이 옵션은 각각 인자 값이 실행하는 걸 출력한다. 이 옵션을 실행하면 bash스크립트 디버깅에 도움이 된다. 어떤 부분에서 bash프로그램이 입력을 받아서 확장이 되는지 확인이 가능하다.

```
$ bash -x set3.sh
```

## set -x(xtrace)

디버깅시 다음과 같은 방법으로 진행이 가능하다. **set -x** 를 CLI에 같이 선언 혹은 매직-뱅(Magic-Bang)에 다음과 같이 선언한다.

```
#!/bin/bash -x
```

또한 set의 종료 문자는 “--”으로 사용이 가능하며, 이는 위치 값(positional parameter)로 사용이 가능하다.

## set -x(xtrace)

"-x"를 선언 전. 일반적인 결과만 출력한다.

```
$ nano set7.sh
#!/bin/bash
var = 5
while [ $var -gt 0 ] ; do
    var=$(( $var - 1 ))
    echo $var
    sleep 2
done
```

## set -x(xtrace)

```
$ nano set8.sh
#!/bin/bash
set -x
var = 5
while [ $var -gt 0 ] ; do
    var=$(( $var-1 ))
    echo $var
    sleep 2
done
```

## set -x(xtrace)

혹은 다음처럼 실행하여도 된다.

```
bash$ -x <SCRIPT>
```

## set -o(option-name)

셸 옵션을 설정한다. 앞에서 사용하였던 옵션들을, 이름으로 활성화한다. 예를 들어서 "errexit 옵션(-e)"를 활성화하기 위해서 다음과 같이 선언이 가능하다.

```
$ nano set6.sh
#!/bin/bash
set -o errexit
os="linux"
echo $os
echo $unix
```



# declare

앞에서 잠깐 사용하였던 declare는 내장 명령어이다.

이를 통해서 변수, 배열 혹은 함수를 구현한다. 변수 구현을 위해서 자주 사용하지는 않지만, 셸의 현재 상태 정보 및 다른 정보들 set를 통해서 훑어보기(overview)가 가능하다. 다만, 'set'으로 설정한 변수는 'declare'로 변경이 불가능하다.

# declare

```
$ declare -p | head -5
```

```
declare -- BASH="/bin/sh"
```

```
$ declare -r BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars:sourcepath"
```

## declare -r(readonly)

변수를 읽기 전용으로 선언. 읽기 전용으로 선언이 되면 해당 변수는 변경이 불가능하다.

```
$ declare -r test=ok
$ echo $test
ok
$ test=not-ok
-bash: test: readonly variable
```

# declare -i (integer)

정수형 변수를 선언한다. 소수점 선언은 불가능하다.

```
$ declare -i x=10
$ echo $x
$ x=ok
$ echo $x
0
```

## declare -i (integer)

```
$ x=15
```

```
$ echo $x
```

```
15
```

```
$ x=15+5
```

```
$ echo $x
```

```
20
```

```
$ x=15.1
```

# declare

이 명령어를 사용할 때 -f 옵션을 -p와 함께 사용하면 declare는 선언이 되어 있는 모든 함수에 대해서 출력한다.

**-f:** 쉘의 함수(function)들만 목록에 출력

**-p:** 변수의 속성(property) 출력

변수 속성을 제어 할 때, "+, -" 기호로 속성 제어가 가능. "+"는 활성화, "-"는 비활성이다.

# unset

선언된 변수를 메모리에서 해제한다. 'alias'도 비슷한 명령어를 가지고 있다.

```
$ unset -f <function_name>  
$ unset <value_name>  
$ unalias <alias_name>
```

# 긴 문자열 처리

만약, 긴 문자열을 화면에 출력해야 하는 경우, 보통 아래와 같이 핸들링 한다. 보통, 스크립트에서 메뉴나 혹은 설명과 같은 부분을 출력 시 사용한다.

```
$ longstring1.sh
#!/bin/bash
case $1 in
    -h | --help )
cat << 'EOF'
-h, --help: show this help
-q, --quiet: run without diagnostics
-v, --verbose: add extra diagnostics
EOF
        exit 0
    ;;
esac
```



# 긴 문자열 처리

긴 문자열 처리가 필요 할 때는 연속적으로 받기 위해서 입력 재출력을 통해서 "<<" 추가적으로 계속 받는다.

이때 어느 부분이 파일의 끝 부분인지 표시하기 위해서 'EOF'라는 키워드를 사용한다. "cat << 'EOF'" 명령어는 입력 라인에 EOF라는 문자열이 들어오면 더 이상 입력을 중지한다.

다른 문자열로 변경은 가능하나, 일반적으로 "EOF" 많이 사용한다.

# 긴 문자열 처리

```
$ longstring2.sh
#!/bin/bash
cat << EOF
Hello, $USER; you are running this program on $(hostname -s).
Your home directory is $HOME
EOF
```

# 긴 문자열 처리

```
$ longstring3.sh
#!/bin/bash
cat <<- 'EOF'
    leading tabs
        will not be included in the output
EOF
```

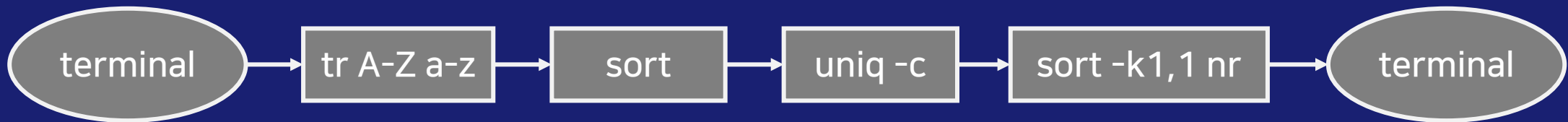
# 긴 문자열 처리

여러가지 명령어를 동시에 사용하기 위해서 셸 스크립트 형식으로 명령어를 작성한다. 이를 동시에 처리하기 위해서 파이프를 사용한다. 예제로 **tr**명령어 기반으로 간단하게 구성을 하였다.

# 긴 문자열 처리

위와 같이 사용하는 경우 쉘 스크립트처럼 여러 명령어를 사용하기 때문에 다음처럼 파이프로 구성하여 동시에 처리하도록 한다.

```
$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr
```



# 긴 문자열 처리

아래처럼 문자열을 입력 후, "ctrl+d" 누르면 아래와 같이 내용이 출력이 된다.

```
$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr
sdlkjskljieurwjldklskdj
akljsnkcjnnjhdsiakjslia
38473904889orjksdjm,snbns
1 ($&$
    1 2948
    1 38473904889orjksdjm,snbns
```

# 긴 문자열 처리

위에서 사용한 내용을 셸 스크립트로 변경하면 다음처럼 변경한다.

```
$ longstring4.sh  
#!/bin/bash  
tr A-Z a-z |  
sort |  
uniq -c |  
sort -k1,1nr
```

# 자르기(sprintf)

```
$ ta='type:json'
```

```
$ printf 'Type: %s\n' "${ta#type:}"
```

```
Type: json
```

```
$ ta='type:json'
```

```
$ printf 'Field name: %s\n' "${ta%:*}"
```

```
Field name: type
```



## 자르기(sprintf)

```
$ text='*WARNING'
```

```
$ printf '%s\n' "${text#\*}"
```

```
$ printf '%s\n' "${text#'\*'}"
```

```
$ path=/usr/local/test/bin/test
```

```
$ printf 'Removed the path as look like a file : %s\n' "${path##*/}"
```

```
Removed the path as look like a file: test
```

# 자르기(printf)

문자열 길이는 'wc'같은 명령어로 확인이 가능하다. 셸 빌트인 사용하는 경우 좀 더 쉽게 가능하다.

```
$ alpha='abcdefghijklmn'  
$ printf '%u\n' "${#alpha}"  
11
```

## 문자열 바꾸기(sprintf)

```
$ notice='I'\''ll do it today.'  
$ printf '%s\n' "${notice/today/tomorrow}"  
  
$ notice='Yes, today. I'\''ll do it today.'  
$ printf '%s\n' "${notice/today/tomorrow}"  
Yes, tomorrow, I'll do it today.  
$ printf '%s\n' "${notice//today/tomorrow}"  
Yes, tomorrow, I'll do it tomorrow.
```

## 문자열 바꾸기(sprintf)

```
$ notice='Yes, today. I\'\'ll do it today.'  
$ printf '%s\n' "${notice/today*/I\'ll do it soon.}"  
Yes, I'll do it soon.
```

# 대소문자 변경(printf)

```
$ text='helloWorld'  
$ printf '%s\n' "${text^^}"  
HELLOWORLD
```

```
$ printf '%s\n' "${text^}"  
HelloWorld
```

```
$ printf '%s\n' "${text,}"  
helloWorld
```

# 외부 변수 합치기 (printf)

```
#!/bin/bash  
temp="${test:-foo}"  
printf '%s\n' "${temp#.ext}"
```

# 연습문제

1. 'printf'명령어를 통해서 "Hello World" 문자열을 출력 합니다.
2. 'echo'명령어를 통해서 "Hello World2"문자열을 출력 합니다.
3. 'echo \$USER', echo '\$HOME', 'echo ~'명령어를 수행하여 차이점을 확인 합니다.
4. 'pushd'명령어를 사용하여 다음처럼 디렉터리를 이동한다.  
-> /tmp  
-> /var/tmp/  
-> /usr/share/doc  
-> /var/logs
5. 'popd'명령어를 통해서 어떻게 동작하는지 확인한다.
6. find명령어를 통해서 readme파일을 찾아서 전부 readme.tar로 묶은 후, /tmp 위치에 저장한다.
7. /bin디렉터리에 있는 목록을 sort명령어로 정렬 후, 결과에 /tmp/bin-sort.log에 저장한다.
8. grep명령어를 통해서 /usr/share/doc에서 Network라는 단어를 가지고 있는 파일 목록을 /tmp/grep-filelist.log에 저장한다.

# 조건문

LOOP/CONDITION



# 조건문

셸에서 조건문을 사용하기 위해서 다음과 같은 조건이 필요하다. 아래 나와 있는 셸 조건은 빌트인 셸 조건문이다.

1. `if`
2. `[`, `[[`
3. `((`
4. `case`
5. `for`
6. `while`

# 조건문

if문을 사용하기 위해서는 다음과 같은 방법으로 구현이 가능하다.

```
$ grep -q bash /etc/shells && printf "bash is a system shell\n"
```

먼저 중간에 있는 "&&"기호는 뒤에 두번째 명령어를 실행하는데, 이를 "**and 연산자**"라고 부르기도 한다. "-q" bash 문자열이 /etc/shells에 있으면 true를 반환하면, 그 다음 명령어 printf를 실행한다.

만약, 해당 문자열이 없으면 false반환이 되기 때문에 뒤에 명령어가 실행이 되지 않는다. 반대로 "&&"가 아닌 "||"도 있으며, 이를 "**or 연산자**"라고 부르기도 한다.

```
$ git commit || echo "Commit failed"
```

# 조건문

위의 내용을 배시 if문으로 변경하면 다음처럼 된다.

```
$ if grep -q bash /etc/shells ; then printf 'bash is a system shell\n' ; fi
```

좀 더 읽기 편하게 하기 위해서 다음처럼 구현이 가능하다.

```
if grep -q bash /etc/shells ; then  
    printf 'bash is a system shell\n'  
fi
```

셸은 두 가지 방식으로 조건문 사용이 가능하다.

1. 인라인 방식
2. 쉘 스크립트 파일 방식

# 조건문

if 명령어는 반드시 **fi**로 종료를 해야 한다. 파일로 작성 시, 다음과 같이 작성한다.

```
$ if1.sh
if cd /usr/local ; then
    printf ">&2 'changed directory\n'
fi
```

# 조건문

부정적인 조건은 아래처럼 사용이 가능하다. 간단한 조건문은 대 괄호로 감싸지 않아도 된다. 아래와 같은 방식은 실제로 사용하지 않는 걸 권장한다.

```
$ if2.sh
if ! cd /usr/local/ ; then
    printf ">&2 'Failed to changed directory\n'
    exit 1
fi
```

# 조건문

if문에서 조건이 맞지 않았을 때 동작하는 else문법을 구현하기 위해서는 **elif** 키워드를 사용한다.

```
$ if4.sh
if cd /usr/local/mysql ; then
    printf >&2 'Changed to primary directory'
elif cd /opt/mysql ; then
    printf >&2 'Changed to secondary directory'
else
    printf >&2 'Cound\'\'t find a directory!'
    exit 1
fi
```

# 조건문 지사자

위와 같이 조건문을 사용하는 경우는 간단하게 `true`, `false`인 경우만 사용이 가능하다. 고급 적이고 복잡한 조건문을 사용하는 경우 아래 페이지와 같은 조건문들 사용이 가능하다.

옵션	설명
[ <code>-a FILE</code> ]	파일이 존재하면 참.
[ <code>-b FILE</code> ]	파일이 존재하고 블록파일이면 참.
[ <code>-c FILE</code> ]	파일이 존재하고 특별한 캐릭터 파일이면 참.
[ <code>-d FILE</code> ]	파일이 존재하고 그리고 디렉터리가 존재하면 참.
[ <code>-e FILE</code> ]	파일이 존재하면 참.
[ <code>-f FILE</code> ]	파일이 존재하고 일반 파일이면 참.

# 조건문 지사자

옵션	설명
[ -g FILE ]	파일이 존재하고 SetGID가 설정이 되어 있으면 참.
[ -h FILE ]	파일이 존재하고 심볼 링크면 참.
[ -k FILE ]	파일이 존재하고 Sticky이면 참.
[ -p FILE ]	파일이 존재하고 FIFO이면 참.
[ -r FILE ]	파일이 존재하고 읽기전용이면 참.
[ -s FILE ]	파일이 존재하고 파일 크기가 0보다 크면 참.



# 조건문 지사자

옵션	설명
[ -t FD ]	파일이 FD(file Descriptor)이며, 터미널이 열려 있으면 참.
[ -u FILE ]	파일이 존재하고 SUID가 설정이 되어 있으면 참.
[ -w FILE ]	파일이 존재하고 쓰기가 가능하면 참.
[ -x FILE ]	파일이 존재하고 실행이 가능하면 참.
[ -O FILE ]	파일이 존재하고 사용자가 소유하고 있으면 참.
[ -G FILE ]	파일이 존재하고 그룹 사용자가 소유하고 있으면 참.
[ -L FILE ]	파일이 존재하고 심볼릭 링크인 경우.
[ -N FILE ]	파일이 존재하고 마지막으로 읽은 이후 수정된 경우 참.
[ -S FILE ]	파일이 존재하고 소켓 파일인 경우.

# test

if 키워드 대신, test라는 도구를 사용할 수 있다. 그냥 test명령어만 사용하는 경우 다음처럼 사용하면 된다.

```
$ yourshell=bash
$ test "$yourshell" = 'sh' && printf 'DingDong!\n' echo -ne '\007\'
$ test "$yourshell" = 'bash' && printf 'DingDong!\n' echo -ne '\007\'
```

# 조건문

**-e:** file이라는 디렉터리나 파일이 있으면 참.

```
$ test -e <FILENAME>  
$ echo $?
```

**-n:** \$vars라는 변수가 선언이 되어 있고 비어 있지 않는 경우.

```
$ test -n "$vars"  
$ echo $?
```

위에서 이야기한 조건문 지시자를 test명령어로 그대로 사용이 가능하다.

# if

'if'문은 다음과 같이 사용한다. 좀 더 복잡한 형태로 조건문을 사용하려면 대괄호 기반으로 위에서 언급한 조건문을 사용한다.

```
if [ -f /var/log/message ]  
then  
    echo "The log file exist!"  
fi
```

## if elif else fi

'if'문에서 'else if'는 'elif'키워드로 사용이 가능하다. 또한 'else', 'elif'를 사용하는 경우 반드시 'then'키워드를 같이 사용해야 한다.

```
if [ condition ]; then
    elif [ other_condition ]; then
    else
fi
```

# if elif else fi

```
$ vi elifelse.sh
#!/bin/bash
seconds=$1
if [ "$seconds" -eq 0 ]; then
    timezone_string="KST"
    echo $seconds
elif [ "$seconds" -gt 0 ]; then
    timezone_string=$(printf "%02d:%02d" $((seconds/3600)) $(((seconds / 60) % 60)))
    echo $timezone_string
else
    echo "Unknown parameter"
fi
```

# while/until

'until'은 'while'과 비슷하지만, 조금 다른 부분은 코드가 실행된 후에 거짓이면 계속 실행 참이면 반복문을 종료한다.

```
$ until.sh
#!/bin/bash
COUNTER=20
until [ $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

# for

'for' 반복문은 카운터 변수에 입력된 개수만큼 동작한다. 중지되는 기준은 "End Of Array"이다. 아래는 ls명령어를 실행하여, 출력된 결과를 변수 i에 대입하여 화면에 출력한다.

```
$ for.sh
#!/bin/bash
for i in $(ls); do
    echo item: $i
done
```



# 루프 및 조건문

**-d:** 디렉터리 이름이 dir인 경우

```
$ test -d /tmp  
$ echo $?
```

**-s:** 파일이 존재하며 파일이 비어 있지 않는 경우

```
$ test -s /bin/ls  
$ echo $?
```

## test/if([, [[)

test명령어는 "[" 명령어. "[" 명령어는 두 개의 명령어가 있어야 올바르게 동작한다. "[" , "]"이 두개의 명령어는 쉘 스크립트에서 많이 사용하며, 또한 이것들은 그냥 명령어이다. 쉘 스크립트의 키워드와 다르다.

```
$ [ "$yourshell" = 'bash' ] && printf 'Correct!\n'
```

```
#
```

```
# test명령어는 if로 사용이 가능하다.
```

```
#
```

```
myshell=bash
```

```
if [ "$myshell" = 'bash' ] ; then
```

```
    printf 'Correct!\n'
```

```
fi
```

## test, [, [[

변수 보안 및 내부 키워드 심볼릭 변수를 사용하기 위해서는 '[' 명령어로 처리해 한다. 가급적이면 새로운 형식으로 사용한다. 이 방식은 배시 셸 3.0 이후부터 지원한다.

```
#!/bin/bash
yourshell=bash
if [[ $yourshell = 'bash' ]] ; then
    printf 'correct!\n'
fi
```

## test, [, [[

"[, "["하고 제일 큰 차이점은 다음과 같다. 첫번째 예제는 일반적인 예제이기 때문에 보기에는 크게 차이가 없다.

다만, 두번째 예제는 "-o"옵션을 사용하여 둘 중 하나가 참이면 작업이 수행이 되지만, "["으로 처리하는 경우, 간단하게 "yes","y"문자열이 포함이 되어 있으면 참으로 핸들링 한다.

```
if [ "$myshell" = 'bash' ] ; then
if [[ $yourshell = 'bash' ]] ; then
```

```
if [ "$answer" = y -o "$answer" = yes ]
if [[ $answer =~ ^y(es)?$ ]]
```

"["으로 표현하는 경우, 변수를 심볼 형태로 처리가 가능하며, "[" 경우에는 변수를 쌍 따옴표로 감싸야 한다.

## 결론(test, [, [[)

구체적인 차이는 다음과 같다. 배시 셸은 버전 2,3버전을 거쳐가면서 조건문에 대한 버그가 있는지 확인이 되지 않았다.

"["강화 버전은 "["이며, 강화가 된 이유는 이전 bash 3.x때 치명적인 변수 버그로 파싱 부분에 대해서 강화가 필요 하였다. 이러한 이유로 "["방식으로 사용을 권장한다.

```
if [ -f "$file" ] -> if [[ -f $file ]]
```

셸 버그에 대해서는 아래 링크에서 확인이 가능하다.

[https://en.wikipedia.org/wiki/Shellshock\\_\(software\\_bug\)](https://en.wikipedia.org/wiki/Shellshock_(software_bug))

## test, [, [[

새로운 방식으로 적용하면 아래와 같다.

```
yourshell=$SHELL  
[[ $yourshell = "$yourshell" ]]  
echo $?
```

하지만, 기존 방식하고 혼동이 될 수 있기 때문에 그냥 이전에 사용하던 방식으로 사용해도 문제가 없다. "[" 방식과 비슷하지만, "["에서 크게 문제 없이 동작한다.

```
[ "$yourshell" = "$yourshell" ]
```

# test, [, [[ 글러빙

기존에 사용하였던 **test**, "["처럼 "["에서도 조건문 사용이 가능하다.

```
[[ $yourshell = b* ]] ; echo "I am okay"
```

위와 같이 실행하는 경우 이전 앞에서 사용한 글러빙처럼 "b"로 시작하는 문자열을 찾는다. 혹은 아래와 같은 방법으로 실행이 가능하다.

```
echo [[ $yourshell = b* ]] ; echo "I am okay"
```

## test, [, [[

혹은 다음 "=~" 새로운 지시자(operator)를 사용해서 시작 문자를 지정할 수 있다.

```
[[ $yourshell =~ 'sh$' ]]  
if [[ $yourshell =~ 'ba' ]] ; then echo "okay" ; fi
```

위의 명령어를 실행하면 마지막 문자가 sh로 끝나는 찾아서 화면에 출력한다.



## test, [, [[

복잡한 오퍼레이션 사용이 필요한 경우 좀 더 간단하게 표현이 가능하다.

```
if [ "$answer" = y -o "$answer" = yes ] ; then echo $answer ; fi
if [[ $answer =~ ^y(es)?$ ]] ; then echo 'the answer is "Yes"' ; fi
```

이전에 사용이 가능하였으나, 사용하기 조건이 복잡하였던 **틸드(~)**도 파싱 및 사용이 가능하다.

# 산술조건(Arithmetic condition)

POSIX기준으로 **'test'**, **"["**명령어를 사용해서 산술 연산자 사용이 가능하다. **"["**으로 처리하는 경우, 굳이 쌍 따옴표를 명시하지 않아도 된다.

- [ "\$a" -eq "\$b" ]: 같다
- [ "\$a" -ne "\$b" ]: 같지 않다
- [ "\$a" -lt "\$b" ]: 작거나
- [ "\$a" -le "\$b" ]: 작거나 혹은 같거나
- [ "\$a" -gt "\$b" ]: 크거나
- [ "\$a" -ge "\$b" ]: 크거나 혹은 같거나

# 산술조건(Arithmetic condition)

count변수가 "100"이면, echo명령어를 실행한다. 이때 -eq를 사용하여 변수와 그리고 비교 값이 같은 지 확인한다.

```
count=100
if [ $count -eq 100 ]
then
echo "Count is 100"
fi
```

# 산술조건(Arithmetic condition)

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

```
(( 2 > 0 )) : True, exit 0
((a = 1)): True, exit 0
(( 0 >3 )): False, exit 1
(( 0 )): False, exit 1
((a = 0 )): False, exit 1
```

```
$ (( 2 > 0 )) && echo $?
$ ((a = 1)) && echo $?
$ (( 0 >3 )) && echo $?
$ (( 0 )) && echo $?
$ ((a = 0 )) && echo $?
```

# CASE

CASE문은 조건에 맞는 값이 들어오면, 해당 함수나 혹은 명령어를 실행한다. SystemV에서 사용하는 init 셸 스크립트를 통해서 프로세스를 제어한다. 제어하는 방식으로 셸 스크립트 CASE문법을 사용한다. 문법 사용법은 매우 간단하다.

```
case $type in
    args1)
        function
        ;;
    *)
        default
        ;;
esac
```

# CASE

```
case $1 in
    start)
        systemctl start httpd
        ;;
    stop)
        systemctl stop httpd
        ;;
    *)
        default
        ;;
esac
```

# 연습문제

다음과 같은 조건으로 조건문을 구성 및 완성한다. 작성 방식인 인-라인 혹은 쉘 스크립트 파일 형식으로 작성하여도 상관이 없다.

1. BASH 시스템 변수를 사용해서 /bin/bash 경로와 다르면, "This is not BASH"라는 메시지를 출력한다.
2. value1, value2의 값에 각각 20라는 숫자를 입력 후, 같은 값인지 비교한다.
3. 쉘 스크립트 이름을 helloworld.sh라고 만든 후 다음과 같은 내용을 출력한다.
  1. 쉘 스크립트 자신의 이름.
  2. 인자 값 총 6개를 받아서 화면에 인자 값 출력.
4. if, test, && 중 아무거나 사용하여, /etc/vsftpd/vsftpd.conf파일이 존재하는지 확인.
  1. 존재하지 않으면 "the vsftpd package doesn't installed"라고 메시지 출력.
  2. vsftpd 패키지가 없으면 자동으로 설치가 되어야 한다.
  3. 설치된 서비스가 실행이 안되고 있으면, 서비스 자동으로 실행이 되어야 한다.

# 함수

함수 선언 및 활용



# 함수 설명

함수는 명령어를 기능별로 묶을 때 사용하는 기능. C언어의 "void function(){} return 0"과 비슷한 기능.

함수는 일반적으로 여러가지 명령어를 사용하여 하나의 세트로 만든 다음에 그 기반으로 동작 순서(**routine**)를 구성한다. 앞에서 간단하게 함수를 사용하였지만, 여기서 직접적으로 간단한 함수 구성 및 선언하여 활용한다.

함수는 다음과 같은 구조로 생성한다. 중괄호 사용시 띄어쓰기에 주의하자.

```
function HelloWorld() {  
    echo "Hello World"  
}
```

호출은 다음과 같은 명령어로 함수를 호출한다.

```
HelloWorld
```

# 함수 포맷

셸에서 함수 선언 방법은 두 가지 방법이 있다.

1. 인라인 함수 선언
2. 파일 함수 선언

인라인 함수 선언은 잘 사용하지 않지만, 보통 셸에서 다음과 같이 설정한다.

```
foo() { echo "this inline foo\n"; }
```

혹은 다음과 같은 양식으로 파일로 작성이 가능하다.

```
foo(){  
    echo "this is a normal function formatting in script file"  
}
```

# 함수 호출

함수를 호출하기 위해서는 다음과 같이 호출한다. 앞서 작성한 코드 기반으로 호출 방법을 명시한다.

```
$ foo
```

파일로 작성하는 경우, 파일 안에서 함수 이름을 호출해야 한다. 보통 파일 밑에 함수 이름을 적는다.

```
$ vi foo.sh
foo(){
    echo "this is a normal function formatting in script file"
}
foo
$ sh foo.sh
```

# 함수 예약 값

함수에는 다음과 같은 예약 값이 있다.

인자 값	역할
\$0	함수 이름과 위치가 전달이 된다. 이 값은 쉘 함수에 예약된 변수 값이다. \$0으로 호출이 가능하다.
\$1, \$2, etc.	1~9번까지 호출이 가능하다. 입력 순서에 따라서 저장 및 호출이 된다. 구분은 스페이스로 처리한다. 호출 방법은 변수에 \$1, \$2, \$3...와 같이 불러온다.
\$#	전달받은 인자 값 만큼, 인자 개수를 가지고 있다.
\$@ and \$*	전달받은 인자 값의 인자 목록을 가지고 있다.
"\$@"	리스트로 전달 받은 값을 \$1, \$2형태로 가지고 있다.
"\$*"	전달 받은 문자열을 스페이스로 분리하여 인자 값을 저장한다. 위와 동일하게 \$1, \$2... 형태로 저장한다.

# 함수 전환/종료

```
$ vi ret-exit-func.sh
retfunc(){
    echo "This will be good run as a function"
    return 1
}
exitfunc(){
    echo "This will exit when after run it"
    exit 1
}
retfunc
echo $?
exitfunc
echo $?
```

# 함수 외부인자

함수는 외부에서 인자 값 혹은 변수를 가져와서 처리가 가능하다. 외부 변수 및 인자 값 처리 방법은 다음과 같다.

```
$ vi extfunc.sh
function foo(){
    echo ${1}
    echo ${2}
    echo ${3}
    local loVar=${1}
    echo ${loVar}
}
foo $1 $2 $3
```

# 함수 외부인자

다음과 같은 방법으로 외부에서 값을 받아서 변수에 사용자가 입력한 값으로 설정이 가능하다. 이때 사용하는 키워드 명령어는 'read'명령어 이다.

```
$ vi showyourname.sh
#!/bin/bash
read -p "Enter your Family Name: " FM_NAME
read -p "Enter your Last Name: " LT_NAME

function showyourname(){
    echo "${FM_NAME} ${LT_NAME}"
}

showyourname
```

# 간단한 함수 구현

```
#!/bin/bash  
  
function HelloWorld() {  
    echo "Hello World"  
}  
  
HelloWorld
```



# 함수 확인

생성한 함수를 셸에서 확인 하고 싶은 경우, 다음명령어로 확인이 가능하다. 아래 명령어를 수행하기 위해서는 반드시 위의 명령어 작업이 수행이 되어야 한다.

```
$ declare -F <function>  
$ delclare -F HelloWorld
```

함수 생성 후 함수를 제거(unset)하고 싶은 경우 다음과 같은 명령어로 함수 선언을 해제한다.

```
$ unset <function>  
$ unset HelloWorld
```

# Variable in Function

함수에서 변수 선언은 매우 단순한다. 다만, 여기에서 `local`이라는 내부 키워드 명령어를 사용하여, 내부 변수로 값을 제정의 한다. 즉 함수 외부에서는 '1', '2'는 그대로 호출이 되며, 내부에서는 `local`키워드 명령어로 다시 선언이 된 "var1", "var2"는 값 "5"로 변경이 된다. 다만, 이 변수는 "change()"안에서만 적용이 된다.

```
#!/bin/bash
var1=1
var2=2
function change(){
    echo internal function
    echo "Variable 1 is: $var1"
    echo "Variable 2 is: $var2"
    local var1=5
    local var2=5
    echo
    echo "After change internal function"
    echo Variable 1 is locally $var1
    echo Variable 2 is gobally $var2
}
```

# Function + local

```
function change(){  
    echo internal function  
    echo "Variable 1 is: $var1"  
    echo "Variable 2 is: $var2"  
    local var1=5
```

local은 function안에서만 동작한다. 셸 에서  
다음과 같이 명령어를 실행한다. 어떻게 동작하는가?  
**\$ local var1=5**

# 함수 인자

bash에서 함수 인자 값은 다음처럼 다룬다. 쉘 스크립트에서 인자 값을 처리하는 방법은 다음과 같다. 하지만, 이 모든 걸 굳이 외울 필요가 없다. 필요할 때 그때 그때 필요한 함수 처리 방법을 가져와서 사용하면 된다.

1. `$`: 예약된 쉘 함수 인자 값. 보통 `$0`번은 명령어(스크립트)실행 위치를 표시
2. `$1, $2, etc`: `$1~$9`번까지는 일반적으로 받는 변수 이름
3. `$#`: 현재 가지고 있는 인자 값 출력
4. `$@, $*`: 둘 다 기능은 비슷하다. 차이점은 리스트이냐 혹은 문자열이냐 차이다. 분리 기호는 일반적으로 띄어쓰기(space)로 구별한다.
5. `$@`: 문자열을 리스트 형태로 변환한다. 스페이스 기반으로 리스트 배열을 구성한다.

# 함수 인자

위의 내용을 가지고 간단하게 쉘 스크립트를 작성 후, 직접 인자 값 테스트를 해본다.

1. 함수 a는 입력된 값을 하나씩 호출하여 출력.
2. 함수 b는 입력된 인자 값을 리스트로 출력한다. 차이점은 이 방식은 문자열로 구별한다.
3. 함수 c는 입력된 인자 값을 리스트로 출력한다. 차이점은 띄어쓰기로 입력 값을 구별한다.

```
$ vi bash_arguments.sh
#!/bin/bash
function a {
    echo "-$1-" "-$2-" "-$3-";
}
function b {
    a "$@"
}
function c {
    a "$*"
}
$ bash_arguments.sh "I love you" "I hate you" "What the hell"
```

# 종료코드

bash에서 사용하는 리턴 코드는 보통 다음과 같다. 종료 코드는 여러 프로그램에서 많이 사용하기 때문에, 간단하게 눈에 익혀두면, 어떤 문제로 프로그램 실행이 종료가 되었는지 유추가 가능하다.

코드	설명	예제
1	일반적인 오류 코드.	let 5/0, cp, \$ ls -l /root/
2	빌트인 키워드 올바르게 찾지 못함, 예를 들어서 올바르지 않는 함수.	
126	명령어 실행이 되지 못함. 예를 들어서 /dev/null를 명령어로 접근.	cat /dev/null
127	명령어를 찾을 수 없음. \$PATH나 혹은 오타 typo)	5/0, lls, noexistcommand
128	올바르지 않는 리턴 값. 예를 들어서 exit 3.14159.	return (3.14)
128+n	예를 들어서 프로그램 종료 시 kill -9한 경우 128+9.	kill -9 <PID>
130	스크립트 종류. 예를 들어서 ctrl+c 128코드에 +2.	
255	프로그램이 종료가 되었으나, 범위에 포함이 되지 않음.	

# 연습문제

간단하게 다음과 같은 함수를 구현 후, 화면에 내용을 출력한다.

1. 함수의 이름은 "userinfo"로 생성한다.
  - 사용자 이름은 변수 "uname"으로 저장한다.
  - 사용자의 나이는 변수 "uage"으로 저장한다.
  - 사용자의 국적은 변수 "unational"으로 저장한다.

2. 입력은 순서대로 받으며, 받은 내용을 다음과 같은 텍스트로 출력한다.

```
Hello <uname>. You age is <uage> and nationality is <unational>.
```

# 활용하기3

외부 변수 입력

GTK/TUI



# 리소스 입력

입력 방법

# 리소스 입력

bash에서 사용자에게 변수 입력 방법은 일반적으로 read를 통해서 사용자에게 변수를 입력 받는다. 코드는 아래와 같다.

```
echo -n "Enter your name: "  
read yourname
```

```
read -p "Enter your name: " yourname  
echo $yourname
```

```
printf "%s" "Enter your name: "  
read yourname
```

# 리소스 입력

혹은 select명령어를 통해서 메뉴 구성이 가능하다. 예제는 아래와 같다.

```
PS3="Select your a httpd task please: "  
select t_cmd in check start failed exit delete install  
do  
    case $t_cmd in  
        delete)  
            delete_httpd  
            ;;  
        install)  
            install_httpd  
            ;;  
    esac  
done
```

# 리소스 입력

PS3는 select명령어를 사용할 때 출력되는 프롬프트(prompt)이다. 위의 코드를 실행하면, 셸 프롬프트는 "Select your a httpd task please: "으로 출력이 된다. selec문법은 보통 case와 함께 사용한다.

```
PS3="Select your a httpd task please: "
```

# Dialog Menu

**!!주의!!** 아래는 코드임!! :)

```
dialog --backtitle "여기는 뒤에 TUI패널" --menu "메뉴 선택" 10 40 3 1 "야호" 2  
"잘생기고 싶다"
```

위의 명령어를 실행하면, 좌표에 명시된 위치에 메뉴를 생성한다. dialog는 패키지가 설치가 안되어 있기 때문에 실행하기 위해서는 아래 명령어로 설치한다.

```
dnf install dialog -y
```

# GTK Menu

GUI가 필요한 경우, freedesktop에서 표준으로 사용하는 GTK+기반으로 GUI화면 구성이 가능하다. 코드는 아래와 같이 간단하게 구성이 가능하다.

사용하기 위해서는 zenity패키지 설치가 필요하다.

```
# dnf install zenity -y
# dnf install langpacks-ko -y
```

# GTK Menu

```
#!/bin/bash
items=("못" "생겼다" "아니요...")
while item=$(zenity --title="$title" --text="$prompt" --list --column="Options" "${items[@]}")
do
    case "$item" in
        "${items[0]}") echo "못생김 $item, item #1";;
        "${items[1]}") echo "못생김 $item, item #2";;
        "${items[2]}") echo "못생김 $item, item #3";;
        *) echo "Ooops! Invalid option.";;
    esac
done
```

# 응용하기

TUI도구 구성하기

간단한 쿠버네티스 관리도구 만들기



# 랩 문제

활용 셀 스크립트

# 랩 문제1

셸 스크립트 기반으로 다음과 같은 스크립트를 구성한다. **httpd**, **vsftpd**, **mariadb**를 설치 및 서비스 관리하는 스크립트를 생성한다. 조건은 아래와 같이 적용이 된다.

- 모든 패키지는 쉘 메뉴를 통해서 설치가 진행이 된다.
- 설치가 진행이 되면, 방화벽 및 서비스를 구성한다.
  - `systemctl`
  - `firewall-cmd`
- 서비스가 올바르게 동작하는지 쉘 메뉴에서 확인이 가능해야 한다.
- 모든 이벤트(동작)들은 `logger`를 통해서 시스템 로그에 기록을 남긴다.
- 위의 모든 기능은 `function`기반으로 구성한다.

## 랩 문제 2

컨테이너 기반의 웹 서비스를 관리하는 셸 스크립트를 작성. 컨테이너 런타임 설치 및 구성하는 명령어는 다음과 같다.

- `dnf install podman -y`
- `mkdir /root/htdocs`
- `echo "Hello shell training" > /root/htdocs/index.html`
- `podman run -d --rm -p 8080:8080 -v /root/htdocs:/var/www/html/ --name apache quay.io/centos7/httpd-24-centos7`
- `podman ps`
- `curl http://localhost:8080`

## 랩 문제 2

- 위의 컨테이너 명령어 기반으로 함수 구성해야 한다.
  - 최소 한 개 이상의 함수를 사용해서 구성한다.
  - 최소 한개의 function {}를 사용해서 작업 구성
- 포드만 설치부터 웹 파일 및 컨테이너 실행.
  - podman 패키지는 반드시 설치가 되어야 한다.
- 컨테이너가 실행하면서 웹 페이지를 출력하기 위해서 반드시 "/root/htdocs/"에 "index.html"파일이 존재해야 됨.
  - index.html파일은 "Hello shell training"의 내용을 가지고 있어야 됨.
- 문제없이 실행이 되면 관리메뉴에서 확인 시 "the container service is okay"라고 메시지 출력
  - 최소 조건문 if, test, &&를 통해서 메세지 출력
  - 출력되는 메세지는 사용자가 자유롭게 지정
- 작업 수행 및 확인 시
  - 각각 작업이 완료가 될 때, "작업완료"라고 메세지 출력
  - curl명령어로 웹 페이지 확인 되면 "작업완료"라고 메세지 출력
  - grep를 통해서 웹 페이지 문자열 확인

# 앤서블 모듈

셸 스크립트 응용

# 설명

앤서블 모듈은 파이썬 언어 기반으로 작성 권장. 하지만, 이전에 사용하던 스크립트를 바로 앤서블 플레이 북이나 혹은 파이썬 모듈로 변경이 어렵기 때문에, 셸 기반으로 모듈 확장 혹은 수정.

여러 언어들을 사용하여 모듈을 만들 수 있지만, 간단한 지식을 통해서 빠르게 만들 수 있는 언어는 아마 셸 스크립트가 유일하다.

**Perl, Python, Go Lang, PHP**같은 언어로 만들려고 하는 경우 다시 언어에 대해서 기본적인 학습이 필요하기 때문에 그러한 과정을 최대한 줄이고 간결하게 사용할 수 있도록 리눅스에서 많이 사용하는 bash script 기반으로 작성한다.

# vimrc

```
$ vi ~/.vimrc
```

```
autocmd FileType yaml setlocal et ts=2 ai sw=2 nu sts=0
```

```
set cursorline
```

# 설명

앤서블 모듈은 다음과 같은 인자 값 핸들링 한다.

```
sh.check_agent:  
  name: cisco  
  type: network
```

위와 같은 구조를 가지고 있다.



# 설명

셸 변수 핸들링은 앞에서 학습 하였던 방식으로 입력을 받아온다.

```
$ source $1  
$ echo $1 $2
```

1, 2번째 인자 값을 셸을 통해서 전달 받는다

# 출력

앤서블 모듈은 다음과 규칙이 적용이 된다.

## changed

성공적으로 작업이 완료가 된 경우 출력되는 메시지. 올바르게 수행이 되면 **true**, 수행이 되지 않으면 **false**으로 반환한다.

## failed

작업 수행이 실패한 경우, 해당 작업에 대해서 실패하면 true를 반환한다.

## msg(ok)

성공/오류/변경 같은 작업이 문제없이 수행이 되는 경우 메시지를 반환 합니다.

# 출력

위의 기능은 JSON기반으로 동작하면 메시지는 다음과 같은 형식으로 전달 합니다.

```
echo '{“changed”: true, “msg”: “Updated some thing”}'
```

문제는 쉘에서는 JSON핸들링을 직접 하는 방법이 없기 때문에, 다음과 같은 방법으로 출력을 처리한다.

# 출력 시 주의사항

하지만, 쉘 스크립트에서는 특수문자 핸들링 하기 위해서는 다음과 같이 두 가지 해결 방법이 있다.

```
echo "{\“changed\”: true, \“msg\”: \“Updated some thing\”}"
```

혹은 다음과 같이 bash에서 "printf"를 사용하여 출력이 가능하다.

```
printf '{\“changed\”: true, \“msg\”: \"%s\"}' \"$msg"
```

셸에서 JSON형태로 출력 시 위의 같은 형태로 구현만 해주면 된다.

# 출력 시 주의사항

하지만, bash또한 JSON형태로 출력이 불가능하기 때문에 파이썬 같은 도구를 사용해서 출력 도움을 받아야 한다.

```
msg="{\"changed\": true, \"msg\": \"$msg\"}"
```

각 상태 메시지(\$msg)는 앤서블에서 변수 전달하기 때문에, 보통은 변수 전달 및 핸들링을 권장한다.

```
echo $(echo "$msg" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
```

더 이상 위의 방법을 사용하지 않아도 된다. 하지만, 여전히 출력 표준은 JSON기반을 권장하고 있다.

# 출력

간단하게 정리하면, **echo**, **printf** 둘 중 아무거나 사용하여도 됨. 올바르게 JSON 형태로 출력하면 된다.

# 간단한 문법

앤서블 모듈 작성시 꼭 필요한 기본 함수. 명시된 함수는 없으나 모듈에서 사용하는 옵션 그대로 함수를 생성 해주어야 함.

예를 들어서 다음과 같은 옵션이 있으면 다음처럼 함수를 만들어야 함.

```
backup:
  src: /usr/share/docs
  dest: /tmp/docs.tar
  label: 20220202_backup_ansible
```

# case문 예외처리

이 문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
case $type in
    network)
        network
        ;;
    os)
        os
        ;;
    *)
        echo "default"
        ;;
esac
```



# 앤서블 모듈 예제 및 함수 구성

```
sh.check_agent:
```

```
  name: cisco
```

```
  type: network
```

```
function network{}
```

```
function os{}
```

# 코드 내려받기

`https://raw.githubusercontent.com/tangt64/training\_memos/main/opensource/bash-101/ansible-modules-examples/sh.agent\_check`

# 코드 구성

```
#!/bin/bash
function network
{
if [ $type == "network" ] && [ $name == "cisco" ]; then
    $result_cisco=$(ps -ef | grep cisco | awk '{ print $NF }' | uniq | wc -l)
    if [ result_cisco -gt 1 ]; then
        contents=$(printf "$result_cisco" | python2 -c 'import json,sys; print json.dumps(sys.stdin.read())')
        output=$(printf "$result_cisco" 2>&1 | python2 -c 'import json,sys; print
json.dumps(sys.stdin.read())')
        changed="false"
        msg="the cisco agent is alive"
    fi
}
```

# 코드 구성

```
else
    msg="the cisco agent is dead"
    output=$(printf "$result_cisco" 2>&1 | python2 -c 'import json,sys; print
    json.dumps(sys.stdin.read())')
    error="Heelo"
    echo "{ \"failed\": true, \"msg\": \"$msg\", \"output\": \"$output\" }"
    exit 1
fi
}
```

# 코드 구성

```
source $1 $2
if [ -z "$type" ]; then
    printf '{"failed": true, "msg": "missing required arguments: name"}'
    exit 1
fi

changed="false"
msg=""
contents=""
```

# 코드 구성

```
case $type in
    network)
        network
        ;;
    os)
        os
        ;;
    *)
        printf '{"failed": true, "msg": "invalid type: %s"}' "$type"
        exit 1
        ;;
esac
printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"
exit 0
```

# 모듈 테스트

모듈 테스트 하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

```
$ ansible localhost, -m sh.check_agent -a 'name=cisco type=network'
```

# 모듈 테스트

디버깅 하기 위해서는 bash의 -x옵션을 통해서 디버깅이 가능하다.

```
$ echo 'name=cisco type=network' > args  
$ bash -x bashmod args
```