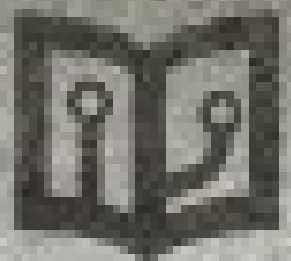


BASH

Shell Script

smug896

Published
with GfBook



차례

| | |
|--------------------------------------|-------|
| Introduction | 0 |
| DISQUS | 1 |
| Basics | 2 |
| Quotes | 2.1 |
| Escape Sequences | 2.1.1 |
| Variables | 2.2 |
| Functions | 2.3 |
| Exit Status | 2.4 |
| Pattern Matching | 2.5 |
| #! | 2.6 |
| Interactive vs Non-Interactive Shell | 2.7 |
| Login vs Non-Login Shell | 2.8 |
| Arrays | 3 |
| Test | 4 |
| Operators | 4.1 |
| Subshells | 5 |
| Commands | 6 |
| Builtin Commands | 6.1 |
| read | 6.1.1 |
| printf | 6.1.2 |
| eval | 6.1.3 |
| Keyword Commands | 6.2 |
| Compound Commands | 7 |
| Special Expressions | 8 |
| Shell Metacharacters | 9 |
| Precedence | 9.1 |
| Expansions and Substitutions | 10 |
| Brace Expansion | 10.1 |
| Tilde Expansion | 10.2 |
| Parameter Expansion | 10.3 |

| | |
|---------------------------|------|
| Arithmetic Expansion | 10.4 |
| Command Substitution | 10.5 |
| Process Substitution | 10.6 |
| Word Splitting | 10.7 |
| Filename Expansion | 10.8 |
| Redirections | 11 |
| Pipe | 11.1 |
| Named Pipe | 11.2 |
| File Descriptors | 11.3 |
| Buffering | 11.4 |
| <<< , << | 11.5 |
| Job Control | 12 |
| Session and Process Group | 12.1 |
| Process State Codes | 12.2 |
| TTY | 12.3 |
| Signals and Traps | 13 |
| kill | 13.1 |
| trap | 13.2 |
| Signals Table | 13.3 |
| Shell Options | 14 |
| Shell Variables | 15 |
| Positional Parameters | 15.1 |
| Special Parameters | 15.2 |
| Command Aliases | 16 |
| Command History | 17 |
| Command Completion | 18 |
| Readline | 19 |
| Debugging | 20 |
| Dash | 21 |
| Colors and Prompt | 22 |
| Tips | 23 |
| Recommand Sites | 24 |

Introduction

Unix 에 shell 이 처음 등장했을 때는 단순히 사용자에게 명령을 입력받아 실행시키는 interactive interpreter 에 불과 했습니다. 그후에 script 기능이 추가되고 command history, alias, tab completion, extended scripting syntax 가 추가됩니다.

shell 의 기본적인 기능은 명령을 실행시키는 것입니다. 그러므로 script 기능은 부가적인 기능으로 볼 수 있습니다. script 에서 사용되는 if, else 문 for, while 문을 보면 프로그래밍 언어를 보는것 같지만 사실은 script 기능을 위해 shell 에 추가된 키워드에 불과합니다.

shell script 에서는 보통 프로그래밍 언어에서 처럼 코드를 작성할 수 없습니다. 물론 shell 에서도 script 실행시에 메타문자, 키워드, 특수표현식을 자체적으로 해석하지만 기본적으로 명령문을 작성하는 기준에 맞지 않으면 오류가 발생합니다. 이와같은 shell 이 가지는 특수한 환경을 인지하고 있어야 왜 공백하나 때문에 아무 문제없는 코드에서 오류가 나는지 알수 있습니다.

Shell 은 OS 를 사용하게 될때 제일 먼저 접하는 부분이고 기초가 되는 부분인데도 불구하고 전반적으로 정리가 되어있는게 없는것 같아서 나름대로 정리해 보았습니다. 본문 중에 잘못된 부분 (내용오류, 오타) 이 있거나 이해가 잘 안되는 문장이 있으면 알려주시기 바랍니다. 또한 좀 더 좋은 책을 만들기 위해 첨가, 수정하면 좋을 내용이 있으면 알려주세요.

Ubuntu 15.04 에서 Bash version 4.3 를 이용하여 설명하였습니다.

Comments

Shell script basics

다음은 Shell script 를 오류 없이 작성하기 위해 기본적으로 알고 있어야 하는 내용들입니다. 특히 **단어분리**와 **globbing** 은 변수와 **명령치환**을 quote 하지 않고 사용할때 항상 주의해야 합니다.

파일명

파일명은 곧 명령을 실행할때 사용되는 이름과 같습니다. 리눅스에서 사용하고 있는 파일시스템에서는 파일 이름으로 `NUL` , `/` 두 문자를 제외하고 전부 허용한다고 합니다. 그러므로 다음과 같은 스트링은 모두 명령이름이 될수 있습니다.

```
[ , [10 , [[AA , {echo , {AA=10}
```

공백

shell 의 기본적인 기능은 명령을 해석해서 실행하는 것입니다. 명령의 기본적인 구조를 살펴보면 다음과 같습니다.

```
command arg1 arg2 arg3 ...
```

제일 앞에 명령 이름이 오고 다음에 '공백' 그다음 첫번째 인수 '공백' 두번째 인수 '공백' ... 이렇게 명령문은 기본적으로 공백으로 분리하여 작성합니다. 만약에 공백을 사용하지 않으면 어떻게 될까요? 다음과 같이 되어 명령이 정상적으로 실행되지 않을것입니다.

```
# 이경우는 'commandarg1' 가 명령이 된다.
commandarg1 arg2 arg3 ...

# 이경우는 'arg1arg2' 가 첫번째 인수가 된다.
command arg1arg2 arg3 ...
```

별로 중요할것 같지 않은 이 개념이 shell script 를 작성할때 자주 오류의 원인이 됩니다. if 문에서 주로 사용하는 `[` 는 키워드 같이 생겼지만 shell 에서 builtin 으로 제공하는 명령으로 `test` 와 동일한 명령입니다. 다만 차이점은 마지막에 인수로 `]` 를 붙인다는것 입니다.

```
# [ 명령과 첫번째 인수 10 을 붙여 사용하여 오류가 발생
$ [10 -eq 10 ]; echo $?
[10: command not found

# 마지막 인수 ] 를 10 과 붙여사용하여 오류 발생
$ [ 10 -eq 10]; echo $?
bash: [: missing ']'

# [ 명령에서 사용되는 연산자들도 모두 인수에 해당합니다.
# 다음의 경우 인수들 사이에 공백을 두지않아 a=b 가 하나의 인수로 인식이 됩니다.
# 그러므로 스트링 "a=b" 과 같은 의미가 돼서 항상 참이 됩니다.
$ [ a=b ]; echo $?
0

# 인수들 사이에 모두 공백을 띄워서 정상적으로 실행되었습니다.
$ [ a = b ]; echo $?
1

-----

# '{' 키워드와 echo 명령을 붙여 사용하여 '{echo' 가 명령 이름이 되었습니다.
$ {echo 1; echo 2 ;}
bash: syntax error near unexpected token '}'

# 공백을 사용하여 정상적으로 실행되었습니다.
$ { echo 1; echo 2 ;}
1
2
```

shell 에는 위에서 살펴본 기본적인 명령구조를 갖지 않는 문장이 하나 있는데 바로 대입연산 입니다. 대입연산을 하는 문장을 명령을 작성하는 식으로 하면 오류가 발생합니다.

```
# 변수 AA 가 명령이 되고 = , 10 는 각각 인수로 인식됩니다.
$ AA = 10
AA: command not found

# 그러므로 shell 에서 대입연산은 반드시 공백 없이 붙여 사용해야 합니다.
$ AA=10
$ echo $AA
10
```

shell script 를 작성할때 대입연산을 제외하고 모두 공백을 두어 작성하는 것도 오류를 줄일 수 있는 방법이 될 수 있습니다.

참 과 거짓

if 문에서 참, 거짓을 판단할때 프로그래밍 언어에서는 0 이 거짓이고 그외 값은 참이지만 shell script 에서는 반대입니다. 0 만 프로그램의 정상종료를 나타내어 참이되고 그외 숫자는 오류를 분류하여 나타내는데 사용되므로 거짓이 됩니다. 그리고 판단에 사용되는 데이터도 명령의 종료 상태 값 \$? 으로만 합니다.

\$? 은 프로그램의 종료 상태 값을 나타내는 shell 에서 제공하는 특수 변수입니다.

```
### 종료 상태 값 0 이 참이고 그 외는 거짓이다.

$ date -%Y      # 인수를 잘못 사용하여 오류발생
date: invalid option -- '%'
Try 'date --help' for more information.

$ echo $?      # 0 이 아닌 종료값은 if 문에서 모두 거짓에 해당합니다.
1

$ date +%Y
2015

$ echo $?      # 정상종료 됐으므로 0 을 리턴. if 문에서는 참이 됩니다.
0

-----

# 전달받은 인수값을 그대로 리턴하는 함수. '$1' 는 첫번째 인수를 나타냄
$ f1() { return $1 ;}

# 'f1 0' 은 종료값으로 0 을 리턴하므로 참
$ if f1 0; then echo true; else echo false; fi
true

# 'f1 1' 은 종료값으로 1 을 리턴하므로 거짓
$ if f1 1; then echo true; else echo false; fi
false
```

return

shell 함수에서 사용되는 return 명령은 프로그래밍 언어와 달리 연산 결과를 반환하는데 사용되지 않습니다. shell script 를 종료할때 exit 명령을 이용해 종료 상태 값을 지정하는 것처럼 함수에서는 return 명령을 사용해 종료 상태 값을 지정합니다.


```
$ myfunc() { expr $1 + $2 ; return 5 ;}

$ AA=$(myfunc 1 2)

$ echo $?      # $? 는 명령의 종료 상태 값을 나타내는 특수 변수
5

$ echo $AA     # expr 명령의 출력값이 연산 결과 값이 된다.
3
```

명령 종료 문자

c/c++, java 같은 언어에서는 문장의 종료를 나타내기 위해 마지막에 항상 `;` 를 붙여야 하지만 shell script 에서는 반드시 문장 끝에 붙일 필요는 없습니다. 왜냐하면 라인개행을 알아서 인식하기 때문인데요. 하지만 개행을 하지 않고 명령들을 한 줄에 연이어 쓸 경우는 반드시 `;` 를 붙여야 합니다. 특히 주의할 점은 명령 grouping 을 위해 중괄호 `{ }` 사용시는 명령의 인수와 구분을 위해 마지막에 `;` 를 붙여줘야 합니다.

```
# 명령들을 한줄에 연이어 쓸 경우는 ';' 를 사용해야 한다.
$ for i in {1..3} do echo $i done
> 오류

$ for i in {1..3}; do echo $i; done
1
2
3

# 'echo 2 }' 하면 '}' 까지 프린트된다. 인수와 구분을 위해 ; 를 붙여야 한다
$ { echo 1; echo 2 }
> 오류

$ { echo 1; echo 2 ;}
1
2
```

소괄호 `()` 는 shell 에서 제공하고 해석되어지는 subshell 메타문자로 공백, `;` 제약 없이 사용할 수 있습니다. 프로그래밍 언어에서처럼 자유롭게 사용할 수 있는 메타문자입니다.

```
$ (echo hello; echo world)
hello
world
```

Escape

Shell 에서 사용되는 명령문에는 단지 명령문을 위한 스트링만 존재하지 않습니다. script 작성을 위해 shell 에서 제공하는 키워드, 메타문자, glob 문자들이 같이 사용되는 환경이기 때문에 만약에 명령문에서 동일한 문자가 사용된다면 escape 하거나 quote 하여 명령문을 위한 스트링으로 만들어 줘야 오류가 발생하지 않습니다.

```
# 명령문에 shell 에서 사용하는 glob 문자 '*' 가 사용되어 에러 발생
$ expr 3 * 4
expr: syntax error

# 다음과 같이 quote 하거나 escape 하여 명령문을 위한 스트링으로 만들어줌
$ expr 3 '*' 4
$ expr 3 \* 4
12

# '<' , '>' 문자는 shell 에서 사용하는 redirection 메타문자
# 마찬가지로 escape 하지 않으면 정상적으로 실행되지 않고 오류가 발생합니다
$ [ a \< b ]
$ test a \> b
$ expr 3 \> 4

# '(' ')' ';' 문자도 shell 에서 사용하는 메타문자
$ find . -type f ( -name "*.log" -or -name "*.bak" ) -exec rm -f {} ;
bash: syntax error near unexpected token '('

# 다음과 같이 모두 escape 해줘야 오류없이 정상적으로 실행이 됩니다.
$ find . -type f \( -name "*.log" -or -name "*.bak" \) -exec rm -f {} \;
```

단어분리

이것은 shell 이 가지는 고유의 기능 중 하나인데 변수나 명령치환을 quote 하지 않으면 값이 출력될때 IFS (Internal Field Separator : 기본적으로 공백문자로 구성) 에 의해 단어가 분리됩니다. 그러므로 뜻하지 않게 인수가 2개 이상으로 늘어난다거나 공백이 포함된 파일이름이 분리가 되거나 하는 오류가 발생할 수 있습니다. Expansions and Substitutions -> Word Splitting 에서 좀 더 자세히 다룹니다.

```
$ f1() {
    echo arg1 : "$1"
    echo arg2 : "$2"
}

$ AA="hello world"

$ f1 "$AA"
arg1 : hello world
arg2 :

# $AA 변수를 quote 하지 않아 f1 함수에 전달한 인수 개수가 2 개가 되었다.
$ f1 $AA
arg1 : hello
arg2 : world

-----

$ AA="hello world"

$ ARR=( "$AA" )
$ echo ${#ARR[@]}
1

# $AA 변수를 quote 하지 않아 단어가 분리되어 ARR 원소 개수가 2 개가 되었다.
$ ARR=( $AA )
$ echo ${#ARR[@]}
2
```

Filename Expansion (Globbing)

Shell에서는 파일을 선택할때 기본적으로 glob 문자를 이용합니다. 그러므로 변수나 명령치환을 quote 하지 않고 사용할때 출력되는 값에 glob 문자가 있으면 뜻하지 않게 globbing 이 발생해 오류가 발생할 수 있습니다. Expansions and Substitutions -> Filename Expansion 에서 좀 더 자세히 다룹니다

```
$ AA="User-Agent: *"      # 변수 AA 값으로 glob 문자 '*' 가 사용됨

$ echo "$AA"             # quote 을 하면 globbing 이 발생하지 않음
User-Agent: *

$ echo $AA               # quote 을 하지 않아 globbing 이 발생해 뜻하지 않은 값이 출력됨
User-Agent: 2013-03-19 154412.csv Address.java address.ser
ReadObject.class ReadObject.java robots.txt 쉘 스크립트 테스트.txt
WriteObject.class WriteObject.java
```

Misc.

명령의 옵션

위에서 명령의 기본구조 예를 들때 사실 빠진 부분이 하나 있습니다. 바로 명령 옵션인데요. 옵션은 `-` 로 시작하고 하나의 문자만 사용하는 `-E` 과 같은 간단버전과 `--` 로 시작하고 여러개의 문자를 사용할수 있는 `--extended-regexp` 긴버전 두종류가 있습니다.

| Short form | Long form |
|--------------------------------|-----------------------------|
| <code>-o</code> | <code>--option</code> |
| <code>-o value</code> | <code>--option=value</code> |
| <code>-oltr</code> 합쳐서 쓸수있다 | X |
| 사용하기 간편하다 | 이름을통해 옵션의 의미를 알기쉽다 |

그런데 이 옵션때문에 명령을 실행할때 오류가 발생할 수 있습니다. 다음은 `grep` 명령을 이용해 현재 디렉토리 이하 파일들로부터 `-n` 스트링을 찾는 것인데요. 그런데 이때 `-n` 이 검색할 스트링이 아니라 `grep` 명령의 옵션으로 인식이 되어 정상적으로 명령이 실행되지 않습니다.

```
$ grep -r '-n'      # '-n' 을 옵션으로 인식해 정상적으로 실행되지 않는다.
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

이와같은 경우 `--` 를 사용하여 "이 뒤로부터는 옵션이 아니다" 라고 선언할 수 있습니다.

```
# '--' 는 end of options 을 나타냄

$ grep -r -- -n      # '--' 를 이용하여 '-n' 은 옵션이 아님을 선언
...

$ compgen -A file -- $cur
```

그러므로 script 작성시 명령의 인수로 변수를 사용할때 값으로 `-` 문자로 시작하는 스트링이 올 수 있을 경우 `--` 를 이용해 분리해야 오류를 줄일 수 있습니다.

```
command -o val -- "$arg1" "$arg2"
```

cd 명령은 종료 값을 확인해야 한다.

다음과 같은 경우 만약에 `cd` 명령이 실패하였다면 현재 디렉토리가 모두 삭제됩니다.

```
cd ~/tempdir
rm -rf *
```

그러므로 `cd` 명령의 성공 여부를 체크해서 사용해야 합니다.

```
cd ~/tempdir && rm -rf *

# 뒤에 이어지는 명령이 많을 경우
cd ~/tempdir || exit 1
...
cd ~/tempdir || { echo "cd ~/tempdir failed"; exit 1; }
...
```

Quotes

Shell 에서 quotes 은 숫자나 스트링 값을 구분하기 위한 용도로 사용하지 않습니다. 123 은 "123" 과 같고 abc 는 "abc" 와 차이가 없으며 모두 다 shell 에서는 스트링입니다. 그럼 quotes 은 어디에 무슨 용도로 사용하느냐 하는 것인데요. shell 에서 quotes 은 라인개행이나 공백문자를 유지하거나 no quote 시에 발생하는 [단어분리](#), [globbing](#) 같은 처리를 제한하는 용도로 사용됩니다. 이와 같은 기능은 echo 나 printf 명령에 속한것이 아니라 quotes 이 가지고 있는 독립적인 기능으로 명령문 어디에서 사용되더라도 동일하게 동작합니다. (echo 나 printf 명령은 [escape 문자](#)를 처리합니다.)

```
----- test.sh -----
#!/bin/bash

echo arg1 : "$1"
echo arg2 : "$2"

-----

$ ./test.sh 111 "111"      # quote 을 한것과 하지않은 것은 차이가 없다
arg1 : 111
arg2 : 111

$ ./test.sh hello world   # 두개의 인수를 나타낸다.
arg1 : hello
arg2 : world

$ ./test.sh "hello world" # quote 을 하면 한개의 인수를 나타냄
arg1 : hello world
arg2 :
```

특수 기능을 갖는 문자들

다음 세 문자는 명령행 상에서 특수한 기능을 가집니다.

| 문자 | 기능 |
|----|----------------------------|
| \$ | 매개변수 확장, 산술 확장, 명령 치환에 사용 |
| ` | 명령 치환에 사용 (backtick) |
| ! | history 확장에 사용 (프롬프트상 에서만) |

```
$ AA=hello

$ echo $AA world `date +%Y`    # $AA 변수가 확장이 되고 date 명령치환이 되었다.
hello world 2015

# 특수문자를 escape 할 경우
$ echo \$AA world `date +%Y`
$AA world `date +%Y`
```

특수기능을 갖는 문자나 단어를 escape 하는 방법

shell에서는 escape 할때 `\` 문자 외에 quote 을 사용할 수 있습니다. quote 을 하면 특수기능이 없어지고 단순히 명령문을 위한 스트링으로 사용됩니다.

```
# ( ) ; shell 메타문자를 quote 하여 기능을 상실. find 명령을 위한 스트링이 되었다.
# \ ( \) \; 한것과 같습니다.
$ find . -type f '(' -name "*.log" -or -name "*.bak" ')' -exec rm -f {} ';'

# backgroun job 을 생성할때 사용되는 & 메타문자를 quote 하여 기능을 상실.
# \& 한것과 같습니다.
$ echo hello '&'
hello &

# escape 문자인 \ 을 quote 하여 기능을 상실. 결과적으로 \n 가 되었다.
# \\n 한것과 같습니다.
$ echo hello world | tr ' ' '\n'
hello
world

# grep 명령에 설정돼 있는 alias 가 escape 되었다.
# \grep 한것과 같습니다.
$ 'grep' 2015-07 data.txt
...

# shell 키워드인 time 이 escape 되어 외부명령인 /usr/bin/time 이 실행되었다.
# \time 한것과 같습니다.
$ 'time'
Usage: time [-apvV] [-f format] [-o file] [--append] [--verbose]
        [--portability] [--format=format] [--output=file] [--version]
        [--quiet] [--help] command [arg...]
```

No Quote

명령행 상에서 공백은 인수들을 구분하는데 사용됩니다. 둘 이상의 공백은 의미가 없으므로 하나의 공백으로 대체됩니다. no quote 에서는 공백도 escape 할 수 있습니다. 공백을 escape 하면 두 개의 인수가 하나가 됩니다. (참고로 인수를 구분하는데 사용되는 공백문자와 IFS 값과는 상관이 없

습니다.)

```
----- test.sh -----
#!/bin/bash

echo arg1 : "$1"
echo arg2 : "$2"
-----

$ ./test.sh hello world          # 인수가 2개
arg1 : hello
arg2 : world

$ ./test.sh hello\ world        # 공백을 escape 하여 인수가 하나가 되었다.
arg1 : hello world
arg2 :

$ echo hello          world      # 둘 이상의 공백은 하나로 줄어든다.
hello world

$ echo hello\ \ \ \ \ \world
hello      world
```

No quote 상태에서는 shell 메타문자, shell 키워드, glob 문자, alias, space, tab, newline 모두를 escape 하여 기능을 disable 할 수 있습니다.

No quote 상태에서 **escape 문자** 사용예.

```
# 모든 문자가 escape 되므로 t n 이 echo 명령에 전달된다.
$ echo -e foo\tbar\n123
footbarn123

# 다음과 같이 하면 \t \n 이 echo 명령에 전달되어 escape 문자가 처리된다.
$ echo -e foo\\tbar\\n123
foo    bar
123
```

! history 확장 escape

```
$ date
Sat Jul 18 00:06:41 KST 2015

$ echo hello !!
echo hello date      # 이전 명령 history 확장
hello date

$ echo hello \!!     # escape 하여 history 확장 기능 disable
hello !!
```


\ 문자 escape

```
# no quote 상태에서는 모든문자가 escape 되므로 'tr : n' 와 같아진다.
$ echo 111:222:333 | tr : \n
111n222n333

$ echo 111:222:333 | tr : \\n
111
222
333
```

" , ' quote 문자 escape

```
$ echo double quote \" , single quote \'
double quote " , single quote '
```

행의 마지막에 \ 를 붙이고 개행을 하면 \newline 과 같이 되어 newline 을 escape 한 결과를 같습니다. (\ 뒤에 다른 문자가 오면 안됨)

```
$ echo "I like \
> winter and \
> snow"

I like winter and snow # newline 이 escape 되어 기능을 상실해 한줄이됨.
```

"" ""

Double quote 안에서는 \$ ` ! 특수기능을 하는 문자들이 해석되어 실행되고 공백과 개행이 유지됩니다. 변수 사용 시에도 동일하게 적용되며 quote 을 하지 않으면 no quote 과 같아져서 공백과 개행이 유지되지 않습니다.

```
$ echo "I
> like
> winter      and      snow"
I                                     # 공백과 개행이 유지 된다.
like
winter      and      snow

##### 변수 사용시 #####

$ AA="this      is
two      lines"

$ echo $AA      # 공백과 개행이 유지되지 않는다.
this is two lines

$ echo "$AA"    # 공백과 개행이 유지된다.
this      is
two      lines
```

특수기능을 하는 문자들이 해석되어 실행된다.

```
$ AA=hello

$ echo "$AA world `date +%Y`"
hello world 2015

# 특수문자를 escape 할 경우
$ echo "\"$AA world `date +%Y`\""
$AA world `date +%Y`
```

Double quote 에서 escape 할 수 있는 문자들

" \$ ` \ newline

! 문자를 이용한 history 확장이 enable 되는 프롬프트 상에서는 ! 문자를 escape 하여 history 확장을 막을수 있으나 \ 문자는 출력에 남게됩니다. 그럴경우 다음과 같은 방법으로 escape 할수있습니다.

```
# history 확장이 되지 않지만 '\' 문자가 출력에 남는다.
$ echo "hello\!516world"
hello\!516world

$ echo "hello"\!"516world"
hello!516world

$ echo "hello"'"!'"516world"
hello!516world
```

Array 와 관련된 특수기능

double quotes 은 array 와 관련해서 특수한 기능이 있는데 전체 원소를 나타내는 `${arr[@]}` 를 quote 하면 그 의미는 `"${arr[0]}" "${arr[1]}" "${arr[2]}" ...` 와 같게 되고 `${arr[*]}` 를 quote 하게 되면 그 의미는 `"${arr[0]}x${arr[1]}x${arr[2]}x..."` 와 같게 됩니다. 여기서 x 는 IFS 값의 첫번째 문자를 나타냅니다.

`"$@"` , `"$*"` [positional parameters](#) 에서도 동일하게 적용됩니다.

❗

별다른 기능 없이 모든 문자를 있는 그대로 표시합니다. escape 도 되지 않습니다. 이 안에서 single quote 을 사용하려면 뒤에 이어지는 `$' '` 을 사용해야 합니다.

```
$ AA=hello

$ echo '$AA world'
> `date`
> \ $AA
> '
$AA world
`date`
\ $AA
```

\$' '

이건 `' '` 와 같은데 [escape 문자](#)를 사용할 수 있습니다. escape 문자가 처리되고 난 결과는 `$` 가 제외된 `' '` 상태가 됩니다.

```
$ echo $'I like\n\'winter\'\'tand\'t\'snow\'' # \n, \t, \' escape 문자가 사용되었다.
I like
\'winter\'    and    \'snow\'

-----

$ IFS=$'\n'
$ IFS=$' \t\n'
```

Quotes 을 서로 붙여 사용기

두개의 quote 을 공백을 두지 않고 서로 붙이면 하나가 됩니다. 이 원리는 변수를 포함하는 명령 스트링을 만들거나 함수에 전달할 인수를 하나로 만들때 유용하게 사용할 수 있습니다.

명령 스트링을 만들 때

' ' 을 사용해 명령문을 작성하였는데 그 안에 shell 변수를 사용할 일이 생기면 다음과 같이 ' ' 을 분리한 후 double quote 한 변수를 공백 없이 붙여 사용하면 됩니다. 변수를 quote 하지 않거나 single quote 과 double quote 사이에 공백이 있으면 안됩니다.

```
# 원본 명령
sed -rn '1p;2,${s/foo/bar/};p}'

# single quote 을 분리한 후 shell 변수를 double quote 하여 공백 없이 붙인다.
sed -rn '1p;2,${s/"$var1/$var2"/};p}'
```

명령의 인수를 만들 때

명령에 인수를 만들어 전달할 때도 두 quote 을 서로 붙여 사용하면 하나의 인수가 됩니다.

```
$ ./args.sh 11 "hello "$'$world \u2665' 33

$0 : ./args.sh
$1 : 11
$2 : hello $world ♥
$3 : 33
```

Escape Sequences

다음과 같은 경우에서 escape 문자가 처리됩니다.

모두 동일하게 처리되는 것은 아니고 조금씩 틀린부분이 있습니다.

- `echo -e " " or ' '` 에서
- `$' '`
- `printf " " or ' '` 에서
- `printf %b` 에서

| Escape Sequence | Character represented | 적용 |
|---------------------------------|---|---------------------------|
| <code>\\</code> | backslash | * |
| <code>\a</code> | alert (bell) | * |
| <code>\b</code> | backspace | * |
| <code>\e</code> | escape (ASCII 033) | * |
| <code>\f</code> | form feed | * |
| <code>\n</code> | newline | * |
| <code>\r</code> | carriage return | * |
| <code>\t</code> | horizontal tab | * |
| <code>\v</code> | vertical tab | * |
| <code>\'</code> | single quote | <code>printf \$' '</code> |
| <code>\"</code> | double quote | * |
| <code>\?</code> | question mark | <code>printf \$' '</code> |
| <code>\<NNN></code> | 8 비트 문자로 <code>N</code> 은 8 진수 값입니다. (1 ~ 3 개) | <code>printf \$' '</code> |
| <code>\O<NNN></code> | 8 비트 문자로 <code>N</code> 은 8 진수 값입니다. (1 ~ 3 개) | <code>echo</code> |
| <code>\x<HH></code> | 8 비트 문자로 <code>H</code> 은 16 진수 값입니다. (1 ~ 2 개) | * |
| <code>\u<HHHH></code> | 유니코드 문자 입니다. <code>H</code> 는 16 진수 값입니다. (1 ~ 4 개) | * |
| <code>\U<HHHHHHHH></code> | 유니코드 문자 입니다. <code>H</code> 는 16 진수 값입니다. (1 ~ 8 개) | * |
| <code>\c</code> | 이후의 스트링은 출력에서 제외됩니다. | <code>echo</code> |
| <code>\cx</code> | Ctrl-x 문자, 가령 <code>\$'\cx'</code> 는 Ctrl-Z (^Z) | <code>\$' '</code> |

적용의 * 문자는 모두를 뜻합니다.

Variables

변수 이름은 알파벳 (대, 소문자), 숫자, `_` 로 만들 수 있으며 이름의 첫 문자로 숫자가 올 수 없습니다. 변수이름을 대문자로 사용할때는 `shell` 환경변수와 중복되지 않도록 주의해야 합니다. 생성한 변수는 `subshell` 이나 `source` 한 스크립트 내에서는 별다른 설정 없이 사용할 수 있으나 `child process` 에서도 사용하려면 `export` 해야 합니다. 현재 `array` 변수는 `export` 할 수 없습니다.

Variable States

Shell 에서는 변수의 상태를 3 가지로 구분해볼 수 있습니다.

1. 변수가 존재하지 않는 상태 또는 `unset` 상태

변수가 존재하지 않는 상태란 변수에 값을 (`null` 값 포함) 한번도 대입한 적이 없는 상태를 말합니다. 그러므로 값을 할당하지 않고 `declare AA` or `local AA` 와 같이 선언만 한 경우도 해당됩니다. 이미 사용했던 변수도 `unset` 명령을 사용하면 이 상태로 됩니다.

2. `null` 값인 상태

다음과 같은 경우 변수가 존재하고 `null` 값을 갖고있는 상태가 됩니다.

```
AA= AA="" AA=' '
```

3. `null` 이외의 값을 가지고 있는상태

다음과 같은 경우 변수가 존재하고 무엇이든 값을 가지고 있는 상태입니다.

```
AA=" " AA="hello" AA=123
```

값을 구분하기

3번 무엇이든 값을 가지고 있는 상태와 그렇지 않은 상태 (1, 2번) 은 다음과 같이 구분할 수 있습니다.

```
# $var 가 1, 2 번 상태일 때 참
if [ -z "$var" ]; then ...

# $var 가 3 번 상태일 때 참
if [ -n "$var" ]; then ...
```

1번 unset 상태와 2번 null 상태는 다음 명령으로 구분할 수 있습니다. unset 상태일 경우는 1 을 그외는 0 을 리턴합니다.

```
$ [ -v asdfghjk ]; echo $? # 현재 asdfghjk 변수는 존재하지 않는 상태
1

$ asdfghjk=""

$ [ -v asdfghjk ]; echo $?
0

$ asdfghjk=123

$ [ -v asdfghjk ]; echo $?
0

$ unset asdfghjk

$ [ -v asdfghjk ]; echo $?
1
```

-u | set -o nounset 옵션을 이용하면 존재하지 않는 변수 사용시 script 실행을 종료할 수 있습니다.

Shell Functions

함수를 만들어 사용하는 이유는 여러개의 명령들을 하나의 group 으로 묶어서 사용하기 쉬운 함수 명을 이용해 같은 context 에서 실행시키는데 있습니다. 명령 group 은 { ; } , () 을 이용해 만드는데 차이점은 { ; } 은 현재 shell 에서 () 는 subshell 에서 실행된다는 점이 다릅니다. 그래서 보통 함수를 정의할때 { ; } 을 사용하지만 필요하다면 () 을 사용할 수도 있습니다.

```
# echo hello world | read var; 는 파이프로 인해 subshell 에서 실행되어
# echo "$var" 는 값이 표시되지 않는다.
$ echo hello world | read var; echo "$var"

# { } 을 이용해 명령 group 을 만들면 read, echo 명령이 같은 context
# 에서 실행되어 정상적으로 값이 표시됩니다.
$ echo hello world | { read var; echo "$var" ;}

# hello 는 터미널에 표시되고 world 만 outfile.txt 에 저장된다.
$ echo hello; echo world > outfile.txt

# 명령 group 을 이용하면 hello world 둘 다 outfile.txt 에 저장된다.
{
    echo hello
    echo world
} > outfile.txt

# 명령 group 을 하나의 짧은 이름으로 사용 ( 함수정의 )
{
    read var1
    read var2
    echo "$var1 $var2"
} < infile.txt

f1() {
    read var1
    read var2
    echo "$var1 $var2"
}

$ f1 < infile.txt
```

함수 이름은 알파벳 (대, 소문자), 숫자, _ 로 만들 수 있으며 이름의 첫 문자로 숫자가 올 수 없습니다. 함수를 사용하는 방법은 일반 명령들과 동일하며 종료값 지정은 return 명령으로 합니다. 정의한 함수는 subshell 이나 source 한 스크립트 내에서는 별다른 설정 없이 사용할 수 있으나 child process 에서도 사용하려면 export -f 해야 합니다.

함수를 정의하는 방법

shell 함수는 정의할때 프로그래밍 언어에서처럼 매개변수를 적지 않습니다. 전달된 인수값은 함수 내에서 `$1 $2 $3 ...` 특수 변수에 자동으로 할당됩니다. 그리고 `function` 키워드는 `bash` 에서만 사용할수있고 `sh`에서는 사용할 수 없으므로 보통 첫번째 방법을 사용합니다.

```
# shell 함수는 정의할때 매개변수를 적지 않는다.
```

```
X. 함수명 ( p1 p2 p3 ) { ... ;}
```

```
1. 함수명 () { ... ;}
```

```
2. function 함수명 () { ... ;}
```

정의된 함수 내용 보기

```
declare -f 함수명
```

현재 shell 에 정의된 모든 함수명 보기

```
declare -F
```

정의된 함수 삭제하기

```
unset -f 함수명
```

현재 shell 에 정의된 모든 변수명 보기

```
compgen -A variable
```

```
# array 변수만 보기
```

```
compgen -A arrayvar
```

정의된 변수 삭제하기

```
unset -v 변수명
```

함수를 실행할 땐 먼저 정의가 돼있어야 한다.

함수를 실행하기 전에 먼저 정의가 돼있어야지 실행할 수 있습니다.

```

foo1                # 여기서는 foo1 함수 정의가 안돼있기 때문에 실행할 수 없다.

foo1() {
    echo "foo1"
    foo2            # foo1 함수를 실행한 곳이 foo2 아래이므로 실행 가능
}

foo2() {
    echo "foo2"
}

foo1                # 여기서는 foo1 함수를 실행할 수 있다.

```

변수는 기본적으로 **global scope**

여기서 global scope 이라는것은 현재 스크립트 파일 입니다. (source 한 파일도 포함).

```

#!/bin/bash

BB=200

foo() {
    AA=100
}

foo

echo $AA
echo $BB

##### output #####

100
200

```

local, declare 명령을 사용하여 지역변수를 설정할 수 있다.

declare 가 함수내에서 사용되면 local 과 동일한 역할을 합니다.

local 은 declare 와 달리 global scope 에서는 사용할 수 없습니다.

```
#!/bin/bash

foo() {
    local AA=100
    BB=200
}

foo

echo $AA
echo $BB

##### output #####

200
```

local 변수를 이용한 recursion

```
# 자손 프로세스 프린트하기
list_descendants ()
{
    local children=$(ps -o pid= --ppid $1)

    for pid in $children
    do
        list_descendants $pid
    done

    echo $children
}

$ ps jf $(list_descendants $$)
```

local 변수도 child 함수에서 읽고 쓸 수 있다.

Shell 함수에서 중요한 내용이라고 할 수 있습니다. shell 함수만의 특이한 점 중에 하나인데 local 로 설정한 변수를 child 함수에서 읽고 쓸수가 있습니다. 그리고 변경한 값도 parent 함수에 적용이 됩니다.

sh 에서도 동일하게 동작합니다.

```
#!/bin/bash

f1() {
    local AA=100
    f2
    echo f1 AA after f2 call : $AA
}

f2() {
    echo f2 AA : $AA
    AA=200
}

f1

echo global AA : $AA

##### output #####

f2 AA : 100
f1 AA after f2 call : 200
global AA :
```

중첩함수

함수는 local 을 사용할 수 없습니다. 그래서 함수 내에서 정의가 되더라도 global scope 이므로 외부 함수명과 중복되지 않게 주의해야 합니다. sh 에서도 동일하게 동작합니다.

```
#!/bin/bash

f1() {

    local AA=100
    f1_1() {
        echo f1_1 AA : $AA
        AA=200
    }
    f1_1
    echo f1 AA : $AA
}

f1

echo global AA : $AA

##### output #####

f1_1 AA : 100
f1 AA : 200
global AA :
```

함수에서 연산 결과를 리턴하는 방법

shell script 가 프로그래밍 언어와 다른점 중에 하나가 return 명령의 역할입니다. shell 에서는 return 명령이 함수에서 연산한 결과를 반환하는데 사용되는 것이 아니라 exit 명령과 같이 함수가 정상적으로 종료됐는지 아니면 오류가 발생했는지를 나타내는 종료상태값을 지정하는 용도로 사용됩니다. 그러므로 연산 결과를 반환하는데 return 명령을 사용하면 안됩니다.

그럼 함수에서 연산한 결과를 반환 받으려면 어떻게 해야 하는지가 문제인데요. 앞서 함수는 일반 명령들과 동일하게 사용된다고 했습니다. 다음은 외부 명령을 사용할때 결과값을 받는 방법입니다.

```
$ AA=$(expr 1 + 2)
$ echo $AA
3

$ AA=`date +%Y`
$ echo $AA
2015
```

함수에서도 외부 명령과 동일하게 [명령치환](#) 을 사용하면 됩니다.

```
f1() { expr $1 + $2 ;}
f2() { date "+%Y" ;}
f3() { echo "hello $1" ;}
```

```
$ res=$(f1 1 2)
```

```
$ echo $res
```

```
3
```

```
$ res=`f2`
```

```
$ echo $res
```

```
2015
```

```
$ res=$(f3 world)
```

```
$ echo $res
```

```
hello world
```

함수에 인수를 전달하기

외부 명령을 실행할때 인수를 전달하기 위해 () 를 사용하지 않듯이 함수에서도 동일하게 () 를 사용하지 않습니다. 전달된 인수는 함수내에서 `$1 $2 $3 ...` [positional parameters](#) 에 자동으로 할당되며 scope 은 local 변수와 같습니다. `$0` 은 현재 실행중인 스크립트 파일이름을 나타냅니다.

```
----- test.sh -----
#!/bin/bash

echo
echo number of arguments: $#
echo '$0' : "$0"
echo '$1' : "$1"
echo '$2' : "$2"
echo '$3' : "$3"
echo '$4' : "$4"
echo '$5' : "$5"

-----

$ AA=(22 33 44)
$ test.sh 11 "${AA[@]}" "55 END"

number of arguments: 5
$0 : ./test.sh
$1 : 11
$2 : 22
$3 : 33
$4 : 44
$5 : 55 END
```

`$@` , `$*` 변수는 함수에 전달된 인자들 전부를 포함합니다. 변수를 `quote` 하지 않으면 단어분리에 의해 두변수의 차이가 없지만 `quote` 을 하게 되면 `"$@"` 의 의미는 `"$1"` `"$2"` `"$3"` ... 와 같게되고 `"$*"` 의 의미는 `"$1c$2c$3 ... "` 와 같게됩니다. (여기서 `c` 는 `IFS` 변수값의 첫번째 문자 입니다.)


```
#!/bin/bash

foo() {
    echo \${@} : ${@}
    echo \$* : $*

    echo '====="${@}"====='
    for v in "${@}"; do
        echo "$v"
    done

    echo '====="$*"====='
    for v in "$*"; do
        echo "$v"
    done
}

foo 11 "22      33" 44

##### output #####

${@} : 11 22 33 44
$* : 11 22 33 44
====="${@}"=====
11
22      33
44
====="$*"=====
11 22      33 44
```

indirection 을 이용해 array 를 함수에 전달

```
#!/bin/bash

foo() {
    echo "$1"

    local ARR=( "${!2}" )      # '!2' 부분이 'AA[@]' 로 바뀐다.

    for v in "${ARR[@]}; do
        echo "$v"
    done

    echo "$3"
}

AA=(22 "33 44" 55)
foo 11 'AA[@]' 66

##### output #####
11
22
33 44
55
66
```

Bash version 4.3+ 에서는 named reference 를 사용할 수 있습니다.

```
declare -n refname
local -n refname
```

```
#!/bin/bash

f2() {
    declare -n RVAR=$1      # declare 는 함수내에서 기본적으로 local 변수를 생성
    RVAR=200
}

f1() {
    local VAR=100
    echo f1 : local VAR = $VAR
    f2 VAR
    echo f1 : local VAR after f2 call = $VAR
}

f1

##### output #####

f1 : local VAR = 100
f1 : local VAR after f2 call = 200
```

named reference 를 이용해 array 를 함수에 전달.

indirection 을 이용할 때처럼 새로 array 를 생성하지 않아도 된다.

```
foo() {  
    echo "$1"  
  
    local -n ARR="$2"  
  
    for v in "${ARR[@]"; do  
        echo "$v"  
    done  
  
    echo "$3"  
}  
  
AA=(22 "33 44" 55)  
foo 11 AA 66  
  
##### output #####  
11  
22  
33 44  
55  
66
```

Exit Status

Shell script 는 명령을 다룹니다. 그리고 실행되는 모든 명령은 종료 상태 값을 반환합니다. if, while, until, ||, && 모두 종료 상태 값을 사용해서 참, 거짓을 판단합니다. 프로그래밍 언어에서처럼 특정값 (숫자나 스트링 또는 예약어) 를 사용하지 않습니다.

```
# 종료코드 0 이 참이라고 해서 다음과 같이 사용할 수 없습니다.
```

```
$ if 0; then echo true; else echo false; fi
0: command not found
```

```
$ 0 && echo true
0: command not found
```

종료코드에는 1 byte 를 사용하므로 0 ~ 255 번을 사용할 수 있습니다. 그중에 0 만 정상 종료를 나타내고 나머지는 오류를 분류해 나타내는데 사용됩니다. 1, 2, 126 ~ 165 번은 shell 에서 사용됩니다.

종료값을 지정할때는 shell (subshell 포함) 이나 script 파일에서는 exit 명령으로 하며 function 이나 source 명령으로 읽어들이는 파일에서는 return 명령으로 합니다.

0

정상종료 (Success)

1

일반적인 에러

```
let "var = 1 / 0" : division by 0
```

2

Syntax error, 잘못 사용된 builtin 명령

```
test.sh: line 6: syntax error near unexpected token 'fi'
exit: 3.14: numeric argument required
```

126

명령을 실행할 수 없음

명령은 존재하지만 executable 이 아니거나 퍼미션 문제
bash: ./mylogfile.txt: Permission denied

127

명령 (파일) 이 존재하지 않음

typo 또는 \$PATH 문제
asdfg: command not found

128 + N

Signal N 에의한 종료.

가령 kill -9 PID 로 종료 됐다면 \$? 값은 $128 + 9 = 137$

대입 연산의 종료값은?

= , += 메타문자를 이용한 식은 명령문이 아닙니다. 그래서 ; 로 구분없이 한줄에 여러개를 쓸 수도 있습니다. 종료값은 기본적으로 항상 0 이지만 명령치환과 함께 사용될 경우는 명령치환 종료값을 따릅니다.

```
# 대입연산은 명령문이 아니므로 한줄에 여러개를 쓸수도 있다.
$ AA=11 BB=22 CC=33

$ echo $AA $BB $CC
11 22 33

-----

$ [ 1 -eq 2 ]
$ echo $?
1

# 대입연산의 종료값은 기본적으로 0 이다
$ [ 1 -eq 2 ]
$ AA=11
$ echo $?
0

-----

# 명령치환과 사용될 경우 명령치환 종료값을 따른다.
$ readlink -e asdfg; echo $?
1
$ AA=$( readlink -e asdfg ); echo $?
1

$ readlink -e test.sh; echo $?
/home/mug896/tmp/test.sh
0
$ AA=$(readlink -e test.sh); echo $?
0
$ echo $AA
/home/mug896/tmp/test.sh

# 그러므로 if 문에서 사용할 수도 있다.
if AA=$( readlink -e test.sh ); then ...

# 단 local, declare 명령과 함께 사용될 경우는 적용되지 않습니다.
$ AA=$( readlink -e asdfg ); echo $?
1
$ declare AA=$( readlink -e asdfg ); echo $?
0
```

Pattern Matching

Regex 이 `[[]]` 표현식에 한에서 제한적으로 사용할수 있다면 glob 문자 (`*` , `?` , `[]`) 를 이용한 패턴매칭은 shell script 전반에서 사용할 수 있습니다.

- case 문 에서
- 매개변수 확장에서 (substring removal, search and replace)
- `[[]]` 표현식 에서
- filename matching (globbing)

Glob 문자의 의미

| 문자 | 의미 |
|----------------------|----------------------------------|
| <code>*</code> | empty 스트링을 포함해 모든 문자와 매치됩니다. |
| <code>?</code> | 하나의 문자와 매치됩니다. |
| <code>[...]</code> | bracket 표현식에 매치되는 하나의 문자와 매치됩니다. |

Bracket 표현식

| 표현식 | 의미 |
|--|---|
| <code>[XYZ]</code> | X, Y or Z 문자에 대해 매치됩니다. |
| <code>[X-Z]</code> | 위 표현식을 - 문자를 이용해 range 로 나타낼수 있습니다. |
| <code>[:class:]</code> | POSIX character class 와 매치됩니다. (alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, xdigit) |
| <code>[^ ...]</code> or <code>[! ...]</code> | <code>^</code> , <code>!</code> 문자는 Not 을 의미합니다. 가령 <code>[^XYZ]</code> 이라면 XYZ 이외의 문자와 매치됩니다. sh 에서는 <code>!</code> 문자를 사용해야 합니다. |

사용예)

```
$ AA="inventory.tar.gz"

$ [[ $AA = *.tar.gz ]]; echo $?
0
$ [[ $AA = inventory.tar.?? ]]; echo $?
0

$ ls
address.class  address.java  read.c  read.h  write.c  write.h
$ ls *. [ch]
read.c  read.h  write.c  write.h
```

패턴에 공백을 사용하려면 **escape** 합니다.

```
#!/bin/bash

AA='hello dog cat world'

[[ $AA = *dog\ cat* ]]; echo $?
```

Extended Pattern

`shopt -s extglob` shell 옵션을 설정하면 다음과 같은 확장패턴을 사용할수 있습니다. `!` 문자를 구분자로 하여 여러개의 패턴을 사용할수 있습니다.

| 표현식 | 의미 |
|-------------------|--|
| ?(<PATTERN-LIST>) | 주어진 패턴이 zero or one 발생하면 매치됩니다. |
| *(<PATTERN-LIST>) | 주어진 패턴이 zero or more 발생하면 매치됩니다. |
| +(<PATTERN-LIST>) | 주어진 패턴이 one or more 발생하면 매치됩니다. |
| @(<PATTERN-LIST>) | 주어진 패턴이 one 발생하면 매치됩니다. |
| !(<PATTERN-LIST>) | <code>!</code> 문자는 Not 의 의미로 주어진 패턴과 맞지 않으면 매치됩니다. |

사용예)


```
# *.jpg 파일을 제외하고 전부
$ ls !(*.jpg)

# *.jpg , *.gif , *.png 파일을 제외하고 전부
$ ls !(*.jpg|*.gif|*.png)

# AA 변수값의 leading space 와 trailing space 를 제거
$ AA=${AA##+([[:space:]])}; AA=${AA%%+([[:space:]])}

-----

$ AA=apple

$ [[ $AA = @(ba*(na)|a+(p)le) ]]; echo $?
0

$ AA=banana

$ [[ $AA = @(ba*(na)|a+(p)le) ]]; echo $?
0

$ AA=applebanana

$ [[ $AA = @(ba*(na)|a+(p)le) ]]; echo $?
1

$ [[ $AA = +(ba*(na)|a+(p)le) ]]; echo $?
0
```

#!

She (#) bang (!) 또는 shabang, hashbang 라인은 스크립트 파일의 첫줄에 사용하여 스크립트가 어떤 명령에 의해 실행될지를 지정합니다. 프로그램의 경로는 절대경로나 현재 디렉토리로부터 상대경로를 사용할 수 있으며 변수는 사용할 수 없습니다. 그리고 shebang 라인에서 사용할 수 있는 옵션은 하나로 제한됩니다.

```
#!/bin/bash          # bash 스크립트 실행을 위한 shebang line
...
#!/usr/bin/perl -T    # perl 스크립트 실행을 위한 shebang line
...
#!/bin/sed -f         # sed 스크립트 실행을 위한 shebang line
...
#!/usr/bin/awk -f     # awk 스크립트 실행을 위한 shebang line
...
```

#! 문자는 텍스트 파일인 스크립트를 바이너리 실행파일처럼 실행할 수 있게 해주는 역할을 합니다. OS 가 프로그램을 실행할때 처음에 **#!** 문자를 만나면 뒤에 이어지는 나머지 라인을 해당 파일을 실행하기 위한 인터프리터로 취급합니다. 그러므로 가령 foo 라는 이름의 스크립트가 있고 첫줄이 `#!/bin/sed -f` 로 시작한다면 프롬프트 상에서 `foo arg1 arg2 arg3` 명령을 실행할 경우 실제로는 `/bin/sed -f foo arg1 arg2 arg3` 와 같이 실행되게 됩니다.

다음은 `#!/bin/bash -x` shebang 라인을 갖는 bash 스크립트를 실행했을때 ps 입니다.

```
PID TTY      STAT   TIME COMMAND
4410 pts/7    Ss     0:00 bash
22666 pts/7    S+     0:00  \_ /bin/bash -x ./test.sh 11 22 33
22667 pts/7    S+     0:00      \_ sleep 20
```

Portability

간혹 `#!/usr/bin/env` 을 사용하는 shebang 라인을 본적이 있을겁니다. 이와 같이 사용하는 이유는 OS 별로 프로그램의 위치가 다를수 있기 때문인데요 가령 python 프로그램은 `/usr/bin/python` 에 위치할 수도 있고 `/usr/local/bin/python` 에 위치할 수도 있습니다. 이럴경우 특정 OS 에서는 shebang 라인을 수정해서 사용해야 합니다. 하지만 `#!/usr/bin/env python` 와 같이 사용하면 \$PATH 를 검색해서 python 프로그램을 실행하게 되므로 문제를 해결할 수 있습니다. 이 방법의 단점은 python 에 옵션을 사용할 수 없다는 것입니다.

스크립트 파일에는 set uid 를 설정할 수 없습니다.

프로그램을 root 권한으로 실행하게 만들수 있는 set uid 설정을 shell script 에 적용할 경우 보안과 관련해서 너무 많은 문제가 생길 수 있다고 합니다. 그리고 shell script 은 기본적으로 명령들을 다루기 때문에 어떤 명령이 보안과 관련해서 버그가 있을 경우 바로 문제로 이어질 수 있습니다.

Shebang line 을 쓰지 않아도 될 때

스크립트를 직접 명령으로 실행시킬 때 외에는 shebang line 을 쓰지 않아도 됩니다. source 명령으로 읽어들이는 스크립트, ~/.bashrc , ~/.profile 같은 환경설정 파일들, 명령 자동완성 함수를 작성할 때 등등 ...

Interactive vs Non-Interactive Shell

Shell 이 실행되는 환경을 두가지로 나누어 볼수 있습니다. 사용자로 부터 프롬프트를 통해 직접 명령을 입력받아 실행시키는 interactive shell 과 작성한 script 파일을 실행하는것과 같은 non-interactive shell 입니다. history 확장이나 alias, job control 과 같은 기능은 기본적으로 interactive shell 에서 사용하기 위한 것으로 script 을 실행할 때는 disable 됩니다.

Interactive shell 인지 아닌지는 다음 명령을 통해서 알아볼 수 있습니다.

```
# '$-' 변수는 set 명령에 의해 설정돼있는 option flags 을 담고 있으며
# 'i' 는 interactive 를 의미합니다.
case $- in
  *i*) echo interactive shell ;;
  *) echo non-interactive shell ;;
esac
```

Script 실행시에는 alias 기능이 disable 됩니다.

사용자마다 다른 alias 설정을 가지고 있기 때문에 script 가 배포되어 실행될때 alias 적용이 된다 면 오류가 발생할 수 있습니다. (alias 는 설정할 경우 우선순위가 제일 높습니다.)

Script 실행시에는 history 확장이 disable 됩니다.

history 확장 기능은 interactive shell 에서 사용하기 위한 것으로 script 실행 시에는 기본적으로 disable 됩니다. `history` builtin 명령도 사용할 수 없습니다.

Script 실행시에는 job control 이 disable 됩니다.

job control 도 interactive shell 에서 사용하기 위한 것으로 script 실행 시에는 disable 됩니다. 그렇다고 해서 `&` 메타문자를 이용해 background job 을 생성하지 못한다는건 아니고 다만 `bg`, `fg`, `suspend` 명령을 사용할 수 없습니다. 하지만 `jobs`, `wait`, `disown` 명령들은 사용할 수 있습니다.

Script 실행시에는 exec 명령이 실패하면 바로 종료됩니다.

프롬프트 상에서는 `exec` 명령이 실패하면 에러 메시지와 함께 다음 프롬프트가 뜨게되지만 script 실행시에는 바로 종료하게 됩니다.

시작시 실행하는 파일이 다르다.

Interactive shell 이 시작될때는 `~/.bashrc` 를 실행하고 shell script 나 `bash -c` 로 시작하는 Non-Interactive shell 이 시작될때는 `BASH_ENV` 변수에 설정된 파일을 실행합니다.

Login vs Non-Login Shell

이번엔 좀 다른 관점에서 shell 이 실행되는 환경을 두가지로 나누어 볼 수 있습니다. 리모트 서버에 ssh 이나 telnet 으로 접속해서 login 과정을 거쳐 사용하게 되는 login shell 과 윈도우 매니저에서 메뉴로 제공하는 터미널 프로그램을 실행시켜 사용하게 되는 non-login shell 입니다.

login shell 인지 아닌지는 `shopt -q login_shell; echo $?` 을 통해 알아볼 수 있습니다.

Login Shell

- Login shell 은 시작시 `/etc/profile`, `~/.bash_profile`, `~/.bash_login`, `~/.profile` 순서로 파일을 읽어들이어 실행합니다. (`bash -l -c` 와 같은 non-interactive login shell 에도 적용됩니다.)
- Login shell 에서는 `logout` (또는 `exit`) 시에 `~/.bash_logout` 파일을 실행합니다.
- Login shell 에서는 `shopt -s huponexit` 옵션 설정을 통해 `logout` 시에 background 로 실행되는 job 들에게 HUP 시그널을 보내 종료하게 할 수 있습니다.
- `echo $0` 했을때 `-bash` 로 표시되면 login shell 을 의미합니다.
- Login shell 에서는 `logout builtin` 명령을 사용할수 있습니다.

Non-Login Shell

- Interactive non-login shell 은 시작시 `.bashrc` 파일을 실행합니다. 여기서 `interactive` 를 붙인 이유는 non-interactive shell 인 `script` 가 실행될때는 `.bashrc` 파일을 읽지 않기 때문입니다. `script` 가 실행될 때마다 읽어들이는 파일은 `BASH_ENV` 환경변수에 등록할 수 있습니다.

.rc 파일의 유래: 1965 년 MIT Compatible Time-Sharing System (CTSS) 에는 하나의 파일에 여러 명령을 넣어놓고 실행하는 기능이 있었는데 'run commands' 를 뜻하는 의미로 'runcom' 이라 불렀다고 합니다. 여기서 앞 글자를 따서 .rc 파일 이라 한다고 합니다.

- Non-login shell 에서도 `su - userid` 나 `bash -l` 같은 명령을 사용해서 login shell 을 만들 수 있습니다.

Real user id vs Effective user id

Real user id 는 시스템에 로그인 했을때의 id 를 말하며 `$UID` readonly 환경변수에 저장됩니다. effective user id 는 `set uid` 비트가 설정된 프로그램을 실행했을때 변경되며 `$EUID` readonly 환경변수에 저장됩니다.

```

### alice 계정 ###
alice@box:/tmp/priv$ cp /bin/bash bash
alice@box:/tmp/priv$ chmod u+s bash      # set uid 비트 설정

### bob 계정 ###
bob@box:/tmp/priv$ ./bash -p
bash-4.3$ id                # euid 가 alice 로 변경되었다
uid=1002(bob) gid=1005(gamers) euid=1004(alice) groups=1002(bob),1005(gamers)

```

.bashrc.d 만들어 사용하기

shell 을 사용하다 보면 내가 만든 함수, alias, 자동완성 이 쌓이게 되는데요. 이럴때 `~/.bashrc.d` 디렉토리를 만들어 한 곳에서 관리하면 좋습니다. 이름 뒤에 `.d` 를 붙인것은 `.bashrc` 에서 사용하는 파일들이 위치하는 디렉토리 라는 뜻입니다.

```

# ~/.bashrc 에 들어갈 내용

# alias 설정 읽어들이기
if [ -r ~/.bashrc.d/aliases ]; then
    source ~/.bashrc.d/aliases
fi

# 함수 설정 읽어들이기
for file in "$HOME"/.bashrc.d/functions/*.bash ; do
    [ -r "$file" ] && source "$file"
done

# 자동완성 설정 읽어들이기
for file in "$HOME"/.bashrc.d/completions/*.bash ; do
    [ -r "$file" ] && source "$file"
done

unset -v file

```

Arrays

Array 는 bash 에서 제공하는 특수 표현식에 속하며 POSIX 에는 정의되어있지 않습니다. 그러므로 sh 에서는 사용할 수 없습니다. array 는 index 로 숫자를 사용하는 indexed array 와 스트링을 사용할수 있는 associative array 가 있습니다. indexed array 는 별다른 설정없이 사용할수 있으나 associative array 는 `declare -A 변수명` 으로 먼저 선언을 해줘야 합니다. 현재 사용중인 array 를 empty 로 만들때 `AA=()` 를 사용하는데 이것은 `unset -v AA` 와 동일한 효과를 갖습니다. 그리고 현재 array 는 export 할 수 없습니다.

Array 의 선언

| 구분 | 문장 |
|--------------------------|--|
| indexed array | <code>declare -a array_name</code> <code>local -a array_name</code> |
| associative array (필수) | <code>declare -A array_name</code> <code>local -A array_name</code> |

Array 값의 사용

`AA=(11 22 33)` 와 같은 array 가 있을경우 `$AA` 값은 첫번째 원소인 11 과 같습니다. 여기서 `AA[2]` 값을 사용하기 위해 `$AA[2]` 와 같이 한다면 먼저 `$AA` 가 변수확장이 되어 11 이 되고 이어 `11[2]` 에서 globbing 이 일어나게 됩니다. 그러므로 array 값을 사용할 때는 반드시 `{ }` 매개변수 확장을 이용해야 합니다.

```
$ AA=(11 22 33)
```

```
# 만약 현재 디렉토리에 '112' 라는 파일이 있다면 globbing 이 일어나 결과는 '112' 가 됩니다.
```

```
$ echo $AA[2]
```

```
11[2]
```

```
$ echo ${AA[2]}
```

```
33
```

Array 값 프린트

`declare -p array_name` 명령을 이용하면 array 값을 볼 수 있습니다.


```
$ AA=(apple banana orange)

$ declare -p AA
declare -a AA='([0]="apple" [1]="banana" [2]="orange")'
```

@, * 차이점

Double quote 을 하지 않을 경우 `${array[@]}` 와 `${array[*]}` 는 차이가 없습니다. 왜냐하면 똑 같이 IFS 값에 의해 단어분리가 되기 때문입니다. 하지만 " " 로 quote 하였을 경우 @ 와 * 는 의미가 틀려집니다. `"array[@]"` 는 array 개개의 원소를 " " 로 quote 하여 나열하는 것과 같고 `"array[*]"` 는 array 의 모든 원소를 하나의 " " 안에 IFS 변수값을 구분자로 하여 넣는 것과 같습니다.

| Quote 여부 | 의미 |
|--|--|
| <code>\${AA[@]}</code> <code>\${AA[*]}</code> | quote 하지 않으면 둘은 차이가 없다 |
| <code>"\${AA[@]}"</code> | <code>"\${AA[0]}" "\${AA[1]}" "\${AA[2]}" ...</code> |
| <code>"\${AA[*]}"</code> | <code>"\${AA[0]}X\${AA[1]}X\${AA[2]}..."</code> 여기서 'X' 는 IFS 변수값 중에 첫번째 문자 |

```
# 예를 위해 공백을 5 개씩 주었다.

$ AA=( "Arch      Linux" "Ubuntu      Linux" "Fedora      Linux" )
$ echo ${#AA[@]}
3

# quote 하지 않아 IFS 값에 의해 단어분리가 일어나 공백이 유지되지 않고
# for 문을 돌려도 6 개로 나온다

$ echo ${AA[@]}                                # 연이은 공백이 하나의 공백으로 바뀌었다.
Arch Linux Ubuntu Linux Fedora Linux

$ for v in ${AA[@]}; do echo "$v"; done        # 원소개수가 6 개로 나온다
Arch
Linux
Ubuntu
Linux
Fedora
Linux

$ echo ${AA[*]}                                # 마찬가지로.
Arch Linux Ubuntu Linux Fedora Linux

$ for v in ${AA[*]}; do echo "$v"; done
Arch
Linux
Ubuntu
Linux
Fedora
Linux

##### quote 하였을 경우 #####

$ echo "${AA[@]}"
Arch      Linux Ubuntu      Linux Fedora      Linux  # 공백이 유지된다.

$ echo "${AA[*]}"
Arch      Linux Ubuntu      Linux Fedora      Linux

$ for v in "${AA[@]}; do echo "$v"; done      # "array[@]" 는 원소개수가 유지된다.
Arch      Linux
Ubuntu      Linux
Fedora      Linux

$ for v in "${AA[*]}; do echo "$v"; done      # "array[*]" 는 원소개수가 1 개로 나온다.
Arch      Linux Ubuntu      Linux Fedora      Linux
```

명령을 실행할때 `${arr[@]}` 를 인수에 포함시키면 ?

```
$ AA=(1 2 3 4 5)
$ args.sh "aa ${AA[@]} bb" # 인수가 분리된다.
$1 : aa 1
$2 : 2
$3 : 3
$4 : 4
$5 : 5 bb

-----

$ args.sh "aa ${AA[*]} bb"
$1 : aa 1 2 3 4 5 bb
```

특수 표현식

| 표현식 | 의미 |
|---|--|
| <code>\${#array[@]}</code> <code>\${#array[*]}</code> | array 전체 원소의 개수를 나타냄 |
| <code>\${#array[N]}</code> <code>\${#array[string]}</code> | indexed array 에서 N 번째 원소의 문자수 (stringlength) 를 나타냄 associative array 에서 index 가 string 인 원소의 문자수를 나타냄 |
| <code>\${array[@]}</code> <code>\${array[*]}</code> | array 전체 원소의 value 를 나타냄 |
| <code>\${!array[@]}</code> <code>\${!array[*]}</code> | array 전체 원소의 index 를 나타냄 |
| <code>\${!name@}</code> <code>\${!name*}</code> | name 으로 시작하는 이름을 갖는 모든 변수를 나타냄 예) echo "\${!BASH@}" |

Array 에 값 할당하기

index 를 나타내는 `[]` 에서는 `(())` 특수 표현식에서처럼 산술연산을 할수있고 변수를 사용할 수 있습니다.

```
##### indexed array #####

AA=( 11 "hello array" 22 )

AA=( [0]=11 [1]="hello array" [2]=22 )

AA[0]=11
AA[1]="hello array"
AA[2]=22

##### associative array #####

declare -A AA          # 먼저 declare -A 로 선언 해야 한다

AA=( [ab]=11 [cd]="hello array" [ef]=22 )

AA[ab]=11
AA[cd]="hello array"
AA[ef]=22
```

Array iteration 하기

```
##### indexed array #####

ARR=(11 22 33)

for idx in ${!ARR[@]}; do
    echo ARR index : $idx, value : ${ARR[idx]} # ${ARR[$idx]} 로 해도됨
done

##### associative array #####

declare -A AAR

AAR=( [ab]=11 [cd]="hello array" [ef]=22 )

for idx in ${!AAR[@]}; do
    echo ARR index : $idx, value : ${AAR[$idx]}
done
```

Array 복사하기

array AA 를 BB 로 복사할때 `BB=${AA[@]}` 와 같이 하면 array 복사가 아니라 AA 값 전체가 BB 변수에 할당됩니다. array 복사를 하려면 항상 `()` 를 사용해야 합니다.

```
$ AA=( 11 22 33 )

$ BB="${AA[@]}"
$ echo "${BB[1]}"          # 정상적으로 array 복사가 되지 않는다.

$ echo "$BB"              # array AA 의 전체 값이 BB 에 할당된다.
11 22 33

$ BB=( "${AA[@]}" )       # array 복사는 항상 '( )' 를 사용해야 한다
$ echo "${BB[1]}"
22
```

Array 원소 삭제하기

array 원소를 나타내는 데는 `[]` glob 문자가 사용되므로 globbing 이 발생하지 않게 항상 quote 해야합니다.

| 표현식 | 의미 |
|---|--|
| array=() unset -v array unset -v "array[@]" | array 전체를 삭제 |
| unset -v "array[N]" | indexed array 에서 N 번째 원소를 삭제 |
| unset -v "array[string]" | associative array 에서 index 가 string 인 원소를 삭제 |

array 원소를 삭제하면 자동으로 `${#array[@]}` 값에도 반영이 되고 for 문을 이용해도 삭제된 원소는 나타나지 않습니다. 그러나 삭제된 원소의 index 는 그대로 남아있고 뒤에오는 원소들의 index 값들도 바뀌지 않습니다. 다시 array 를 재할당하면 index 가 정렬됩니다.

```

$ AA=(11 22 33 44 55)

$ unset -v AA[2]

$ echo ${#AA[@]}                                # 삭제후 원소 개수가 정상적으로 반영됨
4

$ for v in "${AA[@]}"; do echo "$v"; done        # for 문에서도 정상적으로 반영됨.
11
22
44
55

$ echo ${AA[1]} : ${AA[2]} : ${AA[3]}           # 그런데 삭제된 index 2 가 공백으로 남아있다!
22 : : 44

$ AA=( "${AA[@]}" )                             # 재할당 하면 index 가 정렬된다
$ echo ${AA[1]} : ${AA[2]} : ${AA[3]}
22 : 44 : 55

```

null 값을 가지고 있는 원소 삭제하기 : unset 한 원소는 존재하지 않는 원소인 반면에 null 값을 가지고 있는 원소는 존재하고 있는 원소입니다. 다음과 같이 삭제할 수 있습니다.

```

$ AA=":arch linux:::ubuntu linux:::fedora linux::"

$ IFS=: read -ra ARR <<< "$AA"

$ echo ${#ARR[@]}
10
$ echo ${ARR[0]}

$ echo ${ARR[1]}
arch linux

# ARR 에 null 값을 가지고 있는 원소 삭제하기

$ set -f; IFS=''                                # 먼저 IFS 값을 null 로 설정
$ ARR=(${ARR[@]})                               # quote 을 사용하지 않는다.
$ set +f; IFS=$' \n\t'

$ echo ${#ARR[@]}
3
$ echo ${ARR[0]}
arch linux
$ echo ${ARR[1]}
ubuntu linux

```

Array 원소 뽑아내기

array 의 특정 원소들을 뽑아내려고 할때는 `${array[@]:offset:length}` 형식을 사용합니다.

```
$ AA=( Arch Ubuntu Fedora Suse Mint );

$ echo "${AA[@]:2}"
Fedora Suse Mint

$ echo "${AA[@]:0:2}"
Arch Ubuntu

$ echo "${AA[@]:1:3}"
Ubuntu Fedora Suse
```

Array 원소 추가하기

```
$ AA=( "Arch Linux" Ubuntu Fedora);

$ AA=( "${AA[@]}" AIX HP-UX);

$ echo "${AA[@]}"
Arch Linux Ubuntu Fedora AIX HP-UX

#####

$ BB=( 11 22 33 )
$ echo ${#BB[@]}
3

$ BB+=( 44 )
$ echo ${#BB[@]}
4

#####

$ declare -A AA=( [aa]=11 [bb]=22 [cc]=33 )
$ echo ${#AA[@]}
3

$ AA+=( [dd]=44 )
$ echo ${#AA[@]}
4
```

전체 array 원소에 패턴을 적용하기

패턴을 적용하여 매칭되는 부분을 바꾸거나 삭제할수 있습니다. 패턴에는 맨앞을 가리키는 `#` , 맨 뒤를 가리키는 `%` anchor 를 사용할수 있습니다.

```
$ AA=( "Arch Linux" "Ubuntu Linux" "Suse Linux" "Fedora Linux" )

# 전체원소들의 'u' 문자가 'X' 로 바뀔것을 볼수있습니다.
# 그런데 Ubuntu Linux 와 Suse Linux 에서는 첫자만 바뀌고 나머지는 바뀌지 않았습니다.

$ echo "${AA[@]/u/X}"
Arch LinXx UbXntu Linux SXse Linux Fedora LinXx

# 원소 전체에 적용하려면 '//pattern' 을 사용합니다.

$ echo "${AA[@]//u/X}"
Arch LinXx UbXntX LinXx SXse LinXx Fedora LinXx

# Su* 패턴과 매칭되는 원소가 없어졌습니다.
# 이것은 원소가 삭제된것이 아니라 공백으로 치환된 것입니다.

$ echo "${AA[@]/Su*/}"
Arch Linux Ubuntu Linux Fedora Linux

$ AA=( "${AA[@]/Su*/}" )

$ echo ${#AA[@]} # 원소개수가 4 개로 그대로다.
4

$ for v in "${AA[@]}"; do echo "$v"; done
Arch Linux
Ubuntu Linux
Fedora Linux # index 2 는 공백으로 나온다.
```

패턴을 이용해 매칭되는 원소 삭제하기

```
$ AA=( "Arch Linux" "Ubuntu Linux" "Suse Linux" "Fedora Linux" )

# "${AA[*]}" 는 "elem1Xelem2Xelem3X..." 와 같습니다.
# 그러므로 IFS 값을 '\n' 바꾸고 echo 한것을 명령치환 값으로 보내면
# '\n' 에 의해 원소들이 분리되어 array 에 저장되게 됩니다.

$ set -f; IFS=$'\n'
$ AA=( $(echo "${AA[*]}/Su*/}") )
$ set +f; IFS=$' \t\n' # array 입력이 완료되었으므로 IFS 값 복구

$ echo ${#AA[@]} # 삭제가 반영되어 원소개수가 3 개로 나온다
3

$ echo "${AA[1]} ${AA[2]}" # index 도 정렬되었다.
Ubuntu Linux Fedora Linux
```

스트링에서 특정 문자를 구분자로 하여 필드 분리하기.


```

$ AA="Arch Linux:Ubuntu Linux:Suse Linux:Fedora Linux"

$ IFS=: read -ra ARR <<< "$AA"
$ echo ${#ARR[@]}
4
$ echo "${ARR[1]}"
Ubuntu Linux

-----

# 입력되는 원소값에 glob 문자가 있을경우 globbing 이 발생할수 있으므로
# noglob 옵션 설정

$ set -f; IFS=:          # IFS 값을 ':' 로 설정
$ ARR=( $AA )
$ set +f; IFS=$' \t\n'   # array 입력이 완료되었으므로 IFS 값 복구

$ echo ${#ARR[@]}
4
$ echo "${ARR[1]}"
Ubuntu Linux

-----

# array 는 아니지만 다음과 같이 할수도 있습니다.

$ set -f; IFS=:          # globbing 을 disable
$ set -- $AA             # IFS 값에 따라 원소들을 분리하여
$ set +f; IFS=$' \t\n'   # positional parameters 에 할당

$ echo $#
4
$ echo "$2"
Ubuntu Linux

```

라인을 array 로 읽어들이기

```

$ cat datafile
100 Emma    Thomas
200 Alex    Jason
300 Madison Randy

$ mapfile -t arr < datafile

$ echo "${arr[0]}"
100 Emma    Thomas

$ echo "${arr[1]}"
200 Alex    Jason

```

Array index 에서 산술연산을 할 수 있다.

산술연산 특수표현식 에서 처럼 `[]` 에서도 동일하게 산술연산을 할 수 있습니다.

```
$ AA=(1 2 3 4 5)
$ idx=2
$ echo "${AA[idx == 3 ? 1 : 2]}"
3
```

Test

터미널을 열면 shell 프롬프트가 뜨는데요. 여기서 많은 명령문을 작성해서 실행해 보셨을 겁니다. shell script 는 이 명령문을 다룹니다. 명령문을 작성할때 인수로 숫자와 스트링을 구분하신 적이 있나요? 명령문을 작성할때 사칙연산 연산자를 이용해 산술연산을 하신 적이 있나요? 산술연산이 필요하면 `expr`, `bc` 같은 외부명령을 사용하지 직접 명령문 상에서 하지는 않죠. shell script 에는 기본적으로 숫자와 스트링을 구분하는 데이터 타입이 없고 사칙연산을 위한 연산자도 없습니다. 명령문 상의 모든 문자는 스트링이며 산술연산은 별도의 표현식을 통해 제공됩니다.

```
# 기본적으로 명령문에 사용되는 문자는 모두 스트링이다.

# 명령행의 인수를 스트링으로 받는 c 프로그램의 main 함수
int main(int argc, char **argv)

# 명령행의 인수를 스트링으로 받는 java 프로그램의 main 함수
public static void main(String[] args)
```

32 는 숫자인가 스트링인가? "32" 는 스트링인가 숫자인가? 일반적으로 프로그래밍 언어에서는 숫자와 스트링의 구분이 명확하여 혼용할 경우 오류가 발생합니다. 하지만 shell 에서는 데이터 타입이 존재하지 않기 때문에 32 , "32" 는 둘 다 같은 값입니다. 스트링을 다루는 곳에서 사용되면 스트링으로 사용되고 산술연산을 하는 곳에서 사용되면 숫자로 사용되는 식입니다.

```
# 숫자가 들어갈 자리에 "16" 을 스트링이 들어갈 자리에 16 을 넣어도된다
$ printf "decimal: %d, float: %g, hex: %x, string: %s\n" "-16" "16.1" "16" 16
decimal: -16, float: 16.1, hex: 10, string: 16

# '+' 는 expr 명령에 전달되는 인수로 그냥 문자
$ expr "-16" + 10
-6

# '*' 는 glob 문자이므로 escape
$ echo "10" + 2 \* 5 | bc
20

# '-gt' 는 숫자를 다루는 연산자인데 "150" 과 비교해도 된다.
$ [ "150" -gt 25 ]; echo $?
0

# 두개의 피연산자 모두 quote 해도 상관없다.
$ [ "150" -gt "25" ]; echo $?
0
```

데이터에 타입이 없기 때문에 shell script의 특징 중에 하나가 두 종류의 연산자를 제공한다는 것입니다. 하나는 숫자로 취급할때 사용되는 연산자 또 하나는 스트링으로 취급할때 사용되는 연산자입니다. 아래는 두값을 비교할때 사용되는 `test` 명령의 `help` 내용중 일부인데 두 종류의 연산자를 제공하는 것을 볼수있습니다.

```
String operators: # 스트링 으로 취급할때 사용

-z STRING      True if string is empty.

-n STRING
STRING        True if string is not empty.

STRING1 = STRING2
              True if the strings are equal.
STRING1 != STRING2
              True if the strings are not equal.
STRING1 < STRING2
              True if STRING1 sorts before STRING2 lexicographically.
STRING1 > STRING2
              True if STRING1 sorts after STRING2 lexicographically.

# 숫자로 취급할때 사용
arg1 OP arg2   Arithmetic tests.  OP is one of -eq, -ne,
               -lt, -le, -gt, or -ge.
```

[], test

스크립트를 작성할때 거의 빠짐없이 등장하는 표현식이 테스트 표현식인데요. 이때 사용하는 명령이 `test`, `[` 입니다. `test` 와 `[` 는 동일한 명령입니다. `[` 은 키워드같이 생겼지만 명령으로 사용법도 `command arg1 arg2 ...` 처럼 일반 명령과 같습니다. 따라서 위의 `help` 문서중에 나오는 연산자들은 `[` 명령의 인수이고 마지막에 붙이는 `]` 도 인수가 됩니다.

```
# test 와 '[' 는 같은 명령이다.

$ test -d /home/user/foo; echo $?
1

$ [ -d /home/user/foo ]; echo $?
1
```

null 이 아닌 값은 모두 true

`[` 명령에서 연산자를 사용하지 않을 경우 존재하지 않거나 null 값인 경우는 `false` 그외는 모두 `true` 에 해당합니다.

```
##### 거짓인 경우 #####

$ [ ]; echo $?
1
$ [ "" ]; echo $?
1
$ [ "$asdfgh" ]; echo $? # 존재하지 않는 변수
1

$ AA=""
$ [ "$AA" ]; echo $? # null 값 변수
1

##### null 이 아닌 값은 모두 true 이다 #####

$ [ 0 ]; echo $?
0
$ [ 1 ]; echo $?
0
$ [ a ]; echo $?
0
$ [ -n ]; echo $?
0
```

false 값이 0 ?

다음 예에서는 true 와 false 모두 테스트 결과가 0 이 나오고 있습니다. 프로그래밍 언어에서 true 와 false 는 키워드나 예약어로 자체 값을 갖지만 shell 에서는 builtin 명령입니다. 그래서 실행이 되어 true 는 0 을 반환하고 false 는 1 을 반환합니다. 그런데 아래서는 [명령의 인수로 사용되어 "false" 스트링과 같은 의미가 되었습니다.shell 에서 null 이 아닌 스트링은 항상 true 입니다.

역사적으로 Bourne shell 에는 true, false 명령이 없었고 true 는 colon 명령 : 의 alias 였다고 합니다.

```
$ [ true ]; echo $?
0
$ [ false ]; echo $?
0

# if 문을 사용하면 true, false 명령이 실행됩니다.
$ if false; then echo true; else echo false; fi
false
$ if true; then echo true; else echo false; fi
true
```

비교하는 변수는 quote 해야합니다.

[명령은 인수로 사용하는 변수를 quote 하지 않으면 정상적인 결과가 나오지 않을 수 있습니다.

아래 첫번째 예는 값이 없는 변수 \$AA 를 사용할때 quote 하지 않아서 결과적으로 [-n \$AA] 표현식은 [-n] 와 같게 되었습니다. 여기서 -n 는 스트링 "-n" 으로 해석되니까 항상 true 가 됩니다. 두번째는 quote 하였기 때문에 표현식이 [-n ""] 와 같게 되어 정상적인 값이 나오게 됩니다.

```
$ AA="" # 변수 AA 값은 null

$ [ -n $AA ]; echo $? # null 이 아니어야 true 인데 결과로 true 가 나왔습니다.
0

$ [ -n "$AA" ]; echo $? # 변수를 quote 해주니 정상적인 값이 나왔습니다.
1
```

다음 예에서는 변수 AA 값이 null 이 아니기 때문에 echo 명령이 실행돼야 하지만 구문 오류가 발생하였습니다. -n 연산자는 하나의 피연산자를 갖는데 \$AA 를 quote 하지 않아서 결과적으로 [-n hello world] 가되어 두개의 피연산자를 갖게 되었습니다.

```
$ AA="hello world"

$ if [ -n $AA ]; then
    echo "$AA"
fi
bash: [: too many arguments
```

위의 예를 통해서 알 수 있듯이 [명령을 사용할 때는 항상 변수를 quote 해줘야 합니다.

숫자를 비교할때 스트링 연산자를 사용하면 안된다.

< , > 연산자는 보통 프로그래밍 언어에서는 숫자를 비교할때 사용하지만 [] , [[]] 에서는 그와 달리 스트링을 비교하는데 사용합니다. 위의 help 문서중에 나왔듯이 스트링을 비교할 때는 사전적으로 (lexicographically) 비교를 합니다. 그러므로 "100" 보다는 "2" 가 큰수가 됩니다. 왜냐하면 사전적으로 볼 때 처음 비교되는 문자가 "1" 보다 "2" 가 크기 때문입니다.

```
$ [ 100 \> 2 ]; echo $?
1

$ [ 100 -gt 2 ]; echo $?
0
```

AND , OR

[명령에서는 and, or 연산자로 `-a` , `-o` 를 제공하지만, `&&` || shell 메타문자를 이용해 분리해서 사용할 수도 있습니다. 두 연산자의 우선순위는 [명령에서 제공하는 연산자일 경우 일반 프로그래밍 언어와 같지만, shell 메타문자를 이용할 경우는 두 메타문자의 우선순위를 같게 취급하므로 주의할 필요가 있습니다 (자세한 내용은 shell metacharacters precedence 참조).

```
# shell 메타문자를 이용해 분리해 사용

if [ test1 ] && [ test2 ]; then ...

if [ test1 ] || [ test2 ]; then ...

# { ; } 를 이용해 우선순위 조절
if [ test1 ] || { [ test2 ] && [ test3 ] ;} then ...

-----

if [ test1 -a test2 ]; then ...

if [ test1 -o test2 ]; then ...

# 우선순위 조절을 위해 `( )` 를 사용할때는 shell 메타문자와 충돌하므로 escape 합니다.
if [ test1 -a \( test2 -o test3 \) ]; then ...
```

Logical NOT

다음과 같이 두가지 형태로 사용할 수 있습니다.

```
# '[' 명령에서 사용되는 '!'
if [ ! test1 ]; then ...

# shell logical NOT 키워드를 이용
if ! [ test1 ]; then ...
```

Array 를 비교할때는 * 을 사용

Quote 을 하지 않을경우 `${array[@]}` 와 `${array[*]}` 는 차이가 없습니다. 어차피 똑같이 IFS 값에 의해 단어 분리가 되기 때문입니다. 하지만 " " 하였을 경우는 @ 과 * 의 의미가 틀려집니다. @ 은 개개의 원소를 " " 하여 나열하는 것과 같고 * 은 모든 원소를 하나의 " " 안에 넣는 것과 같습니다. 그러므로 array 를 비교할 때는 * 을 사용해야 합니다.

```
$ AA=(11 22 33)
$ BB=(11 22 33)

$ [ "${AA[*]}" = "${BB[*]}" ]; echo $?
0

$ [ "${AA[@]}" = "${BB[@]}" ]; echo $?
bash: [: too many arguments
```

[[]]

이것은 생긴모양에서 알수있듯이 `[]` 의 기능확장 버전입니다. `[]` 와 가장 큰 차이점은 `[]` 은 명령이고 `[[]]` 은 shell keyword 라는 점입니다. keyword 이기 때문에 일반 명령들과 달리 shell 에서 자체적으로 해석을 하고 실행하기 때문에 `[]` 처럼 명령이라서 생기는 여러가지 제약사항 없이 편리하게 사용할 수 있습니다. (좀 더 자세한 내용은 Special Expressions 참조).

```
# '<' , '>' 연산자를 quote 하지 않아도 되는 것을 알 수 있습니다.
$ [[ a < b ]]; echo $?
0
$ [[ a > b ]]; echo $?
1

# 사용되는 변수를 quote 하지 않아도 값을 올바르게 인식한다.
$ AA=""
$ [[ -n $AA ]]; echo $?
1
$ [[ -n "$AA" ]]; echo $?
1
```


Test Operators

이 연산자들은 `test`, `[builtin` 명령에서 제공하는 것으로 `[[]]` 에서도 동일하게 사용할 수 있습니다. (단 `-a` , `-o` 연산자는 제외)

File Tests

`test` 되는 파일이 symbolic link 일 경우 `-L` , `-h` 연산자는 링크 자체를 테스트하고 나머지는 링크에 연결된 대상 파일을 테스트합니다. 그러므로 AA 가 symbolick link 이고 BB.sh 파일에 연결돼 있다면 `[-L AA]` , `[-f AA]` 는 모두 `true` 가 됩니다.

| 연산자 | 설명 |
|-------------------------------|--|
| -a <FILE> | 파일이 존재하면 true 입니다. (Logical AND 연산자와 구분이 어려우므로 사용하지 않는 것을 권장.) |
| -e <FILE> | 파일이 존재하면 true 입니다. |
| -f <FILE> | 파일이 존재하고 regular 파일이면 true 입니다. |
| -d <FILE> | 파일이 존재하고 directory 이면 true 입니다. |
| -c <FILE> | 파일이 존재하고 character special 파일이면 true 입니다. (터미널도 character special 에 해당) |
| -b <FILE> | 파일이 존재하고 block special 파일이면 true 입니다. |
| -p <FILE> | 파일이 존재하고 (named, unnamed) pipe 이면 true 입니다. |
| -S <FILE> | 파일이 존재하고 socket 이면 true 입니다. |
| -L <FILE> | 파일이 존재하고 symbolic link 이면 true 입니다. |
| -h <FILE> | 파일이 존재하고 symbolic link 이면 true 입니다. |
| -g <FILE> | 파일이 존재하고 sgid bit 이 설정돼 있으면 true 입니다. |
| -u <FILE> | 파일이 존재하고 suid bit 이 설정돼 있으면 true 입니다. |
| -k <FILE> | 파일이 존재하고 sticky bit 이 설정돼 있으면 true 입니다. |
| -r <FILE> | 파일이 존재하고 readable 이면 true 입니다. |
| -w <FILE> | 파일이 존재하고 writable 이면 true 입니다. |
| -x <FILE> | 파일이 존재하고 executable 이면 true 입니다. |
| -O <FILE> | 파일이 존재하고 uid (user id) 가 같으면 true 입니다. |
| -G <FILE> | 파일이 존재하고 gid (group id) 가 같으면 true 입니다. |
| -N <FILE> | 파일이 존재하고 마지막에 read 한뒤로 modify 되었으면 true 입니다. |
| -s <FILE> | 파일이 존재하고 사이즈가 0 보다 크면 (not empty) true 입니다. |
| -t <fd> | FD 가 존재하고 현재 터미널에 연결돼 있으면 true 입니다. |
| <FILE1> -nt <FILE2> | FILE1 이 FILE2 보다 수정시간이 newer 면 true 입니다. |
| <FILE1> -ot <FILE2> | FILE1 이 FILE2 보다 수정시간이 older 면 true 입니다. |
| <FILE1> -ef <FILE2> | FILE1 과 FILE2 가 서로 hardlink 되어 있으면 true 입니다. |

String Tests

보통 프로그래밍 언어에서 `<` , `>` 는 숫자를 비교할때 사용되는데 shell 에서는 특이하게 스트링을 비교하는데 사용됩니다. 생각 같아서는 스트링 연산자와 산술연산자가 서로 바뀌었으면 헷갈리지 않고 좋을것 같은데 사실 이것도 알고보면 shell 스트립트가 갖는 근본적인 한계입니다. 왜냐하면 shell 에서는 `<=` , `>=` 기호를 사용할수가 없습니다. 저렇게 사용하게 되면 `=` 가 파일명이 되고 `<` , `>` 가 redirection 기호가 되기 때문입니다.

`<` , `>` 연산자는 사전적으로 (lexicographically) 비교하기 때문에 숫자를 비교하는데 사용하면 안됩니다 (100 보다 2 가 크다고 나옵니다). 또한 shell redirection 연산자와 충돌하므로 사용할때 escape 해야 합니다.

| 연산자 | 설명 |
|---|---|
| <code>-z <STRING></code> | 스트링이 null 값을 가지고 있으면 true 입니다. (변수가 존재하지 않는 경우에도 해당됩니다.) |
| <code>-n <STRING></code> | 스트링이 null 이외의 값을 가지고 있으면 true 입니다. |
| <code><STRING1> = <STRING2></code> | 두 스트링 값이 같으면 true 입니다. |
| <code><STRING1> != <STRING2></code> | 두 스트링 값이 다르면 true 입니다. |
| <code><STRING1> < <STRING2></code> | 사전적으로 비교했을 때 스트링1 이 작으면 true 입니다. (사용할때 escape 합니다.) |
| <code><STRING1> > <STRING2></code> | 사전적으로 비교했을 때 스트링1 이 크면 true 입니다. (사용할때 escape 합니다.) |

Arithmetic Tests

| 연산자 | 설명 |
|--|--|
| <code><INTEGER1> -eq <INTEGER2></code> | 두 수가 같으면 true 입니다. |
| <code><INTEGER1> -ne <INTEGER2></code> | 두 수가 같지 않으면 true 입니다. |
| <code><INTEGER1> -le <INTEGER2></code> | INT1 이 INT2 보다 작거나, 두 수가 같으면 true 입니다. |
| <code><INTEGER1> -ge <INTEGER2></code> | INT1 이 INT2 보다 크거나, 두 수가 같으면 true 입니다. |
| <code><INTEGER1> -lt <INTEGER2></code> | INT1 이 INT2 보다 작으면 true 입니다. |
| <code><INTEGER1> -gt <INTEGER2></code> | INT1 이 INT2 보다 크면 true 입니다. |

Misc Syntax

| 연산자 | 설명 |
|------------------------------|--|
| <TEST1> -a <TEST2> | 두 테스트 간에 AND 연산을 할 때 사용합니다. -a 연산자는 파일 테스트에서도 사용되므로 주의해야 합니다. |
| <TEST1> -o <TEST2> | 두 테스트 간에 OR 연산을 할 때 사용합니다. |
| ! <TEST> | Logical NOT 연산을 합니다. |
| (<TEST>) | 우선순위 조절을 위해 사용할 수 있습니다 (사용할때 escape 합니다.) |
| -o <OPTION_NAME> | set builtin 명령으로 설정하는 옵션의 현재 값을 테스트할 때 사용합니다. 현재 -o 상태면 true, +o 상태면 false 입니다. |
| -v <VARIABLENAME> | 변수가 존재하는지 테스트할 때 사용합니다. 존재하지 않는 상태 (unset 상태) 면 1 을, 그 외에는 0 을 리턴합니다. |
| -R <VARIABLENAME> | 변수가 named reference 이면 true 입니다. (bash version 4.3+). |

Subshells

Shell 에서 명령을 실행하면 새로운 프로세스가 생성되어 실행됩니다. 이때 명령을 호출한 process 가 parent 가 되고 새로 실행되는 명령이 child process (subprocess) 가 됩니다. 다음은 프롬프트 상에서 `/bin/sleep` 외부 명령을 실행한 예인데 현재 bash shell process 아래서 sleep 명령의 child process 가 실행되는 것을 볼 수 있습니다.

| PID | TTY | STAT | TIME | COMMAND |
|-------|-------|------|------|-------------|
| 25659 | pts/1 | Ss | 0:01 | bash |
| 6089 | pts/1 | S+ | 0:00 | _ sleep 10 |

이번에는 AA.sh 라는 shell script 를 만들어 실행시킨 예인데 스크립트 실행을 위해 bash child process 가 하나 더 생성되고 그 아래서 sleep 명령이 실행되는 것을 볼 수 있습니다.

| PID | TTY | STAT | TIME | COMMAND |
|-------|-------|------|------|----------------------|
| 20685 | pts/5 | Ss | 0:00 | bash |
| 32332 | pts/5 | S+ | 0:00 | _ /bin/bash ./AA.sh |
| 32333 | pts/5 | S+ | 0:00 | _ sleep 10 |

이번에는 프롬프트에서 아래와 같은 4 종류의 명령을 실행한 결과입니다. 그런데 위에서처럼 shell script 를 실행한 것도 아닌데 bash child process 가 하나 더 생긴 후에 그 아래서 sleep 명령이 실행되는 것을 볼 수 있습니다. 이렇게 `()` `$()` `` `` `|` `&` 를 이용하여 명령을 실행시킬 때 생성되는 shell 을 subshell 이라고 합니다.

| | |
|--|--|
| <code>\$ (echo; sleep 10)</code> | # 1. <code>()</code> , <code>\$()</code> |
| <code>\$ `echo; sleep 10`</code> | # 2. <code>` `</code> backtick 명령치환 |
| <code>\$ echo { echo; sleep 10; }</code> | # 3. <code> </code> 파이프 |
| <code>\$ shellfunc & 또는 { comm1; comm2 ...; } &</code> | # 4. background 로 실행되는 함수 및 명령 group |

| PID | TTY | STAT | TIME | COMMAND |
|-------|-------|------|------|-------------|
| 25659 | pts/1 | Ss | 0:01 | bash |
| 6161 | pts/1 | S+ | 0:00 | _ bash |
| 6162 | pts/1 | S+ | 0:00 | _ sleep 10 |

Child process 가 parent process 로부터 물려받는 것들

- 현재 디렉토리
- export 된 환경 변수, 함수
- 현재 설정되어 있는 file descriptor 들 (stdin(0), stdout(1), stderr(2) ...)

- ignore 된 신호 (trap " INT)

테스트를 위해 터미널에서 실행한 명령

```
$ echo "pid : $$, PPID : $PPID"
pid : 10875, PPID : 1499          # 현재 shell pid 와 ppid

$ pwd                             # 현재 working 디렉토리
/home/mug896/tmp

$ var1=100 var2=200
$ export var1                     # var1 만 export

$ f1() { echo "I am exported function" ;}
$ f2() { echo "I am not exported function" ;}
$ export -f f1                   # f1 만 export

$ trap '' INT                    # INT 신호 ignore
$ trap 'rm -f /tmp/tmpfile' SIGTERM

$ exec 3> /dev/null               # FD 3 생성
```

child process 생성을 위한 test.sh 파일 내용과 실행 결과

```

----- test.sh -----
#!/bin/bash

echo "pid : $$, PPID : $PPID"
pwd
echo "var1 : $var1"
echo "var2 : $var2"
f1
f2
trap
ls -l /proc/$$/fd
-----

$ ./test.sh
pid : 11717, PPID : 10875          # ppid 는 parent shell pid
/home/mug896/tmp
var1 : 100
var2 :                            # export 안된 변수
I am exported function
./test.sh: line 6: f2: command not found # export 안된 함수
trap -- '' SIGINT                # ignore 된 신호만 표시됨
total 0
lrwx----- 1 mug896 mug896 64 Aug  8 16:07 0 -> /dev/pts/13 # FD 0,1,2 는 그대로 물려받음
lrwx----- 1 mug896 mug896 64 Aug  8 16:07 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 Aug  8 16:07 2 -> /dev/pts/13
l-wx----- 1 mug896 mug896 64 Aug  8 16:07 3 -> /dev/null # parent shell 에서 open 했던 FD

```

Subshell 이 추가해서 물려받는 것들

- export 안된 현재 shell 에서 사용중인 변수, 함수들
- trap handler 설정 (trap 'rm -f tmpfile' INT)
- 현재 pid 를 나타내는 \$\$ 변수값
- parent pid 를 나타내는 \$PPID 변수값

다시말해 subshell 은 현재 shell 환경을 거의 그대로 물려받는다고 생각하면 됩니다. subshell 에서 테스트를 실행한 결과는 현재 shell 에서 실행한 것과 차이가 없습니다.

```

$ ( echo "pid : $$, PPID : $PPID"
> pwd
> echo "var1 : $var1"
> echo "var2 : $var2"
> f1
> f2
> trap
> ls -l /proc/$BASHPID/fd )

pid : 10875, PPID : 1499          # pid, ppid 가 현재 shell 과 동일하게 나온다
/home/mug896/tmp
var1 : 100
var2 : 200
I am exported function
I am not exported function
trap -- '' SIGINT
trap -- 'rm -f /tmp/tmpfile' SIGTERM
trap -- '' SIGTSTP
trap -- '' SIGTTIN
trap -- '' SIGTTOU
total 0
lrwx----- 1 mug896 mug896 64 08.08.2015 15:20 0 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 08.08.2015 15:20 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 08.08.2015 15:20 2 -> /dev/pts/13
l-wx----- 1 mug896 mug896 64 08.08.2015 15:20 3 -> /dev/null

```

Parent process 와 child process 의 관계

- parent 에서 물려받은 값들은 child 에서 마음대로 읽고 쓸수 있으나 변경한 값이 parent 에 적용되지는 않습니다. child 에서 export 를 해도 child 의 child 에게 적용이 되지 parent 에는 적용되지 않습니다.
- child 에서 shell 환경 변수를 변경하거나 옵션 설정을 변경해도 parent 에는 영향을 미치지 않습니다.
- parent 에서 특정 값을 변경해도 그것이 이미 실행 중인 child 에 반영되지 않습니다.

Subshell 의 특징

parent process 에서 설정한 변수나 함수는 export 해야지 child process 에서 사용할 수 있습니다. 하지만 subshell 에서는 export 하지 않아도 사용할 수가 있습니다.

```

$ AA=100
$ ( echo AA value = "$AA" )    # 변수 AA 를 export 하지 않아도 값을 사용할 수 있다.
AA value = 100

```


현재 shell 에서 사용중인 변수는 subshell 에서 읽고, 쓰고 할 수 있습니다. 그러나 변경된 값이 현재 shell 에 적용되지는 않으므로 주의해야 합니다.

```
$ AA=100

$ ( AA=200; echo AA value = "$AA" )    # subshell 에서 값을 200 으로 변경.
AA value = 200

$ echo "$AA"                           # 하지만 현재 shell 에서는 변함이없다.
100
```

subshell 을 생성하여 사용한 변수나 shell 환경 설정 변경은 subshell 이 종료되면 사라집니다.

```
$ echo -n "$IFS" | od -a
00000000 sp ht nl

# subshell 에서 IFS 값을 ':' 로 변경하여 사용함.
$ ( IFS=:; echo -n "$IFS" | od -a )
00000000 :

# subshell 이 종료된 후에는 기존의 IFS 값으로 복귀 되었다.
$ echo -n "$IFS" | od -a
00000000 sp ht nl

#####

$ [ -o noglob ]; echo $?
1

# subshell 에서 셸옵션을 변경하여 사용.
$ ( set -o noglob; [ -o noglob ]; echo $? )
0

# subshell 이 종료후엔 이전상태로 복귀되었다.
$ [ -o noglob ]; echo $?
1

#####

$ set -- 11 22 33
$ echo "$@"
11 22 33

# positional parameters 를 subshell 에서 설정하여 사용
$ ( set -- 44 55 66; echo "$@" )
44 55 66

# subshell 종료후 기존 값으로 복귀
$ echo "$@"
11 22 33

#####

$ ulimit -c
0

# core file 생성을 위해 subshell 을 이용해 일시적으로 ulimit 값을 설정
$ ( ulimit -c unlimited; ulimit -c ; ... )
unlimited

# subshell 종료후 'ulimit -c' 값이 0 으로 복귀되었다
$ ulimit -c
0
```

subshell 에서 `cd` 한것은 종료후 이전 상태로 돌아옵니다.

```
$ pwd
/home/test/tmp

# subshell 을 이용하여 cd 사용
$ ( cd ~/tmp2; pwd; ... )
/home/test/tmp2

# subshell 종료후 이전 상태로 복귀
$ pwd
/home/test/tmp
```

subshell 에서도 `exit` 명령을 사용하여 종료할 수 있습니다.

```
$ ( echo hello; exit 3; echo world )
hello

$ echo $?
3
```

디버깅을 위해 `-E | set -o errtrace` or `-T | set -o functrace` 옵션을 이용해 `trace` 할때도 subshell 까지만 됩니다. child process 는 trace 되지 않습니다.

\$\$ 와 \$BASHPID

`$$` 변수는 현재 shell pid 를 나타내는 변수인데 subshell 에서도 동일한 값을 가집니다. 하지만 subshell 도 엄연히 프로세스 이므로 pid 값을 가지는데 `$BASHPID` 변수를 통해서 구할 수 있습니다.

\$SHLVL 과 \$BASH_SUBSHELL

SHLVL 은 child process 를, BASH_SUBSHELL 은 subshell 을 의미합니다.

1. 프롬프트 상일 경우.
SHLVL 값은 1 (shell process 가 실행상태 이므로), BASH_SUBSHELL 값은 0 이 됩니다.
2. `()` subshell 메타문자를 이용해서 실행
BASH_SUBSHELL 값이 올라갑니다.
3. shell script 를 작성해서 실행
이경우는 child process 가 생성되고 SHLVL 값이 올라갑니다.
4. `bash -c 'command ...' [arg1 arg2 ...]`
이때도 child process 가 생성되고 SHLVL 값이 올라갑니다.

Subshell 주의할점

다음은 shell script 작성시 subshell 로인해 범하기 쉬운 오류들에 대한 예제입니다.

예제 .1

```
#!/bin/bash

index=30

change_index() {
    index=40
    echo "changed to 40"
}

result=$(change_index)

echo $result
echo $index

##### output #####

changed to 40
30
```

결과값으로 "changed to 40", 40 을 기대하였으나 "changed to 40", 30 이 나왔습니다.

change_index 함수가 () 안에서 실행됐으므로 subshell 에서 실행되어 parent 변수인 index 값을 갱신할 수 없습니다.

예제 .2

```
$ echo hello | read var; echo $var
$

$ echo hello | { read var; echo $var; }
$ hello
```

echo hello 와 read var 는 | 에 의해 연결돼 있으므로 각각 subshell 에서 실행이 되고 종료 후에 echo \$var 가 실행되게 됩니다. subshell 에서 설정한 변수값을 echo \$var 에서는 읽을 수 없습니다. 두번째와 같이 { } 를 이용한 명령 grouping 을 사용하면 echo \$var 도 함께 subshell 에서 실행되어 원하는 값을 얻을 수 있습니다.

예제 .3

```
#!/bin/bash

nKeys=0

cat datafile | while read -r line
do
    #...do stuff
    nKeys=$((nKeys + 1))
done

echo Finished writing $nKeys keys

##### output #####

Finished writing 0 keys
```

결과값으로 while 문으로 인해 증가한 nkeys 값을 기대하였으나 마지막 문장의 \$nkeys 값은 0 이 표시 되었습니다. while 구문이 파이프 | 로 인해 subshell 에서 실행되어 parent 변수인 nkeys 값을 변경할 수 없습니다. 다음과 같이 수정합니다.

```
#!/bin/bash

nKeys=0

while read -r line
do
    #...do stuff
    nKeys=$((nKeys + 1))
done < datafile

echo Finished writing $nKeys keys

##### output #####

Finished writing 10 keys
```

정리하면

현재 shell 과 subshell 이 가지는 관계나 parent process 에서 export 한 변수, 함수에 대해 child process 가 가지는 관계는 프로그래밍 언어에서 많이 사용하는 closure 와 비슷합니다. subshell 생성시에 parent 환경을 그대로 갖게 되지만 subshell 안에서 별도로 존재한다고 볼 수 있습니다. 그래서 subshell 내에서 값을 변경해도 그 값이 parent 에 적용이 되지 않고 또한 subshell 생성 후 에 parent 에서 특정 값을 변경해도 그 값이 subshell 에 반영이 되지 않습니다.

명령 의 구분

1. 외부 명령

/usr/bin/find 명령과 같이 시스템 디렉토리에 위치한 명령들입니다.

명령의 이름이 중복해서 존재할 경우 경로를 지정하여 실행할 수 있습니다.

2. shell builtins

말 그대로 shell 에 builtin 돼서 제공되는 명령들로 실행을 위해 새로 프로세스를 만들지 않아도 되기 때문에 외부 명령에 비해 좀 더 효율적으로 실행할 수 있고 shell 내부 상태정보를 변경할 수 있습니다.

```
$ compgen -b | column
```

| | | | | | | |
|-----------|----------|---------|---------|---------|----------|---------|
| . | : | [| alias | bg | bind | break |
| builtin | caller | cd | command | compgen | complete | comptop |
| continue | declare | dirs | disown | echo | enable | eval |
| exec | exit | export | false | fc | fg | getopts |
| hash | help | history | jobs | kill | let | local |
| logout | mapfile | popd | printf | pushd | pwd | read |
| readarray | readonly | return | set | shift | shopt | source |
| suspend | test | times | trap | true | type | typeset |
| ulimit | umask | unalias | unset | wait | | |

3. shell functions

사용자가 임의로 원하는 함수를 만들어 사용할 수 있습니다. 함수명은 외부 명령, builtin 명령과 동일하게 사용됩니다. `declare -F` 명령을 이용하면 현재 shell 에 설정돼있는 함수들을 볼 수 있습니다.

4. shell keywords

앞서 소개한 명령들은 사용시 `command arg1 arg2 ...` 형식을 취하고 실행전에 인수들의 확장과 치환작업을 거치나 키워드는 그에 앞서 구문을 해석할때 사용됩니다.

```
$ compgen -k | column
```

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

5. aliases

앞서 소개한 명령들을 **alias** 하여 사용할 수 있습니다. shell 키워드도 **alias** 해서 사용할 수 있을 정도로 우선순위가 제일 높습니다. Non-interactive shell 인 script 실행시에는 기본적으로 disable 됩니다.

명령 이름이 중복되어 나타날때

명령 이름이 앞서 분류한 곳에 중복되어 나타나는 경우가 있습니다.
관련된 정보를 보려면 builtin 명령인 **type** 을 이용합니다.

```
$ type -a kill
kill is a shell builtin
kill is /bin/kill

$ type -a time
time is a shell keyword
time is /usr/bin/time

$ type -a [
[ is a shell builtin
[ is /usr/bin/
```

명령을 찾는 우선순위는 alias, function, builtin, 외부 명령 순 입니다. kill 명령은 builtin 에도 있고 외부 명령에도 있는데 같은 이름의 function 을 새로 만든다면 function 이 실행되게 됩니다.

이렇게 중복되는 이름 문제를 해결하기 위해 shell 에서는 다음과 같은 명령들을 제공합니다.

- **command** : 우선순위가 높은 alias, function 이름을 피해 외부 명령을 (builtin 명령포함) 실행합니다.
외부 명령의 기능을 확장하기 위해 동일한 이름의 함수를 만들어 사용할땐 함수안에서 `command 명령` 을 사용해야 외부 명령을 실행할 수 있습니다.
- **builtin** : 우선순위가 높은 alias, function 이름을 피해 builtin 명령을 실행하기 위해 사용합니다.
- **enable** : builtin 명령을 disable 하여 외부 명령을 실행하게 할 수 있습니다.

alias , keyword 의 escape

alias 나 keyword 는 명령이 아니라 특별한 기능을 하는 단어로 볼 수 있습니다. 우선순위 또한 일반 명령들 보다 높는데 다음과 같은 방법을 이용하면 해당 기능을 disable 할 수 있습니다.

- alias, keyword 이름 앞에 `\` 문자를 붙인다.

- alias, keyword 이름을 quote 한다

time 은 keyword 이면서 /usr/bin/time 외부 명령이기도 한데 다음과 같이 하면 키워드 기능이 escape 되어 외부 명령을 실행할 수 있습니다.

```
$ \time
$ 'time'      # 또는 "time"
```

ls 명령이 alias 되어 있다면 다음 명령으로 alias 기능을 escape 하여 원본 명령을 실행할 수 있습니다.

```
$ \ls
$ 'ls'      # 또는 "ls"
```

function 이나 builtin 명령은 특수한 기능을 하는 단어로 보지 않고 일반명령들과 동일하게 취급하므로 위 방법으로 escape 할 수 없습니다. builtin 명령과 중복되는 외부 명령이 있다면 전체 경로를 입력하여 실행하거나 enable 명령으로 builtin 명령을 disable 할 수 있습니다.

명령들의 차이점

예제 .1

다음은 shell builtin `[` 명령과 `[[` 키워드의 사용상 차이점입니다.

```
$ AA="hello world"

$ if [ -n $AA ]; then
>     echo "$AA"
> fi
bash: [: hello: binary operator expected

-----

$ if [[ -n $AA ]]; then
>     echo "$AA"
> fi
hello world
```

똑같은 문장인데 `[` 명령에서는 오류가 나고 `[[` 키워드에서는 정상적으로 실행이 됩니다. `[` 는 앞서 살펴본 바와같이 builtin 명령으로 일반 명령들과 사용법이 같습니다. `command arg1 arg2..` 이런식이죠. 그러므로 `[-n $AA]` 명령문이 변수확장이 되면 `[-n hello world]` 와 같

계됩니다. 결과적으로 `hello` 를 binary 연산자로 `-n` 와 `world` 를 피연산자로 해석을 해서 `hello` 라는 연산자가 없다는 오류가 납니다. 반면에 `[[` 키워드는 shell 에서 자체적으로 표현식을 해석하므로 `$AA` 값을 올바르게 해석합니다.

예제 .2

다음 예로 builtin 명령과 키워드의 실제 사용되는 시점에 대해 알수있습니다.

```
$ a='['
$ $a -d /tmp ]
$ echo $?
0
```

`$a` 변수확장이 일어나고 `[-d /tmp]` 명령문이 실행되어 정상적인 결과값이 나왔습니다.

```
$ a='[['
$ $a -d /tmp ]]
bash: [: command not found
```

`$a` 변수확장이 일어나고 `[[-d /tmp]]` 표현식이 실행되었으나 오류가 발생했습니다. 이 결과로 알 수 있는 것은 키워드 해석 작업은 변수확장 보다 우선해서 처리된다는 것입니다. 키워드 해석 작업이 이미 끝난 상태이므로 `[[` 명령이 없다는 오류가 발생했습니다.

예제 .3

`time` 명령은 shell 키워드이고 `/usr/bin/time` 명령이기도 합니다. 먼저 키워드를 이용해 다음 문장을 실행해보면 오류 없이 정상적으로 실행이 됩니다.

```
$ time { find . -name '*.sh' | wc -l ;}
24

real    0m0.002s
user    0m0.000s
sys     0m0.004s
```

이번에는 `\` 를 이용해 키워드를 escape 해서 `/usr/bin/time` 명령으로 실행해보면 외부 명령으로 는 구문 에러가 나는 것을 알 수 있습니다.

```
$ \time { find . -name '*.sh' | wc -l ;}
bash: syntax error near unexpected token `}'
```

Aliases, Functions and Scripts

Shell 을 사용하다 보면 명령을 `alias` 로할지 `function` 으로 할지 아니면 외부 명령처럼 `script` 파일로 작성할지 애매한 경우가 있는데요. 기능상의 차이점을 살펴보면 다음과 같습니다.

- `alias`, `function` 은 `shell` 이 시작되면 메모리에 올라가서 실행되며 `script` 는 외부 파일 형태로 존재합니다.
- `alias`, `function` 은 현재 `shell` 에서 사용됩니다. `function` 의 경우 `export` 하면 `child process` 까지 사용할 수 있습니다. `script` 의 경우는 파일로 존재하므로 추가적인 설정 없이 `shell` 외부에서 실행할 수 있습니다.

예를들어 `vi` 에디터에서 외부 명령을 실행할때 `alias` 나 `export` 안된 함수는 실행할 수 없습니다.

- `alias`, `function` 은 현재 `shell` 에서 실행되므로 새로 `process` 를 생성할 필요가 없고 현재 `shell` 의 환경 변수값을 바꾼다던지 옵션 설정을 변경할 수 있습니다. 반면에 `script` 는 `child process` 가 생성되어 실행되므로 `parent shell` 의 환경을 변경할 수 없습니다.
- `alias` 는 `\` 로 간단하게 `escape` 할수 있는 반면에 `function` 은 `command` 명령을 사용해야 한다.

그러므로 `shell` 프롬프트 상에서 주로 사용할 것이라면 `alias` 로 작성하고 `script` 나 `child process` 에서도 사용하려면 `function` 으로 만들어서 `export -f` 하고 `shell` 외부 환경에서도 실행 가능해야 한다면 `script` 파일로 작성하면 됩니다.

Help 문서 보는법

Shell builtins, keywords

help command

```
$ help set
set: set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg ...]
    Set or unset values of shell options and positional parameters.
...
...
```

외부 명령

1. `command --help`
2. `man` 숫자 `command`
3. `info command`

`man page` (`manual page`) 는 `unix` 에서 전통적으로 사용해온 문서 형식입니다.
뒤에 숫자를 이용해서 섹션별로 구분을 하는데 그 의미는 다음과 같습니다.

| 숫자 | 설명 |
|----|---|
| 1 | User Commands |
| 2 | System Calls |
| 3 | C Library Functions |
| 4 | Devices and Special Files (usually found in <code>/dev</code>) |
| 5 | File formats and conventions e.g <code>/etc/passwd</code> , <code>/etc/crontab</code> |
| 6 | Games |
| 7 | Miscellaneous (including macro packages and conventions) |
| 8 | System Administration tools and Deamons (usually only for root) |

man -f page

해당 페이지가 속한 섹션들을 짧은 설명과 함께 볼 수 있습니다.

```
$ man -f printf
printf (3)          - formatted output conversion
printf (1)          - format and print data
```

man -k regex

페이지 이름과 설명에서 매칭되는것을 찾아볼 수 있습니다.

```
$ man -k printf
asprintf (3)        - print to allocated string
dprintf (3)         - print to a file descriptor
fprintf (3)         - formatted output conversion
fwprintf (3)        - formatted wide-character output conversion
printf (1)          - format and print data
...
...
```

man page 는 문서에 별다른 기능이 없어서 복잡한 명령일 경우 내용이 빈약한것이 단점입니다. 그래서 1990년대 초에 GNU 에서 기존의 man page 를 대체하기 위해 나온것이 info 입니다. info 는 문서에 hyperlink 기능과 간단한 markup language 를 사용할 수 있어서 man page 보다 상세한 내용을 볼수있습니다.

```
$ info grep
```

Bourne Shell Builtins

•

`. filename [arguments]`

`filename` 을 읽어들이며 현재 `shell` 에서 실행합니다. 가령 `AA.sh` 파일내에 `. BB.sh` 가 있다면 명령이 위치한 곳에 `BB.sh` 파일의 내용을 읽어들이며 실행합니다. `child process` 가 생성되는 것은 아니고 현재 `AA.sh process` 에서 실행됩니다.

`function` 과 동일하게 인수를 줄 수 있으며 `BB.sh` 에서는 `return` 명령을 사용할 수 있습니다. 인수를 주지 않을 경우 `AA.sh` 실행시 사용된 인수가 동일하게 적용됩니다.

`filename` 에 `/` 를 이용해 경로를 지정하지 않으면 `$PATH` 를 검색하는데 찾지 못하면 오류메시지와 함께 `non-zero` 를 리턴합니다.

:

:

아무일도 하지 않는 명령입니다. 종료상태 값으로 항상 0 을 리턴합니다. 실질적으로 `true` 명령과 동일합니다 (예전에는 `true` 명령이 이 명령의 `alias` 였습니다.). 간단한 스크립트 테스트를 할때 `true` 가 올수있는 자리에 대신 사용하기에 편합니다. 또한 실행 시에 인수부분에서 변수확장, 명령치환, 산술확장이 이루어지므로 디버깅 시에 특정 값을 보기위해 사용할 수도 있습니다. 여러모로 활용성이 있는 명령입니다.

```
while ;; do echo $(( i++ )); sleep 1; done

if ;; then ;; else ;; fi

# DIR 변수가 존재하지 않거나 null 값일 경우 '/usr/local' 을 대입
: ${DIR:=/usr/local}
```

break

`break [n]`

`for`, `while`, `until` 반복구문에서 사용되는 명령입니다. 현재 실행되고 있는 반복을 중단하는 역할을 합니다. 반복구문이 중첩 될수있는데 이때 `n` 인자를 사용할수 있습니다.

cd

```
cd [-L|[-P [-e]] [-@]] [dir]
```

현재 작업중인 디렉토리를 변경합니다. `-P` 옵션을 주면 심볼릭 링크 디렉토리 구조를 따르지 않고 물리적 디렉토리 구조를 따릅니다

continue

```
continue [n]
```

for, while, until 반복구문에서 사용되는 명령입니다. `continue` 명령이 실행되면 현재 반복을 중단하고 다음반복으로 넘어갑니다. 반복구문이 중첩 될수있는데 이때 `n` 인자를 사용할수 있습니다.

eval

```
eval [arg ...]
```

[eval](#) 메뉴 참조.

exec

```
exec [-cl] [-a name] [command [arguments ...]] [redirection ...]
```

- 현재 shell process 를 command 로 대체합니다. command 를 실행할수 없을경우 스크립트는 exit 됩니다. (exit 을 방지하기 위해 `execfail` 옵션을 사용할수 있습니다.)
- 현재 shell 에 redirection 을 적용시킵니다.

exit

```
exit [n]
```

shell 을 exit 합니다. `n` 은 종료상태 값이 되며 `$?` 를 통해 구할수 있습니다. `n` 값을 설정하지 않으면 이전 명령의 종료값이 사용됩니다.

export

```
export [-fn] [name[=value] ...] or export -p
```

AA.sh 스크립트 내에서 BB.sh 스크립트를 실행시키면 (`source` 가 아님) AA.sh 은 parent, BB.sh 은 child process 가 됩니다. 이때 AA.sh 에서 설정한 변수라든지 함수들은 바로 BB.sh 에서 사용할 수가 없는데 `export` 명령을 이용하면 child process 인 BB.sh 에서도 사용할 수가 있게 됩니다.

AA.sh 에서 `export` 한 변수를 BB.sh 에서 읽고 수정하고 할 수 있지만 변경된 사항이 parent process 인 AA.sh 에 적용되지는 않습니다.

함수를 export 할 때는 `-f` 옵션을 사용합니다.

getopts

```
getopts optstring name [arg]
```

명령행에서 사용되는 옵션들을 처리하기 위해 사용되는 명령입니다.

hash

```
hash [-lr] [-p pathname] [-dt] [name ...]
```

현재까지 사용된 명령들의 위치와 사용횟수를 기억하고 관리할수 있습니다.

pwd

```
pwd [-LP]
```

현재 사용중인 디렉토리를 표시합니다. `$PWD` 값과 같으며 `-P` 옵션을 사용할경우 심볼릭 링크 디렉토리 구조를 따르지 않고 물리적 디렉토리 구조를 따라 표시합니다.

readonly

```
readonly [-aAf] [name[=value] ...] or readonly -p
```

변수나 함수를 변경할수 없게 합니다. 설정후 변경을 시도하면 오류메시지와 `1` 을 리턴합니다.

return

```
return [n]
```

function 또는 source 한 스트립트 에서 리턴합니다. `n` 값은 종료값으로 사용되어 `$?` 변수를 통해 얻을수 있습니다. return 명령을 사용하지 않을경우 마지막 실행명령의 종료값이 사용됩니다.

shift

```
shift [n]
```

[positional parameters](#) 메뉴 참조.

test

```
test [expr]
```

[Test](#) 메뉴 참조.

trap

```
trap [-lp] [[arg] signal_spec ...]
```

[trap](#) 메뉴 참조.

umask

```
umask [-p] [-S] [mode]
```

User file-creation mode mask 는 파일과 디렉토리를 생성할때 기본적으로 어떤 퍼미션으로 생성할지를 8진수값으로 설정합니다. mask 를 전혀주지 않는다면 (000) 기본적으로 파일은 666, 디렉토리는 777 로 생성됩니다. 마스크값을 022 로 준다면 파일은 666 에서 각자리수 별로 빼기계산을 하면 644 로 생성이 되고 디렉토리는 755 로 생성이 되게 됩니다. 좀더 자세한 내용은 [여기](#) 를 참고하세요.

unset

```
unset [-f] [-v] [-n] [name ...]
```

현재 설정돼있는 변수나 함수를 삭제하여 존재하지 않는 상태로 만듭니다. readonly 속성이 있을 경우 unset 되지 않습니다. AA=, AA="", AA="" 는 값은 null 이지만 현재 존재하는 (set 돼있는) 변수입니다. unset 을하면 존재하지 않는 상태가 됩니다. [-v name] 으로 테스트 해볼수 있습니다. 이때 name 에는 \$ 를 붙이지 않습니다.

Bash Builtins

[

```
[ arg... ]
```

`test` 명령과 동일한 명령입니다. [Test](#) 메뉴를 참조하세요.

alias

```
alias [-p] [name[=value] ... ]
```

alias 를 설정하는데 사용합니다. alias 는 기본적으로 non-interactive shell 에서는 사용할수 없습니다.

bg

```
bg [job_spec ...]
```

[job control](#) 메뉴 참조

bind

```
bind [-lpsvPSVX] [-m keymap] [-f filename] [-q name] [-u name] [-r keyseq] [-x  
keyseq:shell-command] [keyseq:readline-function or readline-command]
```

readline 의 설정 파일인 inputrc 에서 할수있는 옵션설정, key binding 을 이 명령을 통해서 동일하게 할 수 있습니다. 사용할 수 있는 함수나 옵션의 목록을 볼수도 있으며 현재 설정 상태도 확인할 수 있습니다.

builtin

```
builtin [shell-builtin [arg ...]]
```

우선순위가 높은 alias, function 이름을 피해 builtin 명령을 실행 할때 사용합니다.

caller

```
caller [expr]
```

현재 실행중인 함수의 call stack 을 표시합니다. 인수없이 사용하면 라인넘버, 파일이름 (\${BASH_LINENO[0]} \${BASH_SOURCE[1]}) 을 표시하고 숫자를 인수로 주면 라인넘버, 호출함수, 파일이름 (\${BASH_LINENO[\$i]} \${FUNCNAME[\$i+1]} \${BASH_SOURCE[\$i+1]}) 을 표시합니다. stack trace 에 사용할수 있습니다.

```
#!/bin/bash

die() {
    local frame=0
    while caller $frame; do
        ((frame++));
    done
    echo "$*"
    exit 1
}

f1() { die "*** an error occurred ***"; }
f2() { f1; }
f3() { f2; }

f3

##### output #####

12 f1 ./callertest.sh
13 f2 ./callertest.sh
14 f3 ./callertest.sh
16 main ./callertest.sh
*** an error occurred ***
```

command

```
command [-pVv] command [arg ...]
```

우선순위가 높은 alias, function 이름을 피해 외부명령을 (builtin 명령 포함) 실행 할때 사용합니다.

compgen

```
compgen [-abcdefgjkusv] [-o option] [-A action] [-G globpat] [-W wordlist] [-F function] [-C command] [-X filterpat] [-P prefix] [-S suffix] [word]
```

명령 자동완성에 사용될 단어들을 생성하는데 사용됩니다. 주로 자동완성 함수를 작성할때 COMPREPLY 변수에 값을 채워 넣기 위해 사용합니다. 사용 가능한 옵션은 complete 명령과 거의 동일하며 마지막에 word 인수가 주어지면 word 와 매칭되는 단어들만 선택됩니다.

complete

```
complete [-abcdefgjkusv] [-pr] [-DE] [-o option] [-A action] [-G globpat] [-W wordlist] [-F function] [-C command] [-X filterpat] [-P prefix] [-S suffix] [name ...]
```

특정 이름의 자동완성을 등록하고 설정하는 역할을 합니다.

compt

```
compt [-o|+o option] [-DE] [name ...]
```

complete 으로 등록되어 사용 중인 명령 자동완성 에대해서 옵션을 변경할 수 있습니다.

declare

```
declare [-aAfFgiInrtux] [-p] [name[=value] ...]
```

변수를 선언하고 속성을 적용합니다. 값을 대입하지 않으면 존재하지 않는 상태 (unset 상태) 로 남습니다. 변수, array 의 현재 속성과 값을 보거나 함수정의 볼때도 사용합니다. 유효하지 않은 옵션을 사용하거나 값을 대입할때 오류가 나면 종료상태 값으로 non-zero 를 리턴합니다.

- Associative array 를 만들때는 `declare -A` 명령을 사용해야 합니다.
- 함수 내에서 사용되면 기본적으로 `local` 변수로 설정됩니다.
-g 옵션으로 global 변수로 설정 할수있습니다.
- `declare -n` 을 이용하여 named reference 기능을 이용할수 있습니다.
- `declare -t -f` 함수명 을 사용하면 해당 함수에서 DEBUG, RETRUN trap 을 할수 있습니다.

Options:

| 옵션 | 설명 |
|-----------|--|
| -f | 함수 정의를 볼때 사용합니다. 특정 action 을 함수에 적용시킬때 사용합니다. |
| -F | 함수 이름만 표시합니다. (디버깅 시에는 line number 와 source file 도 표시합니다.) |
| -g | 함수 내에서 declare 는 기본적으로 local 변수를 만듭니다. 하지만 이 옵션을 사용하면 global 변수가 됩니다. |
| -p | NAME 의 속성과 값을 볼때 사용합니다. |

속성을 줄때 사용하는 옵션

- 대신에 + 를 사용하면 해당 속성이 off 됩니다.

| 옵션 | 설명 |
|-----------|---|
| -a | NAME 에 indexed arrays 속성을 줍니다. |
| -A | NAME 에 associative arrays 속성을 줍니다. |
| -i | NAME 에 integer 속성을 줍니다. 값을 대입할때 let 명령과 같이 산술연산을 할 수 있습니다. |
| -n | NAME 에 named reference 속성을 줍니다. |
| -r | NAME 에 readonly 속성을 줍니다. |
| -t | NAME 에 trace 속성을 줍니다. |
| -l | NAME 에 값을 대입할때 소문자로 변환합니다. |
| -u | NAME 에 값을 대입할때 대문자로 변환합니다. |
| -x | NAME 을 export 합니다. |

dirs

```
dirs [-clpv] [+N] [-N]
```

pushd, popd 명령과 함께 directory stack 을 다루는데 사용되는 명령으로 현재 stack 내용을 보여줍니다. 0 번은 항상 현재 디렉토리를 나타냅니다.

disown

```
disown [-h] [-ar] [jobspec ...]
```

[job control](#) 메뉴 참조

echo

```
echo [-neE] [arg ...]
```

기본적으로 [escape 문자](#)는 처리하지 않으며 라인 마지막에 newline 을 붙이는데 필요하지 않을경우 **-n** 옵션을 사용할수 있습니다. 인수를 quote (single, double quotes) 하고 **-e** 옵션을 사용하면 escape 문자가 처리됩니다.

enable

```
enable [-a] [-dnps] [-f filename] [name ...]
```

builtin 명령을 disable 또는 enable 시키는데 사용합니다. builtin 명령과 동일한 이름의 외부 명령을 사용하고자 할 때 이 명령으로 builtin 명령을 disable 시킬 수 있습니다.

false

```
false
```

항상 종료값 `1` 을 리턴하는 명령입니다.
명령이므로 실행이 되어 종료 값을 얻을 수 있습니다.

```
$ echo false
false
$ false
$ echo $?
1
```

fc

```
fc [-e ename] [-lnr] [first] [last]
```

`command history` 에서 지정한 범위의 명령들을 에디터로 불러와 수정하여 스크립트 처럼 한번에 실행시킬수 있습니다.

```
fc -s [pat=rep] [command]
```

`!` 을 이용한 `history` 확장처럼 명령을 검색하여 실행할수 있습니다.

```
$ printf "%s\n" "hello\tworld"
hello\tworld

$ fc -s %s=%b printf
printf "%b\n" "hello\tworld"
hello      world
```

fg

```
fg [job_spec]
```

[job control](#) 메뉴 참조

help

```
help [-dms] [pattern ...]
```

Shell builtin 명령의 도움말을 보여줍니다.

history

```
history [-c] [-d offset] [n] or history -anrw [filename] or history -ps arg [arg...]
```

[command history](#) 메뉴 참조.

jobs

```
jobs [-lnprs] [jobspec ...] or jobs -x command [args]
```

[job control](#) 메뉴 참조

kill

```
kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
```

[kill](#) 메뉴 참조.

let

```
let arg [arg ...]
```

산술연산 표현식을 위한 명령입니다. 식을 작성할때 기본적으로 공백없이 붙여야 합니다. 그러나 quote 를 할경우는 공백을 사용할 수 있습니다. 좀 더 자세한 내용은 [special expressions](#) 메뉴를 참조하세요

local

```
local [option] name[=value] ...
```

local 변수를 설정할때 사용합니다. 그러므로 함수 내에서만 사용할수 있습니다. `declare` 명령에서 사용하는 옵션들을 동일하게 사용할수 있습니다. 재미있는것은 local 로 설정한 변수를 child function 에서 읽고 쓸수 있다는것입니다. 그리고 변경한 내용도 parent function 에 적용이 됩니다.

logout

```
logout [n]
```

login shell 일 경우에 logout 합니다. login shell 이 아니면 오류가 발생합니다.

mapfile

```
mapfile [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]
```

라인을 array 변수로 읽어들이입니다. array 변수명을 주지 않으면 MAPFILE 이 사용됩니다. 몇번째 라인부터 얼마나 읽어들이지, 저장되는 array 에는 몇번째 index 부터 입력할지 정할수 있고 callback 명령을 사용할 수도 있습니다.

```
mapfile -t array < datafile    # 이명령은 실질적으로 아래와 같습니다.

while read -r line
do
    array[i]="$line"
    i=$((i + 1))
done < datafile
```

- `-t`
읽어들이는 라인에서 `newline` 을 제거합니다.
- `-s count`
몇번째 라인부터 읽어들이지 지정합니다.
- `-n count`
몇개의 라인을 읽어들이지 지정합니다. 0 이면 전체 라인을 읽어들이입니다.
- `-O origin`
array 변수에 몇번째 index 부터 입력할지 지정합니다.
- `-u fd`
file descriptor 로부터 라인을 읽어들이입니다.

입력되는 array 와 별개로 callback 명령을 사용할 수 있습니다.

- `-C callback`
callback 명령을 지정합니다.
- `-c quantum`
몇개의 라인을 읽어들이후 callback 을 호출할지 지정합니다.

```
$ cat datafile
100 Emma Thomas
200 Alex Jason
300 Madison Randy
400 Sanjay Gupta
500 Nisha Singh

# callback 명령의 $1 에는 index 값이, $2 에는 라인이 전달됩니다.
$ f1() { echo "$1 : $2" ;}

$ mapfile -t -C f1 -c 1 < datafile      # -c 1 설정
0 : 100 Emma Thomas
1 : 200 Alex Jason
2 : 300 Madison Randy
3 : 400 Sanjay Gupta
4 : 500 Nisha Singh

$ mapfile -t -C f1 -c 2 < datafile      # -c 2 설정
1 : 200 Alex Jason
3 : 400 Sanjay Gupta

$ mapfile -t -C f1 -c 3 < datafile      # -c 3 설정
2 : 300 Madison Randy
```

callback 에서 read 명령을 사용하면 array 에 입력되는 라인이 skip 됩니다.

```
$ f1() { read -r line ;}

$ mapfile -t -C f1 -c 1 arr < datafile

# 200, 400 라인이 skip 되었다.
$ echo "${arr[0]}"
100 Emma Thomas
$ echo "${arr[1]}"
300 Madison Randy
$ echo "${arr[2]}"
500 Nisha Singh
```

popd

```
popd [-n] [+N | -N]
```

pushd, dirs 명령과 함께 directory stack 을 다루는데 사용되는 명령으로 stack 에서 디렉토리 항목을 삭제합니다.

printf

```
printf [-v var] format [arguments]
```


[printf](#) 메뉴 참조

pushd

pushd

popd, dirs 명령과 함께 directory stack 을 다루는데 사용되는 명령으로 인수로 사용된 디렉토리를 stack 에 추가합니다.

read

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]
```

[read](#) 메뉴 참조.

readarray

```
readarray [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]
```

`mapfile` 명령과 동일합니다.

set

```
set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg ...]
```

Shell option 을 설정하거나 명령의 인수값을 나타내는 `$1` `$2` `$3` ... 변수를 설정하는데 사용합니다. enable 된 옵션은 `$SHELLOPTS` 변수에 `:` 로 분리되어 저장되며 `$-` 변수에는 옵션 flags 가 저장됩니다. 주의할점은 enable 시에는 `-` 기호를 disable 시에는 `+` 기호를 사용한다는 것입니다. `[-o 옵션명]` 으로 현재 설정값을 테스트해볼 수 있습니다.

shopt

```
shopt [-pqsu] [-o] [optname ...]
```

Bash option 을 설정하는데 사용합니다. enable 된 옵션은 `$BASHOPTS` 변수에 `:` 로 분리되어 저장됩니다. enable 시에는 `-s` 를 disable 시에는 `-u` 옵션을 사용하고 `-q` 옵션을 사용해 값의 설정 여부를 테스트할 수 있습니다.

```
$ shopt -s nullglob
$ shopt -q nullglob; echo $?
0

$ shopt -u nullglob
$ shopt -q nullglob; echo $?
1
```

source

```
source filename [arguments]
```

. (a period) 명령과 동일합니다.

bash 에서만 사용할수 있고 filename 을 찾을때 `$PATH` 에서 찾지 못하면 현재 디렉토리 에서도 찾습니다.

suspend

```
suspend [-f]
```

[job control](#) 메뉴 참조

true

```
true
```

항상 종료값 0 을 리턴하는 명령입니다.
명령이므로 실행이 돼야 종료 값을 얻을 수 있습니다.

```
$ echo true
true
$ true
$ echo $?
0
```

type

```
type [-afptP] name [name ...]
```

-a 옵션과 함께 사용할경우 alias, function, keyword, builtin, 외부 명령을 모두 구분해서 보여줍니다.

typeset

```
typeset [-aAfFgiIrtux] [-p] name[=value]
```

declare 명령과 같이 변수의 값이나 속성을 설정하는데 사용합니다. (Obsolete)

ulimit

```
ulimit [-SHabcdefilmnpqrstuvxT] [limit]
```

shell 의 리소스 제한을 설정합니다. **hard limit (-H)** 은 root 유저만 설정할수 있고 **soft limit (-S)** 은 **hard limit** 이내에서 일반유저가 설정할수 있습니다. 설정된 값은 해당 **shell** 과 생성되는 **child process** 에 적용됩니다. 웹서버나 데이터베이스 서버등을 운영할때 설정값이 너무 낮으면 열린 파일수가 너무 많거나 스레드를 생성할수 없다거나 하는 리소스관련 오류가 발생할 수 있습니다. 이때는 **startup script** 파일에서 **ulimit** 명령을 이용하여 관련된 리소스를 설정하거나 또는 시스템 설정 파일을 수정하여 **reboot** 후에도 유지되게 합니다.

현재 실행중인 프로세스의 설정 상태는 **pid** 를 구한후에 `cat /proc/{pid}/limits` 명령으로 알아볼 수 있습니다.

다음은 프로세스당 열수있는 최대 파일수를 설정하는 예입니다.

```
/etc/security/limits.conf
```

파일에 추가

```
* soft nfile 65536
```

```
* hard nfile 65536
```

```
root soft nfile 65536
```

```
root hard nfile 65536
```

```
-----
```

```
/etc/pam.d/common-session-noninteractive
```

```
/etc/pam.d/common-session
```

파일에 추가

```
session required pam_limits.so
```

unalias

```
unalias [-a] name [name ...]
```

설정된 **alias** 를 삭제 하는데 사용합니다.

wait

```
wait [-n] [id ...]
```

[job control](#) 메뉴 참조

read

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]
```

read 명령은 stdin 으로부터 라인을 읽어들이어서 IFS 값에 따라 단어를 분리한 다음 지정한 [name ...] 에 값을 할당합니다. awk 의 용어를 빌려보자면 라인은 record 에 해당하고 단어는 field 에 해당하며 -d delim 옵션으로 지정하는 값은 RS (record separator) , IFS 값은 FS (field separator) 와 같은 의미가 됩니다. name 값을 주지 않으면 읽어들이는 라인은 REPLY 변수에 할당됩니다.

[name ...] 에 값을 할당하는 방법

```
# name 이 하나면 단어 전체를 할당
$ read v1 <<< "1 2 3 4 5"
$ echo $v1
1 2 3 4 5

-----

# name 이 단어 개수보다 적을경우 마지막 name 에 나머지를 할당
$ read v1 v2 v3 <<< "1 2 3 4 5"
$ echo $v1
1
$ echo $v2
2
$ echo $v3
3 4 5

-----

$ read _ v2 _ v4 _ <<< "1 2 3 4 5"
$ echo $v2
2
$ echo $v4
4
```

read 명령은 IFS 값에 따라 단어를 분리 합니다.

```
$ IFS=':|@' read -r col1 col2 col3 col4 <<< "red:green|blue@white"
$ printf "col1: %s col2: %s col3: %s col4: %s\n" "$col1" "$col2" "$col3" "$col4"
col1: red col2: green col3: blue col4: white

-----

datetime="2008:07:04 00:34:45"
IFS=': ' read -r year month day hour minute second <<< "$datetime"
```

파이프에 연결된 명령은 subshell 에서 실행되므로 다음과 같이 할 수 없습니다.

```
$ echo 1 2 3 4 5 | read v1 v2 v3
$ echo $v1 $v2 $v3

-----

# 다음과 같이 명령 group 을 이용하면 값을 표시할 수 있습니다.
$ echo 1 2 3 4 5 | { read v1 v2 v3; echo $v1 $v2 $v3 ;}
1 2 3 4 5
```

IFS 가 데이터 마지막에 올 경우 마지막 필드는 항목에서 제외됩니다.

```
$ IFS=, read -a arr <<< "1,2,3,4,5"
$ echo ${#arr[@]}
5

$ IFS=, read -a arr <<< "1,2,3,4," # 마지막 항목은 array 원소에서 제외됩니다.
$ echo ${#arr[@]}
4
```

Options

- **-r** : 읽어 들이는 데이터에서 `\` 문자를 이용한 escape 을 disable 합니다 (raw read). 이 옵션은 항상 사용하는 것을 권장합니다.
- **-d delim** : 라인 구분자를 의미하며 기본적으로 newline 입니다.

```
# find 명령에서 -print0 을 이용해 출력했으므로 -d 값을 null 로 설정
# 파일이름이 단어분리가 되는것을 금지하기 위해 IFS 값도 null 로 설정
find . -print0 | while IFS= read -rd '' file; do
    echo "$file"
done
```

```
-----

# -d 값을 null 로 설정하면 파일 전체 라인을 읽어드립니다.
$ read -rd '' whole < datafile
```

```
$ echo "$whole"
20081010 1123 xxx
20081011 1234 def
20081012 0933 xyz
...
```

- **-a array** : 단어를 분리해서 array 에 입력합니다.

```
$ read -ra arr <<< "1 2 3 4 5"

$ echo ${#arr[@]}
5
$ echo ${arr[1]}
2
```

- **-p prompt** : 사용자에게 값을 입력받을때 prompt 를 설정할 수 있습니다.
- **-e** : 사용자에게 값을 입력받을때 readline 을 사용합니다.
- **-i text** : **-e** 옵션과 같이 사용하며, 초기 입력값을 설정할 수 있습니다.
- **-s** : 사용자에게 값을 입력받을때 타입 한 값을 화면에 표시하지 않습니다.

```
$ read -p "Enter the path to the file: " -ei "/usr/local/" reply
Enter the path to the file: /usr/local/bin

$ echo "$reply"
/usr/local/bin
```

- **-n nchars** : nchars 만큼 문자를 읽어 드립니다. 중간에 라인 구분자를 만나면 중단합니다.
- **-N nchars** : 라인 구분자를 상관하지 않고 무조건 nchars 만큼 읽어 드립니다.

```

$ read -n 8 v1 <<END
12345
6789
END

# 중간에 라인구분자 newline 을 만나므로 5 까지만 표시됩니다.
$ echo "$v1"
12345

-----

$ read -N 8 v1 <<END
12345
6789
END

# newline 이 포함되므로 7 까지 표시됩니다.
$ echo "$v1"
12345
67

-----

asksure() {
    echo -n "Are you sure (Y/N)? "
    while read -n 1 answer; do
        if [[ $answer = [YyNn] ]]; then
            [[ $answer = [Yy] ]] && return 0
            [[ $answer = [Nn] ]] && return 1
            break
        fi
    done
}

-----

pause() {
    read -s -n 1 -p "Press any key to continue..."
}

```

- **-t timeout** : 사용자에게 입력을 받을때 timeout 시간을 지정할 수 있습니다.
이것은 용도가 하나 더 있는데 timeout 값을 0 으로 설정하면 데이터를 읽어들이지 않고 바로 리턴합니다. 이때 종료상태 값은 읽어들이 데이터가 있을 경우는 0 을 그외는 128 이상의 값을 리턴합니다.
- **-u fd** : stdin 대신에 file descriptor 로 부터 데이터를 읽어 드립니다.


```
exec 3< infile

while read -r -u 3 line; do
    echo "$line"
done

exec 3>&-
```

종료 상태 값

다음의 경우는 오류에 해당하고 0 이 아닌값을 리턴합니다.

- end-of-file 상태를 만났을때
- read times out (이때는 128 이상의 값을 리턴합니다.)
- 변수에 값을 할당할때 오류발생
- `-u` 옵션에 사용된 유효하지 않은 FD (file descriptor)

여기서 첫번째 경우 end-of-file 상태를 만났을때를 살펴볼 필요가 있는데요. 보통 에디터에서 텍스트를 입력하면 매 라인마다 newline 을 붙입니다 (마지막 라인을 개행을 안하고 저장해도 newline 이 붙습니다). 그래서 read 명령으로 라인을 읽어 들일때 end-of-file 상태를 만나지 않는데요. 그런데 `-d` 옵션 값으로 null 을 사용하거나 다음과 같이 newline 이 포함되지 않을 경우는 라인을 읽어들이는 도중에 end-of-file 상태를 만나게 되어 에러를 리턴하게 됩니다.

```
$ echo -n hello > tmpfile

$ cat tmpfile | od -a          # 라인 마지막에 newline 이 없다.
00000000  h   e   l   l   o

$ read -r v1 < tmpfile

$ echo $?                      # end-of-file 상태를 만나게 되고 에러를 리턴한다.
1

$ echo "$v1"
hello
```

printf

```
printf [-v var] format [arguments]
```

printf에서는 single, double quotes 모두에서 escape 문자가 처리됩니다. 또한 quotes의 고유 기능은 그대로 유지 되므로 double quotes에서는 변수확장, 명령치환이 됩니다.

```
$ AA=world

# double quotes
$ printf "hello\t$AA\n"    # \t \n escape 문자가 처리되고 변수확장이 된다.
hello      world

# single quotes
$ printf 'hello\t$AA\n'    # \t \n escape 문자만 처리된다.
hello      $AA
```

16진수, 8진수로 문자를 출력할때는 printf "%c" ... 형식으로 하지 않습니다.

```
$ printf "hello \x41\x42\x43\n"    # 16진수
hello ABC

$ printf "hello \101\102\103\n"    # 8진수
hello ABC
```

`-v var` 옵션은 출력값을 변수에 저장합니다. (bash only)

```
$ printf -v IFS " \t\n"
```

Arguments

printf는 [format tags](#)와 그에 상응하는 인수를 이용하여 여러가지 형태로 출력할 수 있습니다. 이때 인수에 사용되는 숫자는 다음과 같은 형식을 사용할 수 있습니다.

- `N` : 10진수 (decimal) 숫자
- `0N` : 8진수 (octal) 숫자
- `0xN` : 16진수 (hexadecimal) 소문자 숫자
- `0XN` : 16진수 (hexadecimal) 대문자 숫자
- `'X'` : X는 a character
- `"X"` : X는 a character

```
# %d 는 인수 값을 10 진수로 표시

$ printf "%d\n" 10      # decimal
10
$ printf "%d\n" 010     # octal
8
$ printf "%d\n" 0x10     # hexadecimal
16
$ printf "%d\n" "'A'"    # a character
65
```

format tags 개수보다 인수의 개수가 많을 경우는 명령이 반복됩니다.

```
$ printf "< %d >" 11
< 11 >

$ printf "< %d >" 11 22 33    # %d 는 하나인데 인수는 3개 이므로 3번 반복
< 11 >< 22 >< 33 >

$ printf "< %d >\n" 11
< 11 >

$ printf "< %d >\n" 11 22 33
< 11 >
< 22 >
< 33 >
```

Format Tags

format 스트링에서 다음과 같이 format tag 을 구성하여 인수 값의 출력 형태를 변경할 수 있습니다.

```
%[flags][width][.precision]specifier
```

Specifier

- `%d`, `%i` : signed decimal number 로 표시합니다.
- `%u` : unsigned decimal number 로 표시합니다.
- `%o` : unsigned octal number 로 표시합니다.
- `%x` : unsigned hexadecimal number (소문자) 로 표시합니다.
- `%X` : `%x` 와 같으나 대문자로 표시합니다.

```
$ printf "%d, %d\n" 10 -10    # signed decimal
10, -10

$ printf "%u, %u\n" 10 -10    # unsigned decimal
10, 18446744073709551606

$ printf "%o, %o\n" 10 -10    # unsigned octal
12, 17777777777777777766

$ printf "%x, %x\n" 10 -10    # unsigned hexadecimal
a, ffffffffffffffff6
```

- `%f` : floating point number 로 표시합니다.
- `%e` : scientific notation 으로 표시합니다.
- `%E` : `%e` 와 같으나 대문자 E 를 사용합니다.
- `%g` : 값에 따라 floating point number 또는 scientific notation 을 사용합니다.
- `%G` : `%g` 와 같으나 scientific notation 에서 대문자 E 를 사용합니다.
- `%a` : C99 형식의 hexadecimal floating point number 로 표시합니다. (bash only)
- `%A` : `%a` 와 같으나 대문자로 표시합니다. (bash only)

```

$ printf "%e\n" 123          # scientific notation
1.230000e+02

$ printf "%f\n" 123          # floating point number
123.000000

$ printf "%g\n" 123
123

$ printf "%f\n" 123.4567
123.456700

$ printf "%g\n" 123.4567     # floating point number
123.457

$ printf "%f\n" 12345678.123
12345678.123000

$ printf "%g\n" 12345678.123 # scientific notation
1.23457e+07

$ printf "%f\n" 0.0000123
0.000012

$ printf "%g\n" 0.0000123    # scientific notation
1.23e-05

$ printf "%a\n" 123.4567     # hexadecimal floating point number
0xf.6e9d495182a9931p+3

```

- `%s` : 인수 값을 **escape 문자** 처리 없이 그대로 출력합니다.
- `%b` : 인수 값을 **escape 문자** 처리하여 출력합니다.

```

$ printf "%s\n" 'hello\tworld'
hello\tworld          # escape 문자 처리 없이 그대로 출력

$ printf "%b\n" 'hello\tworld'
hello    world        # escape 문자를 처리하여 출력

```

- `%q` : shell 에서 input 으로 사용할수 있게 quote 하여 출력합니다.

```
$ printf "%q\n" 'a test file'
a\ test\ file

-----

sshc() {
    remote=$1; shift
    ssh "$remote" "$(printf "%q " "$@")"
}

$ sshc user@server touch "a test file" "another file"
```

- `%%` : `%` 문자를 프린트할때 사용합니다.
- `%c` : 인수의 첫번째 문자를 프린트합니다.
- `%(FORMAT)T` : FORMAT 에따라 date-time 을 프린트합니다.

```
$ printf "foo %%%s%%\n" bar
foo %bar%

$ printf "%c %c\n" abc def
a d

$ printf "today is %(Y-%m-%d)T\n"
today is 2015-08-31
```

- `%n` : 앞에서 출력된 문자수를 인수로 주어진 변수에 대입합니다. (bash only)

```
$ printf "12345%n6789%n\n" num1 num2
123456789

$ echo $num1 $num2
5 9
```

Width

- `N` : field width 를 설정합니다.
- `*` : field width 값을 인수로 받을 수 있습니다.

```
$ printf "%d %d %d\n" 100 200 300
100 200 300
$ printf "%10d %10d %10d\n" 100 200 300           # field width 를 10 으로 설정
      100          200          300
$ printf "%*d %*d %*d\n" 10 100 15 200 20 300      # 10, 15, 20 은 각각 * 에 대응하는 값
      100          200          300
```

Flags

- `-` : field width 내에서 값을 left 정렬합니다. (default 는 right 정렬입니다.)

```
$ printf "%10d %10d %10d\n" 100 200 300
      100        200        300
$ printf "%-10d %-10d %-10d\n" 100 200 300
100          200          300
```

- `0` : field width 에 맞게 zero padding 합니다.

```
$ printf "%06d\n" 12 345 6789
000012
000345
006789
```

- `+` : 숫자에 `+` , `-` sign 기호를 붙여서 표시합니다.

```
$ printf "%+10d %+10d %+10d\n" 100 -200 +300
      +100        -200        +300
```

- `space` : `+` 를 사용하지 않을경우 sign 자리에 space 를 두어 정렬합니다.

```
$ printf "%d\n" 100 -200 +300
100
-200
300
$ printf "% d\n" 100 -200 +300
100
-200
300
```

- `'` : 1000 의 자리마다 `,` 를 넣어 표시합니다. (bash only)

```
$ printf "%'d\n" 123456789
123,456,789
```

- `#` : alternative format 을 사용할 수 있습니다.

`%#o` 은 값을 octal number 로 표시할때 앞에 `0` 을 붙입니다.

```
$ printf "%#o\n" 10
012
```

`%%x` , `%%X` 은 값을 hexadecimal number 로 표시할때 앞에 `0x` , `0X` 를 붙입니다.

```
$ printf "%%x %%X\n" 10 30
0xa 0X1E
```

`%%g` , `%%G` 은 precision 내에서 trailing zero 를 붙입니다.

```
$ printf "%g\n" 12.34
12.34
$ printf "%#g\n" 12.34
12.3400
```

Precision

. 을 이용하면 왼쪽에는 field width 를 오른쪽에는 precision 을 설정할 수 있습니다.
field width 크기는 precision 을 포함합니다.

- `.N` : precision 값을 설정합니다.
- `.*` : precision 값을 인수로 받을 수 있습니다.

```
$ printf "%f\n" 123.987654321
123.987654
$ printf "%.3f\n" 123.987654321 # precision 을 3 으로 설정
123.988 # 소수 넷째 자리에서 반올림이 된다
$ printf "%. *f\n" 3 123.987654321 # '*' 을 이용하여 precision 값을 인수로 받음
123.988
```

`%f` 와 `%g` 는 precision 값을 처리하는 방식이 다릅니다.

`%f` 는 소수점 이후의 개수를 나타내고 `%g` 는 전체 유효숫자 개수를 나타냅니다.

```
$ printf "%.5f\n" 123.12345678 # 소수점 이후 5개
123.12346

$ printf "%.5g\n" 123.12345678 # 전체 유효숫자 5개
123.12

$ printf "%.5f\n" 0.0012345678
0.00123

$ printf "%.5g\n" 0.0012345678
0.0012346
```

`%.s` or `%.0s` 는 해당 인수를 출력에서 제외하는 효과를 갖습니다.


```
$ printf "%5d%5d%.s%.0s%5d\n" 11 22 33 44 55
    11    22    33    44    55
```

예제)

0 ~ 127 까지 숫자를 decimal, octal, hexadecimal 로 출력하기

```
$ for ((x=0; x <= 127; x++)); do
> printf '%3d | %04o | 0x%02x\n' $x $x $x
> done
 0 | 0000 | 0x00
 1 | 0001 | 0x01
 2 | 0002 | 0x02
...
...
125 | 0175 | 0x7d
126 | 0176 | 0x7e
127 | 0177 | 0x7f
```

1 ~ 2 자리로 되어있는 hexadecimal number 를 2 자리로 맞추기

```
$ mac_addr="0:13:ce:7:7a:ad"

$ printf "%02x:%02x:%02x:%02x:%02x:%02x\n" 0x${mac_addr//:/ 0x}
00:13:ce:07:7a:ad
```

현재 라인 우측 끝에 메시지 출력하기

`tput cols` 은 현재 터미널 `columns` 수를 출력. `echo $COLUMNS` 과 동일

```
printf "%*s\n" $(tput cols) "hello world"
```

eval

```
eval [ arg ... ]
```

`eval` 명령의 기본적인 기능은 `shell` 명령 구문을 실행하는 것입니다. 다음과 같은 경우는 특별히 `eval` 명령이 필요하지 않습니다.

```
# 다음의 경우 echo foo bar 명령문이 실행됩니다.
$ $( echo "echo foo bar" )
foo bar
```

하지만 명령문을 작성할 때는 해당 명령에서 사용되는 스트링외에 `shell` 메타문자, `shell` 키워드 그리고 대입연산이 함께 사용됩니다. 이와 같은 스트링은 명령치환이 일어나기 전에 `shell` 에 의해 해석되므로 다음과 같이 명령치환을 이용해 실행할 수 없습니다.

```
$ $( echo "if test 1 -eq 1; then echo equal; fi" )
if: command not found

$ $( awk 'BEGIN { print "AA=100" }' )
AA=100: command not found
```

이럴때 `eval` 을 사용합니다.

```
$ eval "$( echo "if test 1 -eq 1; then echo equal; fi" )"
equal

$ eval "$( awk 'BEGIN { print "AA=100" }' )"
$ echo $AA
100
```

eval 명령의 실행단계

`eval` 명령은 주어진 인수들을 읽어 들인 후 실행합니다. 여기서 주목해야 될 점은 두 단계를 거쳐서 실행이 된다는 점입니다. 첫째. 읽어들이는 단계 , 둘째. 실행하는 단계. 그래서 결과적으로 인수부분에서 확장과 치환이 두번 일어나게 됩니다. 읽어들일때 한번 실행할때 한번.

```
$ AA=100 BB=200
```

```
# 인수 부분에서 확장, 치환이 일어나 $BB 가 200 이되고
# 마지막에 값을 표시하는데 불필요한 quote 을 삭제한다.
```

```
$ echo '$AA' $BB
$AA 200
```

```
# 1. eval 읽어 들이는 단계에서 위와 같이 확장, 치환이 되고 quote 이 삭제된다.
```

```
# echo $AA 200
```

```
# 2. 실행단계 에서도 확장, 치환이 일어나므로 $AA 는 100 이된다.
```

```
$ eval echo '$AA' $BB
100 200
```

```
# 1. single quote 이 escape 됐으므로 read 단계에서 남게된다.
```

```
# echo '$AA' 200
```

```
# 2. 실행단계 에서 quote 이 삭제되어 표시된다.
```

```
$ eval echo \"'$AA'\" $BB
$AA 200
```

Brace 확장에서는 range 값으로 변수를 사용하지 못하는데 (변수확장이 뒤에 일어나므로) 이와 같은 eval 명령의 특성을 이용하면 brace 확장에서도 변수를 사용할 수 있습니다.

```
$ a=1 b=5
```

```
# 1. eval 명령 read 단계에서 변수확장이 일어난다.
```

```
# echo {1..5}
```

```
# 2. 실행단계 에서 brace 확장이 된다.
```

```
$ eval echo {$a..$b}
1 2 3 4 5
```

명령에 선행하는 대입연산을 이용할 때도 eval 명령을 사용할 수 있습니다. 명령 앞에서 대입한 값은 명령이 실행된 후에야 사용할 수 있는데 다음의 경우는 echo 명령이 실행되기 전에 \$A, \$B, \$C 인수확장이 일어나서 기존의 값을 표시하게 됩니다.

```
$ A=1 B=2 C=3
```

```
$ A=4 B=5 C=6 echo $A $B $C
```

```
1 2 3
```

다음과 같이 eval 명령을 사용하면 해결할 수 있습니다.

```
$ A=1 B=2 C=3

# 1. eval 명령 read 단계에 quote 이 삭제된다
# echo $A $B $C
# 2. 실행단계 : 이미 eval 명령이 실행중에 있으므로 대입한 값이 사용된다.
$ A=4 B=5 C=6 eval 'echo $A $B $C'
4 5 6
$ echo $A $B $C      # 명령 종료 후에는 원래 값으로 복귀
1 2 3
```

스크립트 실행 중에 명령문을 만들어 실행하기

```
$ AA='find . -name'

$ BB='\"*.sh\"'

$ eval \"$AA $BB\"
args.sh
sub.sh
test.sh
...
...
```

eval 명령을 사용하다 보면 복잡해 보일 때가 있는데 이때는 읽기 부분은 eval 을 echo 로 바꾸어 실행해 봅니다. echo 명령을 실행해서 나온 결과가 eval 명령이 실행단계에서 실행하게 될 명령입니다. 그리고 그 명령이 실행돼서 나온 종료값이 eval 명령의 종료값이 됩니다.

```
$ eval "$(echo hello world | sed 's/hello/wc <<< /')"
1 1 6
$ echo $?
0

# 1. 읽기단계: eval 을 echo 로 변경하여 실행해봄.
$ echo "$(echo hello world | sed 's/hello/wc <<< /')"
wc <<< world      # 이 명령이 eval 이 실행단계에서 실행하게 될 명령임.

$ wc <<< world    # 2. 실행단계
1 1 6            # 이 값이 eval 명령의 stdout 값이 되고
$ echo $?
0                # 이 값이 eval 명령의 종료 상태 값이 됩니다.
```

간접적 변수값 대입에 활용

다음과 같이 간접적으로 변수값을 대입할때 eval 명령을 활용할 수 있습니다.

```
$ AA=BB

$ echo $AA
BB

$ eval $AA=100

$ echo $BB
100
```

함수를 만들어 실행

```
$ main() {
>     local fbody='() { echo "function name is : $FUNCNAME"; }'
>     local fname
>     for fname in f{1..10}; do
>         eval "${fname}${fbody}"    # 이 부분에서 함수 정의가 됨
>         $fname                    # 함수명으로 실행
>     done
> }

$ main
function name is : f1
function name is : f2
function name is : f3
...
...
```

Keyword Commands

time

`time [-p] pipeline`

명령을 실행하는데 걸린 시간을 표시해줍니다. 명령들이 파이프로 연결되거나 `{ }`, `()` 로 group 되어있을 경우 모든 명령들을 포함합니다. TIMEFORMAT 환경변수를 이용해 출력 포맷을 변경할수 있습니다.

- real

wall clock 시간이라고 하며 프로그램이 시작된 후부터 종료될 때까지를 시계로 잰것과 같습니다. 그러므로 프로그램 실행시 단순히 I/O 를 위해 wait 한 시간이나 sleep 한 시간도 모두 포함됩니다.

```
$ time { sleep 1; sleep 2 ;}
```

```
real    0m3.002s
user    0m0.000s
sys     0m0.000s
```

- user

user 모드에서 사용한 cpu 시간입니다. wait 한 시간은 포함되지 않습니다. `user + sys` 시간이 실질적으로 프로그램이 사용한 cpu 시간이라고 볼 수 있습니다.

- sys

kernel 모드에서 사용한 cpu 시간입니다. 메모리를 할당하거나, 디스크 같은 장치에 접근하는 것은 system call 을 통해서 kernel 모드에서 실행됩니다.

real ≠ user + sys

요즘은 대부분 multi-core cpu 를 사용합니다. 그래서 가령 2 core cpu 를 사용 중이라고 가정했을 때, 프로그램이 실행후 종료될때까지 1분이 걸렸고, 각 core 에서 1분의 cpu 시간을 사용했다면 `real` 은 1분이지만 `user + sys` 는 2 분이 될수있습니다.

```
real    1m47.363s
user    2m41.318s
sys     0m4.013s
```

Exit Status:

종료상태 값은 pipeline 의 리턴값이 됩니다.

coproc

```
coproc [NAME] command [redirections]
```

coproc (coprocess) 는 두 프로세스 간에 양방향 통신을 가능하게 합니다. coproc 는 command 를 background 로 실행시키는데 이때 stdin, stdout 을 파이프를 통해 FD 에 연결해서 외부로부터 데이터 입력을 받고 연산결과를 출력할 수 있습니다. 설정된 FD 는 `$COPROC` array 변수에 원소로 등록되므로 프로세스로 데이터를 보내기 위해서는 `>& ${COPROC[1]}` , 받기 위해서는 `<& ${COPROC[0]}` 을 이용할 수 있습니다. background pid 는 `$COPROC_PID` 변수에 저장됩니다.

```
# background 로 실행됨
$ coproc while read -r line; do eval expr "$line"; done
[1] 18008

$ echo "1 + 2" >& ${COPROC[1]}

$ read -r res <& ${COPROC[0]}

$ echo $res
3

$ kill $COPROC_PID
```

위의 명령은 다음과 동일하다고 볼 수 있습니다. 그러니까 coproc 는 파이프를 생성해서 FD 를 연결하고 삭제하는 작업을 자동으로 해준다고 보면 되겠습니다.

```
$ mkfifo inpipe outpipe
$ exec 3<>inpipe 4<>outpipe

$ while read -r line; do eval expr "$line" > inpipe ; done < outpipe &
[1] 17647

$ echo "1 + 2" >& 4

$ read -r res <& 3

$ echo $res
3

$ kill 17647

$ exec 3>&- 4>&-
$ rm -f inpipe outpipe
```

다음은 coproc 를 실행했을때 FD 상태입니다.

```
1E$ declare -p COPROC
declare -a COPROC='([0]="63" [1]="60")'

1E$ ls -al /proc/$$/fd
total 0
dr-x----- 2 mug896 mug896 0 08.15.2015 00:56 ./
dr-xr-xr-x 9 mug896 mug896 0 08.15.2015 00:56 ../
lrwx----- 1 mug896 mug896 64 08.15.2015 00:56 0 -> /dev/pts/1
lrwx----- 1 mug896 mug896 64 08.15.2015 00:56 1 -> /dev/pts/1
lrwx----- 1 mug896 mug896 64 08.15.2015 00:56 2 -> /dev/pts/1
lrwx----- 1 mug896 mug896 64 08.15.2015 12:27 255 -> /dev/pts/1
l-wx----- 1 mug896 mug896 64 08.15.2015 12:27 60 -> pipe:[4344774]
lr-x----- 1 mug896 mug896 64 08.15.2015 12:27 63 -> pipe:[4344773]
```


Script 의 구성

Shell script 의 기본적인 기능은 프롬프트 상에서 실행시키던 명령들을 하나의 파일에 넣어 놓고 한번에 실행시키는 것인데요. 명령들을 구성할수 있는 방법을 분류해 보면 다음과 같습니다.

1. 한줄에 하나씩

```
command1
command2
command3
...
```

2. ; 메타문자를 이용하면 한줄에 여러 명령을 놓을 수 있다

```
command1; command2; command3 ...
```

여기서 ; 메타문자는 newline 과 같은 역할을 합니다. 1, 2 번은 각 명령들이 순서대로 실행이 됩니다. command1 이 종료돼야 그다음에 command2 가 실행되고 command2 가 실행을 종료해야 다음에 command3 이 실행됩니다.

3. 파이프를 이용해 여러명령을 동시에 실행

```
command1 | command2 | command3
```

파이프에 연결된 명령들은 동시에 실행 됩니다. command1 의 stdout 출력이 command2 의 stdin 입력으로 들어가고 command2 의 stdout 출력이 command3 의 stdin 입력으로 들어가고 최종적으로 command3 의 stdout 출력이 터미널로 표시됩니다.

4. &&, || 메타문자를 이용한 간단한 조건부 실행

```
command1 && command2
```

&& 메타문자는 command1 의 실행이 정상 종료하면 command2 를 실행하고 command1 이 오류로 종료할 경우는 command2 를 실행하지 않습니다.

```
command1 || command2
```

|| 메타문자는 command1 이 오류로 종료할 경우 command2 를 실행합니다. command1 이 정상 종료하면 command2 는 실행되지 않습니다.

```
A && {
  B && {
    echo "A and B both passed"
  } || {
    echo "A passed, B failed"
  }
} || echo "A failed"
```

5. { ... ; } , (...) 을 이용한 명령 grouping

여러 명령들을 하나의 group 으로 만들어 사용할 수 있습니다. 대표적인게 { ; } 을 이용한 function 이죠. 이렇게 group 을 만들면 안에 있는 명령들이 실행시에 같은 context 를 가집니다.

6 . Shell keyword 를 이용한 복합 명령 구성

위에서 알아본 1, 2, 3, 4, 5 번 방법만 가지고는 그다지 실용적인 스크립트를 구성할 수 없습니다. 그래서 shell 에서는 프로그래밍 언어에서처럼 조건분기 및 반복기능을 구성할수 있게 keyword 를 제공합니다.

Compound Commands

`compgen -k | column` 명령으로 볼 수 있는 대부분의 shell 키워드들이 복합 명령 구성을 하는데 사용됩니다. 이 키워드를 이용하면 shell 에서도 if else 문과 for, while 문을 만들 수 있습니다. 키워드는 명령 이름이 위치하는 곳에서 사용되며 인수 부분에서 사용될 경우 키워드로 기능하지 않습니다. 이 키워드들은 사용방법에 있어 다음과 같은 특징이 있습니다.

키워드로 시작해서 키워드로 끝난다.

테이블을 보면 loop 문은 전부 done 키워드로 끝나는걸 알 수 있습니다.

| 구분 | 구성 |
|----------|-----------------|
| if 문 | if ... fi |
| case 문 | case ... esac |
| select 문 | select ... done |
| while 문 | while ... done |
| until 문 | until ... done |
| for 문 | for ... done |

열고 닫는 키워드에 redirection, |, & 를 붙여 사용할 수 있다.

열고 닫는 키워드에 redirection, 파이프, & 를 붙이면 안에 있는 전체 명령에 적용됩니다.

```
if ;; then
    read line
    echo "$line"
else
    cat
fi < infile > outfile

-----

var=1
case $var in
    1)
        read line
        echo "$line"
        ;;
    *)
        cat ;;
esac < infile > outfile

-----

for (( i=10; i<20; i++ )); do
    read line
    echo "$line"
done < infile > outfile

-----

while read -r line; do
    echo "${line//banana/banana}"
done < infile > outfile

-----

if ! ;; then
    cat
else
    case $var in
        *)
            cat ;;
    esac < dbfile2
    cat
fi < dbfile1 > outfile
```

Conditional Constructs

if

```

if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi

```

- if 문은 `if`, `elif`, `else`, `fi` 그리고 `then` 키워드를 사용합니다.
- 키워드 다음에 명령이 오는 순서입니다. if 다음에 명령; then 다음에 명령; ... 마지막엔 fi 로 닫습니다.
- if 나 elif 다음에 오는 명령은 `[]`, `[[]]` 을 이용하는 것 외에도 어떤 명령이나 `group` 도 사용할 수 있습니다.
- `!` 키워드를 이용하면 logical NOT 연산을 할 수 있습니다.

```

if ! mount /mnt/backup &> /dev/null; then
    echo "FATAL: backup mount failed" >&2
    exit 1
fi

-----

if grep -q '2014-07-20' dbfile; then
    ...
fi

```

case

```

case word in [ ( [ pattern [ | pattern]...) command-list ;;)... esac

```

- case 문은 `case`, `in`, `esac` 키워드와 각 case 를 나타내는 `pattern)` 을 사용합니다.
- `pattern)` 의 종료 문자는 `;;` 를 사용합니다.
- `pattern)` 에서는 `|` 을 구분자로 하여 여러개의 패턴을 사용할 수 있습니다.
- 패턴에 공백을 포함하려면 `foo\ bar` 와같이 escape 합니다.
- `*`) 는 모든 매칭을 뜻하므로 default 값으로 사용할 수 있습니다.
- case 문의 종료값은 매칭되는 경우가 없을때는 0 을, 그외는 해당 case 의 마지막 명령의 종료값이 됩니다.
- `shopt -s nocasematch` 옵션을 사용하면 대, 소문자 구분없이 매칭할 수 있습니다.

```
read -p "Enter the name of an animal: " ANIMAL
echo -n "The $ANIMAL has "
case $ANIMAL in
    horse | dog | cat)
        echo -n 4
        ;;
    man | kangaroo)
        echo -n 2
        ;;
    *)
        echo -n "an unknown number of"
        ;;
esac
echo " legs."
```

select

select name [in words ...]; do commands; done

- select 문은 select , in , do , done 키워드를 사용합니다.
- PS3 변수는 프롬프트를 설정하는데 사용됩니다.
- REPLY 변수에는 입력한 값이 저장됩니다.
- in words 부분을 생략하면 in "\$@" 와 같게됩니다.
- 값을 입력하지 않고 enter 를 하면 다시 목록을 표시합니다.
- Ctrl-d 는 select loop 를 종료하고 다음 명령으로 진행합니다.
- select 는 반복해서 입력을 받으므로 선택을 완료하고 다음으로 진행하기 위해서는 break 명령으로 종료해야 합니다.

```

----- 공통부분 -----

AA=( "horse" "dog" "cat" "man" "kangaroo" )

PS3="Enter number: "

----- test1.sh -----

select ANIMAL in "${AA[@]}"
do
    echo "You picked number $REPLY : $ANIMAL"
done

----- test2.sh -----

AA+=( exit )

select ANIMAL in "${AA[@]}"
do
    case $ANIMAL in
        horse | dog | cat)
            number_of_legs=4
            ;;
        man | kangaroo)
            number_of_legs=2
            ;;
        exit)
            exit
            ;;
        *)
            echo Not available
            break
            ;;
    esac
    echo "You picked number $REPLY"
    echo "The $ANIMAL has $number_of_legs legs."
done

```

Looping Constructs

while

```
while test-commands; do consequent-commands; done
```

- while 문은 `while` , `do` , `done` 키워드를 사용합니다.
- 테스트되는 명령이 정상 종료하면 계속해서 반복합니다.
- 종료값은 마지막에 실행된 명령의 종료값이 됩니다.

`while` 과 `read` 명령을 이용해 파일을 읽어들이때 아래와 같이 파일을 `read` 명령에 연결하면 안됩니다. 왜냐하면 `read -r line < infile` 명령이 실행되고 나면 `infile` 과 연결이 `close` 되기 때문입니다. 그래서 다음 반복때 실행되면 다시 파일을 `open` 해서 읽어 들이므로 계속해서 첫라인만 표시되게 됩니다.

```
while read -r line < infile
do
    echo "$line"
done

##### output #####

Fred:apples:20:June:4
Fred:apples:20:June:4
Fred:apples:20:June:4
...
...
```

그러므로 파일을 `read` 명령이 아니라 `while` 이나 `done` 키워드에 연결 시켜야 합니다.

```
cat infile | while read -r line
do
    echo "$line"
done

##### or #####

while read -r line
do
    echo "$line"
done < infile
```

until

`until test-commands; do consequent-commands; done`

- `until` 문은 `until` , `do` , `done` 키워드를 사용합니다.
- 테스트되는 명령이 오류로 종료하면 계속해서 반복합니다.
- 종료값은 마지막에 실행된 명령의 종료값이 됩니다.

```
# until ...; do ...; done 은
# while ! ...; do ...; done 와 같습니다.

read -p "Enter Hostname: " hostname
until ping -c 1 "$hostname"
do
    sleep 60;
done
wget "$hostname/mydata.txt"
```

for

```
for name [ [in [words ...]] ; ] do commands; done
```

- for 문은 `for` , `in` , `do` , `done` 키워드를 사용합니다.
- `words` 는 IFS 값에 따라 분리되며, `words` 개수만큼 반복하게 됩니다. 매 반복때마다 `name` 값이 설정됩니다.
- `in words` 부분을 생략하면 `in "$@"` 와 같게됩니다.

```
$ set -f; IFS=$'\n'

$ for file in $(find .)
do
    echo "$file"
done
.
./WriteObject.java
./WriteObject.class
./ReadObject.java
./2013-03-19 154412.csv
./ReadObject.class
./셸 스크립트 테스트.txt

$ set +f; IFS=$' \t\n'
```

```
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

- `(())` 는 shell 에서 제공하는 산술연산 특수 표현식 입니다. 프로그래밍 언어에서 for 문처럼 사용할 수 있습니다.


```
$ for (( i = 0; i <= 5; i++ )) {  
    echo $i;  
}
```

```
0  
1  
2  
3  
4  
5
```

Special Expressions

Shell 은 기본적으로 `command arg1 arg2 ...` 형식의 명령을 다루지만 그 외에 shell 자체에서 제공하는 표현식이 있습니다. 이것은 shell 에의해서 해석되고 실행되기 때문에 사용방법에 있어서 일반 명령들과 달리 연산자를 escape 해야 한다든지 변수를 quote 해서 사용해야 하는 제약이 없고 여러가지 편리한 기능들을 제공합니다. `$(())` , `(())` , `let` 은 산술연산에 특화된 기능을, `[]` 은 `[` 명령과 같이 테스트에 특화된 기능을 제공합니다.

`$((...))` , `((...))`

먼저 이 표현식은 산술연산을 위한 표현식입니다. 스트링이나 문자는 다루지 않고 오직 숫자만 다룹니다. 그래서 표현식 안에 알파벳으로 된 단어가 오면 그건 변수명이라는 것 외에 다른 의미가 없습니다. 그래서 이 표현식 안에서 변수를 사용할때 보통 `$` 문자를 사용하지 않아도 됩니다. `$var`, `var` 모두 같은 의미입니다. 그리고 주의할 점은 quote 을 사용하면 안되고 (숫자만 다루기 때문에 quote 을 할 필요가 없음) 변수값이 숫자가 아니거나 존재하지 않는 변수를 사용할 경우는 0 과 같습니다.

```
$ A=10
$ (( A = $A + A ))           # 변수 A 에 $ 를 붙이지 않아도 된다.
$ echo $A
20

$ A=(5 6 7)
$ (( A[0] = A[1] + ${#A[@]} )) # { } 을 이용한 매개변수 확장은 $ 를 붙여야한다.
$ echo ${A[0]}
9
```

`$(())` `(())` 은 생긴건 소괄호로 생겼지만 subshell 이 아닌 현재 shell 에서 실행됩니다. 그래서 표현식 내에서 외부 변수값을 수정하여 적용시킬 수 있습니다.

이 표현식에서 사용할 수 있는 연산자는 C 언어에서 사용하는 연산자들과 동일합니다. `var++` , `-var` , `a ? b : c` , `()` 을 이용한 연산자 우선순위 조절, `,` 를 이용한 표현식 list 등 모두 사용할 수 있습니다.

정리하자면 `$(())` `(())` 표현식 내에서는 그냥 프로그래밍 언어를 한다고 생각하시면 됩니다. 참, 거짓 판단도 동일하게 0 이 거짓이고 그외 숫자가 참입니다. (산술연산 이기 때문에)

```

$ (( 1 < 2 )); echo $?      # (( )) 안에서 참으로 종료됐으므로 0
0

$ (( 1 > 2 )); echo $?      # (( )) 안에서 거짓으로 종료됐으므로 1
1

$ (( 0 )); echo $?         # 산술연산에서 0 은 거짓 이므로 종료값은 1
1

$ (( 1 )); echo $?         # 산술연산에서 0 이외의 값은 참 이므로 종료값은 0
0

$ (( var = 0, res = (var ? 3 : 4) ))
$ echo $res
4

$ (( var = 1, res = (var ? 3 : 4) ))
$ echo $res
3

$ AA=10
$ while (( AA )); do echo $AA; (( AA-- )); done

```

for ((i = 0; i < 10; i++)) 를 할수있다!

(()) 는 표현식 이라 ; 를 이용해서 여러 명령을 사용할 수는 없는데요. 그래서 ((i = 0; i < 10; i++)) 만 보자면 오류입니다. 하지만 앞에 for 가 올경우 C 언어 에서처럼 for 문으로 사용할 수 있습니다.

```

for (( i = 0; i < 10; i++ ))
do
    echo $i
done

##### or #####

for (( i = 0; i < 10; i++ )) {
    echo $i
}

```

let

\$(()) (()) 표현식은 사용하기에는 편리하지만 비교적 간단한 식을 작성할 때에는 이중괄호 때문에 좀 장황해지는 단점이 있습니다. 그래서 나온 것이 let 명령 입니다.

```
(( i++ ))          res=$(( var + 1 ))

let i++            let res=var+1
```

let 명령은 (()) 와 같이 안에있는 식을 구분해줄수 있는 괄호가 없기 때문에 하나의 인수만을 갖습니다. 그러므로 식을 쓸때는 기본적으로 공백 없이 붙여 써야 합니다. 하지만 quote 을 이용하면 공백을 사용할 수 있습니다.

```
$ let var = 1 + 2
bash: let: =: syntax error: operand expected (error token is "=")
$ let var += 1
bash: let: +=: syntax error: operand expected (error token is "+=")

$ var=4
$ let "var++, res = (var == 5 ? 10 : 20)"; echo $res
10
$ let "2 < 1"; echo $?
1
```

help let 하면 사용할 수 있는 연산자들을 한눈에 볼 수 있습니다.

[[...]]

이건 생긴 모양에서 알수있듯이 [명령의 확장 버전입니다. 그래서 [명령에서 사용할 수 있는 것은 동일하게 사용할 수 있습니다. 그리고 더해서 스트링의 pattern 매칭과 regex 매칭 기능도 제공합니다.

[명령에서 스트링 연산자를 사용하여 두값을 비교할 때는 스트링 대 스트링 비교입니다. 하지만 [[]] 표현식에서는 스트링 대 스트링 (=), 스트링 대 pattern (=), 스트링 대 regex (=~) 이 가능합니다. 그래서 오른쪽에 pattern 이나 regex 이 올때는 스트링과 구분하기 위해서 quote 을 해주면 안됩니다. 왜냐하면 pattern 이나 regex 을 quote 하면 스트링 대 스트링 비교가 되기 때문입니다.

```
##### pattern matching #####

[[ "hello world" = *llo\ wor* ]]; echo $?      # pattern 매칭
0

$ [[ "hello world" = "*llo wor*" ]]; echo $?    # 스트링 매칭
1

##### regex matching #####

$ [[ "hello world" =~ .*llo\ wor.* ]]; echo $?  # regex 매칭
0

$ [[ "hello world" =~ ".*llo wor.*" ]]; echo $? # 스트링 매칭
1

# regex 의 스트링 매칭은 부분만 매칭이 돼도 참이 됩니다.
$ [[ "hello world" =~ "wor" ]]; echo $?
0
```

표현식 내에서 `&&` , `||` 연산자를 제공합니다.

shell 메타문자가 아니므로 우선순위는 일반 프로그래밍 언어와 같습니다.

```
if [[ test1 && test2 ]]; then ...

if [[ test1 || test2 ]]; then ...

if [[ test1 && test2 || test3 ]]; then ...
```

사용예)

사실 복합 명령 구성을 할수있는 shell keyword 와 `[[]]` , `BASH_REMATCH`, `IFS` 변수, `read` 명령만 있으면 regex 관련해서는 거의 shell 에서 해결할 수 있습니다.

```
[[ ! $COLOR =~ BLUE|RED|GREEN ]] && {
    echo "Incorrect options provided"
    exit 1
}

-----
# 다음은 스트링에서 단어를 분리해 내는 과정입니다.

raw_string='dumpvmcore [--filename=name] info [args...]
injec-tnmi log [--release] | [--debug]] x [group-settings...]
logdest --release | debug-- [destinations...] log-flags
[--release] | [--debug]] [flags...] osdetect os osdmesg
[--lines=lines] | pattern'

for word in $( echo -n "$raw_string" | tr -d '|' ); do
    [[ $word =~ ^[:alnum:]-]+{3,}$ ]] \
    && [[ ! $word =~ ^-|-$ ]] \
    && echo "$word"
done | sort -u
```

[], [[]] 의 피연산자로 변수를 사용할 경우 quote 여부

O : 해야 한다 ("\$var") , **X** : 할 필요 없다 (\$var)

| | 산술연산 | 스트링연산 | 기타 unary 연산 |
|-------|------|---|-------------|
| [] | X | O | O |
| [[]] | X | X, O 우측 피연산자를 패턴이나 regex 이 아닌 스트링으로 다루고자 할때는 quote 한다 | X |

Shell Metacharacters

Shell script에서는 프로그래밍 언어에서처럼 여러 연산자를 제공하지 않습니다. 가령 산술연산을 위해서는 별도로 제공되는 **특수 표현식**을 사용하거나 외부 명령을 사용합니다. 하지만 명령문을 작성할때 사용되는 메타문자들이 있습니다. 이 메타문자들은 명령문 상에서 특별한 기능을 하므로 동일한 문자가 명령문의 스트링에 포함될 경우 반드시 **escape** 하거나 **quote** 해야 오류가 발생하지 않습니다.

```
( ) ` && || & | ;
< > >>           # redirection 문자
* ? [ ]           # glob 문자
" '               # quote 문자
\ $ { }           # 대입연산
= +=
```

()

- subshell 을 생성할 때
- \$() 명령 치환에 사용
- 명령 grouping 에 사용
- 우선순위 조절에도 사용

프로그래밍 언어에서처럼 공백이나 ; 에 대한 제약 없이 자유롭게 사용할 수 있는 메타문자입니다.

`

backtick 문자는 \$() 와 함께 명령 치환에 사용됩니다.

&& ||

AND, OR 논리 메타문자.

이것은 shell 메타문자로 [[]] 특수 표현식에서 제공하는 연산자와는 다른 것입니다. shell 에서 && || 메타문자는 우선순위를 같게 취급합니다.

&

명령을 `background` 로 실행할때 사용합니다.

또한 `;` 와 동일하게 명령문의 종료를 나타내므로 연이어 명령을 사용할때 `;` 를 붙여서는 안됩니다.

|

명령들을 파이프로 연결할때 사용합니다.

`|&` 는 `stdout`, `stderr` 둘 다 파이프로 전달하며 `2>&1 |` 와 동일한 의미입니다.

< > >>

이외에도 `redirection` 에 관련된 여러 메타문자들

`>>` (`append`), `>&`, `<>`, `&>` (`>word 2>&1`), `&>>` (`>>word 2>&1`), `>|` (`override noclobber`), `<<` (`here document`), `<<-` (`no leading tab`), `<<<` (`here string`)

;

한줄에 여러 명령을 연이어 사용할때 분리를 위해 사용합니다.

`;;` 는 `case` 문에서 각 `pattern` 의 종료를 나타내는데 사용됩니다.

* ? []

패턴매칭에 사용되는 `glob` 문자도 메타문자에 속합니다.

" ' `

문장을 `quote` 할때 사용합니다.

`space` 로 분리된 문장을 `quote` 하면 하나의 인수가 됩니다.

\

`escape` 할때 사용되는 문자입니다.

\$

매개변수 확장, 산술 확장, 명령 치환에 사용됩니다.

{ }

이것은 `$` 문자와 함께 매개변수 확장에 사용됩니다.

또한 `shell` 키워드로도 사용됩니다.

= +=

대입연산에 사용됩니다. `+=` 는 `sh` 에서는 사용할 수 없습니다.

메타문자가 명령문의 인수에 포함되면 **escape** 해야한다.

& 메타문자가 들어간 파일 이름을 인수로 사용해 오류 발생

```
# background process 가 실행되면서 오류발생
```

```
$ find . -name foo&bar.txt
```

```
[1] 16962
```

```
bar.txt: command not found
```

```
[1]+ Done
```

```
# 다음과 같이 수정합니다.
```

```
$ find . -name foo\&bar.txt
```

```
$ find . -name foo"&"bar.txt
```

```
$ find . -name foo'&'bar.txt
```

```
$ find . -name "foo&bar.txt"
```

```
$ find . -name 'foo&bar.txt'
```

`find` 명령은 인수로 `shell` 메타문자인 `()`, `,`, `;` 을 사용하는데 `quote` 하거나 `escape` 하지 않으면 정상적으로 실행이 되지 않습니다.

```
# '(' ')' 는 shell 메타문자 이므로 명령문에 바로 사용할시 오류발생

$ find . -type f ( -name "*.log" -or -name "*.bak" ) -exec rm -f {} ;
bash: syntax error near unexpected token '('

# '(' ')' 는 quote 했으나 ';' 메타문자로 인해 오류발생

$ find . -type f '(' -name "*.log" -or -name "*.bak" ')' -exec rm -f {} ;
find: missing argument to '-exec'

# 다음과 같이 shell 메타문자 들을 모두 escape 합니다.

$ find . -type f '(' -name "*.log" -or -name "*.bak" ')' -exec rm -f {} ';'
no error !

$ find . -type f \( -name "*.log" -or -name "*.bak" \) -exec rm -f {} \;
no error !
```

메타문자는 shell 에서 특별히 취급하는 문자다.

그러므로 다른 단어와 공백 없이 붙여서 사용할 수도 있다.

```
# 메타문자 일경우 '(' ')'
$ (true)&&(true;false); echo $?
$ 1

# 메타문자가 아닐경우 '{ }'
$ (true)&&{true;false}; echo $?
bash: syntax error near unexpected token `}'
```

프롬프트 상에서 주의해야 되는 keyword

! 은 메타문자는 아니고 프롬프트 상에서 history 확장에 사용되는 키워드로 사용에 주의해야 합니다.

(non-interactive shell 인 script 실행시에는 history 기능이 disable 됩니다).

! 문자 뒤에 공백이 오면 logical NOT 으로 그렇지 않으면 history 확장으로 해석됩니다.

```

$ !comp                                # history 확장
compgen -k | column

$ echo "hello!516world"                # double quote 에서도 history 확장이 된다!
hellocompgen -k | columnworld

$ echo 'hello!516world'                 # single quote 에서는 확장이 안된다.
hello!516world

# double quote 사용시 다음과 같이 history 확장을 회피할수 있다
$ echo "hello"\!"516world"             # 또는 "hello"'\!'516world"
hello!516world

### '!=' 는 history 확장이 안된다.

$ [ aaa != bbb ]; echo $?
0

```

! 뒤에 공백이 오면 logical NOT 으로 사용됩니다.

```

$ true && ! false ; echo $?
0

```

명령의 인수로 사용될 수 있다

```

$ find . ! -name "*.o"

$ if test ! 2 -eq 3; then echo true; fi
true

```

대입 메타문자로 += 를 사용할 수 있다.

`+=` 대입 메타문자는 변수 와 array 에서 모두 사용할 수 있습니다. (sh 에서는 사용할 수 없습니다.)

```
##### variable #####

$ AA=hello

$ AA+=" world"

$ echo "$AA"
hello world

##### indexed array #####

$ ARR=(11 22 33)

$ ARR+=(44)

$ echo ${ARR[@]}
11 22 33 44

##### associative array #####

$ ARR=([i1]=aaa [i2]=bbb [i3]=ccc)

$ ARR+=([i4]=ddd)

$ echo ${ARR[@]}
ccc bbb aaa ddd

$ echo ${!ARR[@]}
i3 i2 i1 i4
```

Shell 에는 ' , ' 분리자가 없다

기본적으로 명령의 인수나 array 의 원소를 구분하기 위해 공백과 IFS 값을 사용합니다.

```
f1() {
    echo arg1 : $1
    echo arg2 : $2
    echo arg3 : $3
}

$ f1 11 , 22 , 33
arg1 : 11
arg2 : ,      # ' , ' 가 두번째 인수가 되었다.
arg3 : 22

-----

$ ARR=(11,22,33)

$ echo ${AA[0]}
11,22,33
$ echo ${AA[1]}

$ echo ${AA[2]}
```

Metacharacters Precedence

&& , ||

Shell script 에서 주의할 점 중의 하나가 `&&` , `||` 메타문자 우선순위입니다. 보통 프로그래밍 언어에서는 `&&` 가 `||` 보다 우선순위가 높지만 shell 메타문자는 우선순위를 같게 취급합니다. 따라서 다음과 같은 구문이 있을 경우

```
a || b && c
```

a 가 `true` 일 때 구문해석과 실행은 다음과 같습니다.

| 언어 | 해석 | 실행 |
|-------------|--|------|
| c/c++, java | <code>(a) (b && c)</code> | a |
| shell | <code>(a b) && c</code> | a, c |

결과적으로 프로그래밍 언어 에서는 a 가 실행이 된 후에 구문이 종료되나 shell 에서는 `(a || b)` 결과가 true 이므로 이후에 `(true && c)` 식이 실행되어 c 도 함께 실행되게 됩니다.

Redirection 우선순위

- 좌에서 우 로 실행 됩니다.

> 메타문자에 의해 z1, z2 두 파일의 내용은 삭제되며 echo 명령의 출력은 z2 파일로 가게 됩니다.

```
$ echo foofoo > z1 > z2
```

```
$ cat z1
```

```
$ cat z2
```

```
foofoo
```

- `{ ; } , ()` 바깥에서 안쪽으로 실행됩니다.

z1, z2 두 파일의 내용은 삭제되며 echo 명령의 출력은 z1 파일로 가게 됩니다.

```
$ { echo foobar > z1 ;} > z2

$ cat z1
foobar
$ cat z2
```

파이프 보다는 redirection 이 우선순위가 높다.

```
$ echo hello | cat
hello

$ echo hello | cat <<< "world"
world
```

&& || 보다는 파이프가 우선순위가 높다.

```
false | false && true | true ; echo $?
1

$ false | { false && true ;} | true ; echo $?
0
```

우선순위 조절

Shell 에서 메타문자 우선순위 조절은 { ;} , () 두가지 방법을 이용할 수 있습니다. () 는 subshell 이 생성돼야 하므로 가능하면 { ;} 를 사용하는게 좋겠습니다.

```
$ true || true && false ; echo $?
1

$ true || { true && false ;} ; echo $?
0

$ true || ( true && false ) ; echo $?
0
```

Expansions and substitutions

프롬프트에서 명령문을 작성한후 enter 키를 누르면 shell 은 먼저 명령문을 tokens (words) 으로 분리한 다음 해석해야할 표현식이 있을경우 변수확장, 산술확장, 명령치환 을 거쳐 최종 변경된 명령문을 만들게 됩니다. 그리고나서 마지막으로 불필요한 quotes 삭제 처리를 합니다. 확장과 치환 이 이루어 질때 shell 은 다음과 같은 순서로 진행합니다.

1 . Brace expansion

2 . Tilde expansion

3 . 다음 4 개는 left-to-right 순서로 동시에 처리 됩니다.

- Parameter expansion
- Arithmetic expansion
- Command substitution
- Process substitution

```
$ i=1
$ echo $i $((i++)) $i    # left-to-right 순서로 처리되므로
1 1 2                   # 결과로 '1 1 1' 이 아니라 '1 1 2' 가 나왔습니다.
```

4 . Word splitting

5 . Pathname expansion (globbing)

위 순서를 보시면 globbing 시에 공백이 들어간 파일이름이 왜 문제없이 처리되는지 알수있습니다.
(단어분리 이후에 처리되므로)

Brace Expansion

Brace 확장은 여러 확장과 치환 중에서 제일 먼저 일어나는 작업입니다. 그러므로 brace 확장에 사용되는 `start`, `end` range 값으로 변수를 사용할 수 없습니다 (변수확장이 뒤에 일어나므로). `string list` 에서 변수를 사용할 경우는 `"$AA"` or `${AA}` 와 같이 해야 오류가 발생하지 않습니다. brace 확장은 double quote 한 스트링 내에서는 일어나지 않습니다.

String lists

```
{string1,string2,...,stringN}
```

`,` 가 사용되지 않은 단일 항목은 확장되지 않으며 `,` 전,후에 공백을 사용할 수 없습니다. `string` 내에 공백이 포함될 경우 quote 합니다.

```
# ', ' 가 사용되지 않은 단일 항목은 확장되지 않는다.
$ echo {hello}
{hello}

# ', ' 전,후에 공백이 사용되면 확장이 되지 않는다.
$ echo X{apple, banana, orange, melon}Y
X{apple, banana, orange, melon}Y

# string 내에 공백이 포함될 경우 quote 한다.
$ echo X{apple, "ban ana", orange, melon}Y
XappleY Xban anaY XorangeY XmelonY
```

Preamble 또는 postscript 의 사용.

```

$ echo X{apple,banana,orange,melon}          # 여기서 X 는 preamble
Xapple Xbanana Xorange Xmelon

$ echo {apple,banana,orange,melon}Y          # Y 는 postscript
appleY bananaY orangeY melonY

# preamble, postscript 이 없을 경우 단지 space 로 분리되어 표시됩니다.
$ echo {apple,banana,orange,melon}
apple banana orange melon

$ echo X{apple,banana,orange,melon}Y
XappleY XbananaY XorangeY XmelonY

# '/home/bash/test/' 가 preamble 에 해당
$ mkdir /home/bash/test/{foo,bar,baz,cat,dog}
$ ls /home/bash/test/
bar/  baz/  cat/  dog/  foo/

$ tar -czvf backup-`date +%m-%d-%Y-%H%M`.tar.gz \
    --exclude={./public_html/cache,./public_html/compiled,./public_html/images} \
    ./public_html

# 'http://docs.example.com/slides_part' 는 preamble '.html' 는 postscript 에 해당
$ wget http://docs.example.com/slides_part{1,2,3,4}.html
$ ls
slides_part1.html slides_part2.html slides_part3.html slides_part4.html

```

null 값의 활용

```

$ echo -v{,,,,,}
-v -v -v -v -v -v

$ echo b{,,A}a
ba ba ba bAa

$ cp test.sh{,bak}
$ ls
test.sh test.sh.bak

$ ls shell/{,BB/}rc.d
shell/rc.d
...
shell/BB/rc.d
...

```

변수와 같이 사용한 예

```
$ AA=hello

# 변수확장이 뒤에 일어나므로 {a,b}$AA 는 a$AA b$AA 와 같게 됩니다.
$ echo {a,b}$AA
ahello bhello

# 그냥 $AA 로 사용하면 X$AAY 와 같아지므로 "$AA" 나 ${AA} 를 사용합니다.
$ echo X{apple,"$AA",orange,melon}Y
XappleY XhelloY XorangeY XmelonY
```

Ranges

```
{< START >...< END >}
{< START >...< END >...< INCR >}
```

Brace 확장은 변수확장 보다 먼저 일어납니다. 그러므로 start, end 값을 변수로 사용할 수 없습니다.

```
$ a=1 b=10

$ echo ${a..$b}          # brace 확장이 되지 않는다.
{1..10}
```

start, end 값으로 숫자와 알파벳을 사용할 수 있습니다.

```
$ echo {5..12}
5 6 7 8 9 10 11 12

$ echo {c..k}
c d e f g h i j k

$ echo {5..k}          # 숫자와 알파벳을 섞어서 사용할수는 없습니다.
{5..k}

##### Increment #####

$ echo {1..10..2}
1 3 5 7 9
$ echo {10..1..2}
10 8 6 4 2

$ echo {a..z..3}
a d g j m p s v y
```

zero 패딩

```
$ echo {01..10}
01 02 03 04 05 06 07 08 09 10

$ echo {0001..5}
0001 0002 0003 0004 0005

# img001.png ~ img999.png 까지 생성
printf "%s\n" img{00{1..9},0{10..99},{100..999}}.png

# 01 ~ 10 까지 생성
$ for i in 0{1..9} 10; do printf "%s\n" "$i"; done
```

Preamble 또는 postscript 의 사용.

```
$ echo 1.{0..9}
1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9

$ echo __{A..E}__
__A__ __B__ __C__ __D__ __E__

# 'http://docs.example.com/slides_part' 는 preamble '.html' 는 postscript 에 해당
$ wget http://docs.example.com/slides_part{1..4}.html
$ ls
slides_part1.html slides_part2.html slides_part3.html slides_part4.html
```

Combining and nesting

{ } 를 서로 붙이면 combining 이 일어납니다.

```
$ echo {A..Z}{0..9}
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 C0 C1 C2 C3 C4 C5 C6
C7 C8 C9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 F0 F1 F2 F3
F4 F5 F6 F7 F8 F9 G0 G1 G2 G3 G4 G5 G6 G7 G8 G9 H0 H1 H2 H3 H4 H5 H6 H7 H8 H9 I0
I1 I2 I3 I4 I5 I6 I7 I8 I9 J0 J1 J2 J3 J4 J5 J6 J7 J8 J9 K0 K1 K2 K3 K4 K5 K6 K7
K8 K9 L0 L1 L2 L3 L4 L5 L6 L7 L8 L9 M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 N0 N1 N2 N3 N4
N5 N6 N7 N8 N9 O0 O1 O2 O3 O4 O5 O6 O7 O8 O9 P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 Q0 Q1
Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 S0 S1 S2 S3 S4 S5 S6 S7 S8
S9 T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 U0 U1 U2 U3 U4 U5 U6 U7 U8 U9 V0 V1 V2 V3 V4 V5
V6 V7 V8 V9 W0 W1 W2 W3 W4 W5 W6 W7 W8 W9 X0 X1 X2 X3 X4 X5 X6 X7 X8 X9 Y0 Y1 Y2
Y3 Y4 Y5 Y6 Y7 Y8 Y9 Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9
```

{ } 안에서 , 로 분리하여 nesting 을 할수 있습니다.

```
$ echo {{A..Z},{a..z}}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s
```

Range 값으로 변수를 사용하려면?

eval 명령을 사용하면 range 값으로 변수를 사용할 수 있습니다.

```
$ a=1 b=5
$ eval echo ${a..$b}    # eval 명령을 이용하면 원하는 결과를 얻을수 있습니다.
1 2 3 4 5

$ eval echo img${a..$b}.png
img1.png img2.png img3.png img4.png img5.png
```

Brace 확장을 피하려면?

{ or } 를 escape 해주면 됩니다.

```
$ echo \{a..c\}
{a..c}

$ echo {a..c\}
{a..c}
```

Tilde Expansion

`~` 문자를 이용하여 디렉토리 정보를 표시합니다. `double quote` 내에서는 확장되지 않습니다.

Home 디렉토리

`~` 는 현재 사용자의 홈 디렉토리를 나타냅니다. `$HOME` 변수값과 같습니다.

`~USERID` 는 `USERID` 의 홈 디렉토리를 나타냅니다.

```
$ echo ~           # 현재 사용자의 홈 디렉토리를 나타냅니다.  
/home/bashhacker  
  
$ echo ~man        # 'man' 사용자의 홈 디렉토리를 나타냅니다.  
/var/cache/man
```

현재 디렉토리

`~+` 는 현재 디렉토리를 나타내며 `$PWD` 변수값과 같습니다.

이전 디렉토리

`~-` 는 이전 디렉토리를 나타내며 `$OLDPWD` 변수값과 같습니다.

Parameter Expansion

프로그래밍 언어에서 함수를 선언할때 인수를 받는 부분에 사용되는 이름을 매개변수 (parameter) 라 하고 함수 내에서 임의로 저장공간을 만들때 사용되는 이름을 변수 (variable) 라고 하듯이 매개변수와 변수는 사용상에는 차이가 없을지라도 내포하는 의미는 다르다고 할 수 있습니다. 편의상 본문에서 변수확장 이라는 용어를 사용하였지만 정확한 의미로는 매개변수 확장이 맞다고 할 수 있습니다. 아래서 살펴보겠지만 매개변수 확장은 단순히 변수이름을 값으로 바꾸는것이 아니라 여러가지 유용한 연산작업을 수행합니다.

sh에서는 substring expansion, search and replace, indirection, case modification 을 제외하고 모두 사용할 수 있습니다.

기본 사용법

AA=cde 표현식은 AA 라는 변수를 만들고 'cde' 라는 값을 저장합니다. 이 값을 사용하기 위해서는 \$ 문자를 변수이름 앞에 붙여 구분하는데 이것만으로는 부족할 때가 있습니다. 가령 변수 AA 값을 이용해 'abcdefg' 스트링을 만들고자 할때 echo "ab\$AAfg" 와 같이 한다면 변수 'AA' 가 아닌 'AAfg' 값이 적용되어 원하는 결과가 나오지 않게됩니다. 따라서 나머지 스트링과 구분하기 위해 변수이름에 { } 중괄호를 사용할 수 있게 하였는데 이것은 매개변수 확장을 위한 표현식을 작성할 때도 사용됩니다.

```
$ AA=cde

$ echo "ab$AAfg"      # 현재 변수 $AAfg 값은 null 이므로 'ac' 만 나온다.
ac

$ echo "ab${AA}fg"    # '{ }' 을 이용하면 나머지 스트링과 구분할 수 있다.
abcdefg
```

String length

```
${#PARAMETER}
```

변수이름 앞에 # 문자를 붙이면 변수가 가지는 값의 문자 수를 나타냅니다. array 의 @ , * 경우에는 전체 원소 개수를 나타냅니다.

```
$ AA="hello world"
$ echo ${#AA}
11
```

Array 의 경우

```
$ BB=( Arch Ubuntu Fedora Suse )
$ echo ${#BB[1]}      # [1] 번째 원소 Ubuntu 의 문자 수
6
$ echo ${#BB[@]}      # array BB 의 전체 원소 개수
4
```

Substring expansion

```
${PARAMETER:OFFSET}
${PARAMETER:OFFSET:LENGTH}
```

변수가 가지는 값에서 특정부분을 뽑아낼때 사용합니다. offset 위치에서부터 length 길이 만큼을 뽑아냅니다. offset 은 0 부터 시작하고 length 값을 주지 않으면 끝까지 해당됩니다.

offset 과 length 를 음수로 줄수 있는데 이경우 카운트를 값의 시작점에서부터 하지 않고 끝점에서부터 좌측 방향으로 하며 length 만큼 제외 하게 됩니다. offset 에 사용되는 음수는 매개변수 확장에 사용되는 `:-` 표현식과 겹치므로 `()` 를 사용해야 합니다.

```
AA="Arch Linux Ubuntu Fedora"

$ echo ${AA:11}
Ubuntu Fedora
$ echo ${AA:11:6}
Ubuntu
$ echo ${AA:(-6)}
Fedora
$ echo ${AA:(-6):2}
Fe
$ echo ${AA:(-6):-2}
Fedo
```

Array 의 경우

```
$ ARR=(11 22 33 44 55)

$ echo ${ARR[@]:2}
33 44 55

$ echo ${ARR[@]:1:2}
22 33
```

Positional parameters 의 경우


```
$ set -- 11 22 33 44 55

$ echo ${@:3}
33 44 55

$ echo ${@:2:2}
22 33
```

Substring removal

```
${PARAMETER#PATTERN}
${PARAMETER##PATTERN}
${PARAMETER%PATTERN}
${PARAMETER%%PATTERN}
```

변수가 가지는 값을 패턴을 이용해 매칭되는 부분을 잘라낼 때 사용합니다. `#` 기호는 패턴을 앞에서부터 적용한다는 뜻이고 `%` 기호는 패턴을 뒤에서부터 적용한다는 뜻입니다. 기호가 두 개로 중복된 것은 `longest match` 를 뜻하고 하나짜리는 `shortest match` 를 뜻합니다.

```
AA="this.is.a.inventory.tar.gz"

$ echo ${AA#*.}           # 앞에서 부터 shortest match
is.a.inventory.tar.gz

$ echo ${AA##*.}         # 앞에서 부터 longest match
gz

$ echo ${AA%.*}          # 뒤에서 부터 shortest match
this.is.a.inventory.tar

$ echo ${AA%%.*}          # 뒤에서 부터 longest match
this

# 디렉토리를 포함한 파일명에서 디렉토리와 파일명을 분리하기

AA="/home/bash/bash_hackers.txt"

$ echo ${AA%/*}           # 디렉토리 부분 구하기
/home/bash

$ echo ${AA##*/}          # 파일명 부분 구하기
bash_hackers.txt
```

Search and replace

```

${PARAMETER/PATTERN/STRING}
${PARAMETER//PATTERN/STRING}
${PARAMETER/PATTERN}
${PARAMETER//PATTERN}

```

변수가 가지는 값을 패턴을 이용해 매칭되는 부분을 다른 스트링으로 바꾸거나 삭제할때 사용합니다. 매칭되는 부분이 여러군데 나타날수 있는데 `//` 기호는 모두를 나타내고 `/` 는 첫번째 하나만 나타냅니다. 바꾸는 스트링 값을 주지 않으면 매칭되는 부분이 삭제됩니다. `array[@]` 의 각 원소에 대해서도 적용할수 있습니다.

```

$ AA="Arch Linux Ubuntu Linux Fedora Linux"

$ echo ${AA/Linux/Unix}           # Arch Linux 만 바뀌었다.
Arch Unix Ubuntu Linux Fedora Linux

$ echo ${AA//Linux/Unix}          # Linux 가 모두 Unix 로 바뀌었다
Arch Unix Ubuntu Unix Fedora Unix

$ echo ${AA//Linux/}              # 바꾸는 스트링을 주지 않으면 매칭되는 부분이 삭제된다.
Arch Ubuntu Fedora Suse

$ echo ${AA/Linux/}              Arch Ubuntu Linux Fedora Linux Suse Linux

```

`Array[@]` 는 각 원소별로 적용됩니다.

```

$ AA=( "Arch Linux" "Ubuntu Linux" "Fedora Linux" )

$ echo ${AA[@]/u/X}               # Ubuntu Linux 는 첫번째 'u' 만 바뀌었다
Arch LinXx UbXntu Linux Fedora LinXx

$ echo ${AA[@]//u/X}              # 이제 모두 바뀌었다
Arch LinXx UbXntX LinXx Fedora LinXx

```

Use a default value

```

${PARAMETER:-WORD}
${PARAMETER-WORD}

```

`${AA:-world}` : AA 변수값을 사용합니다. 그런데 AA 변수가 존재하지 않거나, null 값을 갖는 경우 world 를 사용합니다.

`${AA-world}` : AA 변수값을 사용합니다. 그런데 AA 변수가 존재하지 않는 경우 world 를 사용합니다. 다시 말해 이 표현식은 AA 값으로 null 도 허용하는것 입니다.

WORD 부분에도 변수확장을 사용할 수 있습니다.

```
$ AA=hello

$ echo ${AA:-world}      # AA 에 값이 있으면 AA 값을 사용한다
hello
$ echo ${AA-world}
hello

$ unset AA

$ echo ${AA:-world}      # AA 가 unset 되어 없는 상태이므로
world                   # 값은 world 가 된다.
$ echo ${AA-world}
world

$ AA=""

$ echo ${AA:-world}      # ':' 는 null 은 값이 없는것으로 보고 world 를 사용한다.
world
$ echo ${AA-world}      # '-' 는 null 도 값으로 취급하기 때문에
                        # 아무것도 표시되지 않는다.

# WORD 부분에도 변수확장을 사용할 수 있다.

$ AA=${AA:-$(date +%Y)}

$ AA=${FCEDIT:-${EDITOR:-vi}}
```

Array 의 경우

```
$ AA=( 11 22 33 )

$ echo ${AA[@]:-44 55 66}    # AA 에 값이 있으면 AA 값을 사용한다
11 22 33
$ echo ${AA[@]-44 55 66}
11 22 33

$ AA=( )                    # 또는 unset -v AA

$ echo ${AA[@]:-44 55 66}    # AA 가 unset 되어 존재하지 않는 상태이므로
44 55 66                    # 값은 44 55 66 이 된다.
$ echo ${AA[@]-44 55 66}
44 55 66

$ AA=("")

$ echo ${AA[@]:-44 55 66}    # ':-' 는 null 은 값이 없는것으로 보고 44 55 66 을 사용한다
44 55 66
$ echo ${AA[@]-44 55 66}    # '-' 는 null 도 값으로 취급하기 때문에
                             # 아무것도 표시되지 않는다.

$ AA=("" 77 88)

$ echo ${AA[@]:-44 55 66}
77 88
$ echo ${AA[@]-44 55 66}
77 88
```

Assign a default value

```
${PARAMETER:=WORD}
${PARAMETER=WORD}
```

이것은 위의 Use a default value 와 동일하게 동작합니다. 하지만 차이점이 하나 있는데요. 바로 대체값을 사용하게 될때 그값을 AA 변수에도 대입한다는 것입니다.

```
AA=""

$ echo ${AA:-world}
world
$ echo $AA                # ':-' 는 AA 변수에 값을 대입하지 않는다.

$ echo ${AA:=world}
world
$ echo $AA                # ':= ' 는 AA 변수에 값을 대입한다.
world
```

Array[@] 값은 대입되지 않습니다.

```
$ AA=( )

$ echo ${AA[@] :=11 22 33}
bash: AA[@]: bad array subscript
11 22 33
```

Use an alternate value

```
${PARAMETER:+WORD}
${PARAMETER+WORD}
```

`${AA:+world}` : AA 변수가 값을 가지고 있으면 `world` 를 사용합니다. 변수가 존재하지 않거나 값이 `null` 이면 `null` 을 리턴합니다.

`${AA+world}` : AA 변수가 값을 가지고 있으면 (`null` 값 포함) `world` 를 사용합니다. 변수가 존재하지 않으면 `null` 을 리턴합니다.

```
$ AA=hello

$ echo ${AA:+world}      # AA 에 값이 있으므로 대체값 world 를 사용한다.
world
$ echo ${AA+world}
world

$ AA=""

$ echo ${AA:+world}      # ':+ ' 는 null 은 값으로 취급하지 않기 때문에 null 을 리턴한다.

$ echo ${AA+world}      # ':' 는 null 도 값으로 취급하기 때문에 대체값 world 를 사용한다.
world

$ unset AA

$ echo ${AA:+world}      # 변수가 존재하지 않으므로 null 을 리턴한다.

$ echo ${AA+world}

# 함수이름을 갖는 FUNCNAME 변수가 있을 경우 뒤에 '()' 를 붙여서 프린트 하고 싶다면
echo ${FUNCNAME:+${FUNCNAME}() }
```

Display error if null or unset

```
${PARAMETER:?[error message]}
${PARAMETER?[error message]}
```

`${AA:?error message}` : AA 변수값을 사용합니다. 그런데 AA 변수가 존재하지 않거나 null 값이면 error message 를 출력하고 script 실행을 종료합니다. 프롬프트 에서는 \$? 값으로 1 을 리턴합니다.

`${AA?error message}` : AA 변수값을 사용합니다. 그런데 AA 변수가 존재하지 않으면 error message 를 출력하고 script 실행을 종료합니다. 프롬프트 에서는 \$? 값으로 1 을 리턴합니다.

error message 를 생략하면 'parameter null or not set' 메시지가 사용됩니다.

```
$ AA=hello

$ echo ${AA:?null or not set}      # AA 에 값이 있으므로 AA 값을 사용한다
hello
$ echo ${AA?not set}
hello

$ AA=""

$ echo ${AA:?null or not set}      # ':' 는 null 은 값으로 취급하지 않기 때문에
bash: AA: null or not set        # error message 를 출력하고 $? 값으로 1 을 리턴한다
$ echo $?
1
$ echo ${AA?not set}              # '?' 는 null 도 값으로 취급하기 때문에
                                  # 아무것도 표시되지 않는다.

$ unset AA

$ echo ${AA:?null or not set}      # 변수가 존재하지 않는 상태이므로
bash: AA: null or not set        # 모두 error message 를 출력 후 종료한다.
$ echo $?
1
$ echo ${AA?not set}
bash: AA: not set
$ echo $?
1
$ echo ${AA?}                     # error message 는 생략할 수 있다.
bash: AA: parameter null or not set
```

Case modification

```
${PARAMETER^}
${PARAMETER^^}
${PARAMETER,}
${PARAMETER,,}
```

`${PARAMETER^}` : 단어의 첫 문자를 대문자로 변경합니다.

`${PARAMETER^^}` : 단어의 모든 문자를 대문자로 변경합니다.

`${PARAMETER,}` : 단어의 첫 문자를 소문자로 변경합니다.

`${PARAMETER,,}` : 단어의 모든 문자를 소문자로 변경합니다.

```
$ AA=( "ubuntu" "fedora" "suse" )

$ echo ${AA[@]^}
Ubuntu Fedora Suse

$ echo ${AA[@]^}
UBUNTU FEDORA SUSE

$ AA=( "UBUNTU" "FEDORA" "SUSE" )

$ echo ${AA[@],}
uBUNTU fEDORA sUSE

$ echo ${AA[@],,}
ubuntu fedora suse
```

Indirection

`${!PARAMETER}`

스크립트 실행 중에 스트링으로 변수 이름을 만들어서 사용할 수 있습니다.

```
$ hello=123

$ world=hello

$ echo ${world}
hello

$ echo ${!world}    # '!world' 부분이 'hello' 로 바뀐다고 생각하면 됩니다.
123
```

함수에 전달된 인수를 표시할 때

```

----- args.sh -----
#!/bin/bash

for (( i = 0; i <= $#; i++ ))
do
    echo \$$i : ${!i}          # ${!i} 이렇게 하면 안됩니다.
done

-----

$ args.sh 11 22 33
$0 : ./args.sh
$1 : 11
$2 : 22
$3 : 33

```

함수에 array 인자를 전달할 때도 사용할 수 있다.

```

#!/bin/bash

foo() {
    echo "$1"

    local ARR=( "${!2}" )      # '!2' 부분이 'AA[@]' 로 바뀐다.

    for v in "${ARR[@]}; do
        echo "$v"
    done

    echo "$3"
}

AA=(22 33 44)
foo 11 'AA[@]' 55

##### output #####

11
22
33
44
55

```


Arithmetic Expression

`$(())` , `(())` 는 bash 에서 산술연산을 위해 특별히 제공하는 표현식입니다. sh 에서는 `$(())` 표현식만 사용할 수 있습니다. 산술연산을 하는 외부 명령으로는 `expr` 도 있습니다. 산술연산의 특징은 참, 거짓 값과 표현식 안에서 쓸수있는 연산자들이나 식들이 프로그래밍 언어와 같다는 점입니다. shell 에서 직접 해석하고 처리하므로 연산자를 `escape` 해야 한다든지 하는 제약 없이 편리하게 사용할수 있습니다. 좀 더 자세한 내용은 **Special Expressions** 메뉴를 참조하세요

array index 를 나타내는 `[]` 에서도 동일하게 산술연산을 할 수 있습니다. `#[]` 표현식은 deprecated 라고 하니 되도록이면 사용하지 말아야겠습니다.

Command Substitution

`$(<COMMANDS>)`

``<COMMANDS>``

명령치환은 표현식 안의 명령 실행 결과가 변수값 형태로 치환되어 사용되는 것으로 명령의 `stdout` 값이 사용됩니다. 표현식은 두가지 형태로 동일하게 동작합니다. `backtick` 은 괄호형 보다 타입하기가 편해서 비교적 간단한 명령을 작성할때 많이 사용합니다. 그런데 표현식을 열고 닫는 문자가 같은 관계로 `nesting` 하여 사용하기가 어렵습니다. 그래서 복잡한 명령을 작성하거나 `nesting` 이 필요할 때는 `$()` 을 사용하는게 좋습니다. 명령치환은 `subshell` 에서 실행됩니다.

```
$ AA=$( echo hello world )
$ echo $AA
hello world

$ AA=`pgrep -d, ibus`
$ echo $AA
17516,17529,17530,17538,17541
```

Quotes 이 중복돼도 된다

명령치환의 결과로 나온 값은 일반 변수를 사용할 때와 같이 단어분리 및 `globbing` 대상이 됩니다. 그러므로 공백문자를 유지해야 하거나 `globbing` 이 발생하면 안될 경우 항상 `double quotes` 을 해야 합니다. 명령문 자체에서도 `quotes` 을 많이 사용하는데 명령치환 표현식에도 사용하게 되면 `quotes` 이 중복되는 경우가 발생합니다. 이를 경우를 위해서 `bash` 는 `$(...)` 표현식을 만나면 그 안의 내용은 별도로 분리해서 개별적으로 처리한다고 합니다. 그러므로 `quotes` 중복문제를 피하려고 `escape` 하거나 할 필요 없이 그대로 명령문을 사용할 수 있습니다.

```
# 명령치환을 quote 하지 않은 경우
$ echo $( echo "
> I
> like
> winter      and      snow" )
I like winter and snow

# 명령치환을 quote 하여 공백과 라인개행이 유지되었다.
$ echo "$( echo "
> I
> like
> winter      and      snow" )"

I
like
winter      and      snow

-----

# quotes 이 여러번 중첩되어도 상관없다.

$ echo "$(echo "$(echo "$(date)")")")"
Thu Jul 23 18:34:33 KST 2015
```

null 문자를 보낼 수 없다

파이프와 달리 명령치환은 값으로 null 문자를 보낼 수 없습니다.

```
$ ls          # a, b, c 3개의 파일이 존재
a b c

# 파이프는 null 값이 정상적으로 전달된다.
$ find . -print0 | od -a
00000000  . nul  . /  a nul  . /  b nul  . /  c nul

# 명령치환은 null 값이 모두 제외되었다.
$ echo -n "$(find . -print0)" | od -a
00000000  . . /  a . /  b . /  c
```

변수에 값을 대입할때 마지막 **newline** 은 제거된다.

```
$ AA=$'hello\n'
$ echo -n "$AA" | od -a
00000000  h   e   l   l   o  nl      # 정상적으로 newline 이 표시된다.

$ AA=$(echo -n $'hello\n\n\n')
$ echo -n "$AA" | od -a
00000000  h   e   l   l   o      # 명령치환은 newline 이 모두 제거된다.
```

Process Substitution

```
<( <COMMANDS> )
```

```
>( <COMMANDS> )
```

Process 치환은 표현식이 명령문 상에서 입, 출력용 파일 형태로 치환되어 사용되는 것으로 명령의 인수로 파일을 사용하는 곳에서 사용될 수 있습니다. 하지만 일반적인 파일과는 다르므로 프로그램 내에서 파일 타입을 직접 체크하는 경우 실행되지 않을 수 있으며 파일을 읽거나 쓰기를 할 때 random access 를 할 수 없으므로 한번에 처음부터 끝까지 읽거나 써야 합니다. 표현식 내의 명령들은 subshell 에서 실행되며 실질적으로 file descriptor 가 pipe 에 연결되어 입출력이 됩니다.

```
# '>( )' 표현식 내의 명령은 subshell 에서 실행되므로 '$$' 값이 같게 나온다.

$ { echo '$$' : $$ >&2 ;} > >( echo '$$' : $$ )
$$ : 504
$$ : 504

# 하지만 '$BASHPID' 는 다르게 나온다.

$ { echo '$BASHPID' : $BASHPID >&2 ;} > >( echo '$BASHPID' : $BASHPID )
$BASHPID : 504
$BASHPID : 22037

-----

$ ls -l <( : )
lr-x----- 1 mug896 mug896 64 02.07.2015 22:29 /dev/fd/63 -> pipe:[681827]

$ [ -f <( : ) ]; echo $? # 일반적 파일인지 테스트
1
$ [ -p <( : ) ]; echo $? # pipe 인지 테스트
0
```

sleep 명령은 현재 shell 에서 실행되고 표현식 내의 cat 명령은 subshell 에서 실행되는 것을 볼 수 있습니다.

```
{ sleep 10 ;} > >( cat )
```

| PID | TTY | STAT | TIME | COMMAND |
|-------|-------|------|------|-------------|
| 26758 | pts/1 | Ss | 1:09 | bash |
| 6303 | pts/1 | S | 0:00 | _ bash |
| 6305 | pts/1 | S | 0:00 | _ cat |
| 6323 | pts/1 | S+ | 0:00 | _ sleep 10 |

현재 shell 과 표현식 subshell 에서의 FD

`command1 > >(command2)` 명령의 경우 `command1` 의 실행결과(stdout) 가 `command2` 의 stdin 으로 입력되게 되며 `command1 < <(command2)` 명령의 경우는 `command2` 의 실행결과(stdout) 가 `command1` 의 stdin 으로 입력되게 됩니다.

현재 shell pid 를 나타내는 `$$` 변수는 subshell 에서도 동일한 값을 가지므로 `>()` 표현식 내에서의 FD 상태를 보기 위해서는 `$BASHPID` 변수를 이용해야 합니다.

>(...)

현재 shell 의 stdout 이 파이프에 연결되어 출력되고 표현식 내의 subshell 에서는 stdin 이 파이프에 연결되어 입력을 받고 있습니다.

```
ES { ls -l /proc/$$/fd >&2 ;} > >( : )
total 0
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 0 -> /dev/pts/13
l-wx----- 1 mug896 mug896 64 04.07.2015 00:32 1 -> pipe:[1028154]
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 10 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 255 -> /dev/pts/13

ES { : ;} > >( ls -l /proc/$BASHPID/fd >&2 )
total 0
lr-x----- 1 mug896 mug896 64 04.07.2015 21:12 0 -> pipe:[1026789]
lrwx----- 1 mug896 mug896 64 04.07.2015 21:12 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:12 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:12 255 -> /dev/pts/13
```

<(...)

표현식 내의 subshell 은 stdout 이 파이프에 연결되어 출력되고 현재 shell 에서는 stdin 이 파이프에 연결되어 입력을 받고 있습니다.

```

ES$ { ls -l /proc/$$/fd >&2 ;} < <( : )
total 0
lr-x----- 1 mug896 mug896 64 04.07.2015 00:32 0 -> pipe:[1026815]
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 10 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 255 -> /dev/pts/13

ES$ { : ;} < <( ls -l /proc/$BASHPID/fd >&2 )
total 0
lrwx----- 1 mug896 mug896 64 04.07.2015 21:14 0 -> /dev/pts/13
l-wx----- 1 mug896 mug896 64 04.07.2015 21:14 1 -> pipe:[1026829]
lrwx----- 1 mug896 mug896 64 04.07.2015 21:14 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:14 255 -> /dev/pts/13

```

사용예)

process 치환을 사용하는 이유는 임시 파일을 만들지 않아도 된다는 점입니다. 가령 ulimit 명령의 soft limit 과 hard limit 출력값을 서로 비교한다면 아래와 같이 명령 실행 결과를 임시파일로 만든 후 비교해야 합니다. 하지만 process 치환을 이용하면 내부적으로 파이프를 이용해 처리하기 때문에 임시파일을 만들 필요가 없습니다.

```

$ ulimit -Sa > ulimit.Sa.out

$ ulimit -Ha > ulimit.Ha.out

$ diff ulimit.Sa.out ulimit.Ha.out

```

process 치환을 사용해 비교

```

# 임시파일을 만들 필요가 없다

$ diff <( ulimit -Sa ) <( ulimit -Ha )
1c1
< core file size          (blocks, -c) 0
---
> core file size          (blocks, -c) unlimited
8c8
< open files              (-n) 1024
---
> open files              (-n) 65536
12c12
< stack size              (kbytes, -s) 8192
---
> stack size              (kbytes, -s) unlimited

```

위의 process 치환을 이용한 비교는 사실 다음과 동일하다고 볼수있습니다.

```
mkfifo fifo1
mkfifo fifo2
ulimit -Sa > fifo1 &
ulimit -Ha > fifo2 &
diff fifo1 fifo2
rm fifo1 fifo2
```

```
$ echo hello > >( wc )
      1      1      6

$ wc < <( echo hello )
      1      1      6

-----

# 입력과 출력용 process 치환을 동시에 사용
$ f1() {
    cat "$1" > "$2"
}

$ f1 <( echo 'hi there' ) >( tr a-z A-Z )
HI THERE

-----

# --log-file 옵션 값으로 입력 process 치환이 사용됨
$ rsync -arv --log-file=>(grep -vF .tmp > log.txt) src/ host::dst/

-----

# tee 명령을 이용해 결과를 4개의 입력 process 치환으로 전달하여 처리
$ ps -ef | tee >(grep tom > toms-procs.txt) \
    >(grep root > roots-procs.txt) \
    >(grep -v httpd > not-apache-procs.txt) \
    >(sed 1d | awk '{print $1}' > pids-only.txt)

-----

# dd 명령에서 입력 파일로 사용
dd if=<( cat /dev/urandom | tr -dc A-Z ) of=outfile bs=4096 count=1
```

스크립트 작성시에 명령 실행 결과를 받아서 처리하고자 할때 많이 사용하는 방법이 `|` 인데 파이프는 연결된 모든 명령들이 각자의 subshell 에서 실행되어 parent 변수에 연산 결과를 저장할 수가 없습니다. 이때 process 치환을 이용하면 현재 shell 에서 처리할 수 있습니다.


```
i=0
sort list.txt | while read -r line; do
    (( i++ ))
    ...
done
```

파이프로 인해 parent 변수 i 에 값을 설정할수 없어 항상 0 이 표시된다.

```
echo "$i lines processed"
0 lines processed
```

```
i=0
while read -r line; do
    (( i++ ))
    ...
done < <(sort list.txt)
```

process 치환을 이용해 while 문이 현재 shell 에서 실행되어 i 값을 설정할수 있다.

```
echo "$i lines processed"
12 lines processed
```

Word Splitting

단어분리는 변수확장, 명령치환 과 함께 일어나는 작업으로 다음과 같은 경우는 발생하지 않습니다.

```
$ set -f; IFS=:
$ ARR=(Arch Linux:Ubuntu Linux:Suse Linux:Fedora Linux)
$ set +f; IFS=$' \t\n'

$ echo ${#ARR[@]}      # 올바르게 분리 되지 않는다.
5
$ echo ${ARR[1]}
Linux:Ubuntu
```

\$AA 변수확장에 대해서는 단어분리가 일어난다.

```
$ AA="Arch Linux:Ubuntu Linux:Suse Linux:Fedora Linux"

$ set -f; IFS=:
$ ARR=( $AA )
$ set +f; IFS=$' \t\n'

$ echo ${#ARR[@]}      # 올바르게 분리 되었다.
4
$ echo ${ARR[1]}
Ubuntu Linux
```

단어분리는 IFS 값에 의해 분리가 되는데 IFS 의 기본값은 **space**, **tab**, **newline** 입니다. 이말은 입력되는 데이터에서 **space** 나 **tab**, **newline** 문자를 만나면 단어가 분리된다는 뜻입니다. IFS 값이 적용되는 방식은 공백문자일 때와 아닐 경우를 나누어 볼 수 있는데 공백문자일 경우는 먼저 연이어진 공백문자들을 하나의 **space** 로 줄입니다. 하지만 공백문자가 아닐 경우는 각각 하나의 **space** 로 치환합니다.

Internal Field Separator (IFS)

단어분리 시 이 값을 기준으로 단어가 분리됩니다. read 명령으로 읽어들이는 라인을 필드로 분리할 때, array 변수에 원소들을 분리하여 입력할 때도 사용됩니다. 기본값은 **space**, **tab**, **newline** 으로 IFS 변수가 unset 됐을때도 적용됩니다. IFS 값이 null 이면 단어분리가 일어나지 않습니다.

```
$ echo -n "$IFS" | od -a
00000000 sp ht nl
```

다음은 IFS 값이 공백문자일 경우와 아닐 경우 차이를 비교한 것입니다.

```
$ AA="Arch:Ubuntu:::Mint"

$ IFS=: # 공백이 아닌 문자를 사용하는 경우

$ ARR=( $AA )

$ echo ${#ARR[@]} # 원소 개수가 빈 항목을 포함하여 5 개로 나온다.
5

$ echo ${ARR[1]}
Ubuntu
$ echo ${ARR[2]}

$ echo ${ARR[3]}

-----

AA="Arch Ubuntu      Mint"

$ IFS=' ' # 공백문자를 사용하는 경우

$ ARR=( $AA )

$ echo ${#ARR[@]} # IFS 값이 공백문자일 경우 연이어진 공백문자들은 하나로 줄어듭니다.
3 # 그러므로 원소 개수가 3 개로 나온다

$ echo ${ARR[1]}
Ubuntu
$ echo ${ARR[2]}
Mint
```

Double quote 을 하면 단어분리가 일어나지 않습니다

```
AA="echo hello world"

# quote 을 하지 않아 단어분리가 일어나 echo 는 명령 hello, world 는 인수가 되었다.
$ $AA
hello world

# quote 을 하여 'echo hello world' 가 하나의 명령이 되었다.
$ "$AA"
echo hello world: command not found
```

Script 작성시 주의할점

파일 이름에 space 가 포함되어 있을 경우 단어분리에 의해 파일 이름이 분리가 될 수 있습니다. 아래는 find 명령치환 값이 단어분리가 일어나는 예입니다. IFS 값을 `$'\n'` 으로 변경하여 실행하면 문제를 해결할 수 있습니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class      쉘 스크립트 테스트.txt  WriteObject.java
```

```
$ for file in $(find .)
do
    echo "$file"
done
.
./WriteObject.java
./WriteObject.class
./ReadObject.java
./2013-03-19          # 파일이름이 2개로 분리
154412.csv
./ReadObject.class
./셸                  # 파일이름이 3개로 분리
스크립트
테스팅.txt
```

```
-----
$ set -f; IFS=$'\n'      # IFS 값을 newline 으로 변경
```

```
$ for file in $(find .)
do
    echo "$file"
done
.
./WriteObject.java
./WriteObject.class
./ReadObject.java
./2013-03-19 154412.csv
./ReadObject.class
./셸 스크립트 테스트.txt

$ set +f; IFS=$' \t\n'
```

Filename Expansion (Globbing)

디렉토리 에서 파일을 조회할 때 * 문자를 사용해본 적이 있을 겁니다. 프롬프트 에서 `ls *.sh` 명령을 실행하면 확장자가 `.sh` 인 파일들을 모두 볼 수 있습니다. 여기서 사용된 * 문자를 glob 문자라 하고 glob 문자에 의해 매칭된 파일들로 치환되는 것을 globbing 이라고 합니다.

glob 문자는 * 외에 ? [] 도 사용할 수 있으며 사실 shell 에서 제공하는 pattern matching 과 동일하게 동작합니다. globbing 은 꼭 파일이름을 다룰 때만 적용되는 것이 아니며 어떤 스트링 이나 변수값에라도 glob 문자가 있으면 발생하므로 주의해야 합니다. quote 을 하면 단어분리, globbing 둘 다 일어나지 않습니다.

```
$ ls
address.c      address.h      readObject.c    readObject.h    WriteObject.class
Address.class  Address.java   ReadObject.class ReadObject.java  WriteObject.java

$ ls *. [ch]
address.c  address.h  readObject.c  readObject.h

$ ls "*. [ch]"      # quote 을 하면 globbing 이 일어나지 않는다
ls: cannot access *. [ch]: No such file or directory

$ echo *.?
address.c address.h readObject.c readObject.h

$ for file in *. [ch]; do echo "$file"; done
address.c
address.h
readObject.c
readObject.h
```

shell 에서 파일을 select 하는 제일 좋은 방법은 globbing 을 이용하는 것입니다. 다음과 같이 사용할 경우는 단어분리, globbing 에 대한 처리를 해주어야 합니다.

```

for file in $(find . -type f) ... # Wrong!
for file in `find . -type f` ... # Wrong!

arr=( $(find . -type f) ) ... # Wrong!

-----
# 단어분리, globbing 에대한 처리를 해주어야 합니다.
# 이방법은 파일 이름에 newline 이 포함될 경우에는 사용할수 없습니다.

set -f; IFS=$'\n'
for file in $(find . -type f) ...
set +f; IFS=$' \t\n'

set -f; IFS=$'\n'
arr=( $(find . -type f) ) ...
set +f; IFS=$' \t\n'

-----
# find 명령에서 -print0 을 이용해 출력했으므로 -d 값을 null 로 설정
# 파일이름이 단어분리가 되는것을 금지하기 위해 IFS 값도 null 로 설정

find . -type f -print0 | while IFS='' read -r -d '' file; do
    echo "$file"
done

# process 치환을 이용
while IFS= read -rd '' file; do
    echo "$file"
done < <( find . -type f -print0 )

```

globbing 은 단어분리 후에 일어나므로 파일이름에 space 가 있어도 문제가 되지 않습니다.

```

$ for file in *
> do
>     echo "$file"
> done
2013-03-19 154412.csv
ReadObject.class
ReadObject.java
셸 스크립트 테스트.txt
WriteObject.class
WriteObject.java

```

Globbing 관련 shell 옵션, 변수

-f | noglob

`set -o noglob` 은 globbing 기능을 disable 합니다. globbing 을 회피하고자 할 때 사용할 수 있습니다. globbing 과 패턴매칭은 별개의 기능으로 globbing 을 disable 한다고 해서 패턴매칭에 glob 문자를 사용할 수 없는것은 아닙니다.

nullglob

glob 문자를 이용하여 매칭을 시도하였으나 매칭되는 파일이 없을 경우 패턴을 그대로 리턴합니다. 이것은 매칭되는 파일을 처리하는 스크립트 에서 오류의 원인이 될수 있습니다. 이때 `shopt -s nullglob` 옵션을 설정하면 매칭되는 파일이 없을 경우 패턴을 리턴하지 않습니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class      쉘 스크립트 테스트.txt WriteObject.java
```

```
$ echo *.sh
*.sh          # 매칭되는 파일이 없을경우 패턴을 그대로 리턴
```

```
$ shopt -s nullglob
```

```
$ echo *.sh
          # 매칭되는 파일이 없을경우 null 을 리턴
```

매칭되는 파일이 없을때 패턴을 그대로 리턴하면 오류의 원인이 될수있다.

```
$ for f in *.sh; do
>     cat "$f"
> done
cat: *.sh: No such file or directory
```

nullglob 옵션을 설정하면 오류를 없앨 수 있다

```
$ shopt -s nullglob
```

```
$ for f in *.sh; do
      cat "$f"
done
```

nullglob 을 설정하지 않고 다음과 같이 할수도 있습니다.

```
for f in *.log; do
    [ -e "$f" ] || continue
    ...
done
```

하지만 매칭되는 파일이 없을때 패턴을 리턴하지 않고 null 을 리턴하는 것이 다음과 같은 경우에는 오류로 이어질 수 있습니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class      쉘 스크립트 테스트.txt  WriteObject.java

$ array=(11 22 33)
$ shopt -s nullglob

# 현재 디렉토리에 array[1] 에 매칭이 되는 파일이 없으므로 null 을 리턴
$ unset -v array[1]

$ echo ${array[1]}      # unset 이 되지 않고 값이 남아 있다.
22

# globbing 을 disable 하기 위해 다음과 같이 quote 을 해야한다.
$ unset -v "array[1]"
```

failglob

Globbering 매칭 실패시 오류메시지와 함께 `$?` 값으로 `1` 을 리턴합니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class      쉘 스크립트 테스트.txt  WriteObject.java

$ echo *.sh
*.sh

$ echo $?
0

$ shopt -s failglob

$ echo *.sh
bash: no match: *.sh

$ echo $?      # failglob 설정후에 매칭실패시 '1' 을 리턴한다.
1
```

nocaseglob

이 옵션을 설정하면 매칭 시에 대, 소문자 구분을 하지 않습니다.

dotglob

이 옵션을 사용하면 `.filename` 와 같이 점으로 시작하는 파일도 매칭에 포함시킵니다.

globstar

****** 문자를 이용하여 하위 디렉토리 까지 recursive 매칭을 할수있습니다.

```
$ echo **          # 모든 파일, 디렉토리
$ echo **/         # 모든 디렉토리
$ echo **/*.sh     # 확장자가 .sh 인 모든파일
```

globasciiranges

C locale 이 아닐 경우 `[a-c]` 와 같이 `-` 문자를 이용한 range 매칭시에 기본적으로 대, 소문자 구분을 하지 않습니다. 다시 말해 `[aAbBcCc]` 와 같은 의미가 되는데요. 이 옵션을 설정하면 대, 소문자 구분을 합니다.

GLOBIGNORE

이건 shell 옵션이 아니고 환경변수 입니다. globbing 에서 사용하는 패턴을 `:` 로 분리해서 등록해 놓으면 매칭에서 제외시킵니다.

Glob 문자가 들어간 스트링은 주의!

명령문의 인수로 사용되는것은 quote 하지 않는 이상 모두 globbing 대상입니다. 그러므로 스크립트를 실행중인 디렉토리에 매칭 되는 파일이 있을경우 예상치 못한 오류가 발생할 수 있습니다.

다음은 unset 명령의 인수로 사용된 `array[12]` 이 globbing 에 의해 파일 `array1` 과 매칭이 되어 unset 이 되지 않고 있습니다.

```

$ array=( [10]=100 [11]=200 [12]=300 )
$ echo ${array[12]}
300

$ touch array1                # 현재 디렉토리에 임의로 array1 파일생성

# unset 을 실행하였으나 globbing 에 의해 array[12] 이 array1 파일과 매칭이되어
# 값이 그대로 남아있음
$ unset -v array[12]
$ echo ${array[12]}
300

$ unset -v "array[12]"        # globbing 을 disable 하기위해 quote.
$ echo ${array[12]}           # 이제 정상적으로 unset 이됨

```

array 에 입력되는 원소 값에 glob 문자가 포함됨

```

$ AA="Arch Linux:*:Suse Linux:Fedora Linux"

$ IFS=:
$ ARR=($AA)
$ IFS=$' \t\n'

$ echo "${ARR[@]}"
Arch Linux 2013-03-19 154412.csv Address.java address.ser
ReadObject.class ReadObject.java 쉘 스크립트 테스트.txt
WriteObject.class WriteObject.java Suse Linux Fedora Linux

-----

$ set -f; IFS=:
$ ARR=($AA)
$ set +f; IFS=$' \t\n'

$ echo "${ARR[@]}"
Arch Linux * Suse Linux Fedora Linux

```

위의 예를 통해서 알 수 있듯이 사용되는 스트링에 glob 문자가 있을 경우 항상 quote 해서 사용하거나 escape 해야 하며 필요할 경우 `set -o noglob` 옵션을 사용해야 합니다.

Redirection

명령을 사용할때 항상 키보드에서 입력을 받고 화면으로 출력하지는 않습니다. 특정 파일의 내용을 입력으로 받아서 처리결과를 화면이 아닌 또 다른 파일이나 명령으로 보낼 수도 있습니다. 리눅스에서는 모든 장치들을 파일로 관리합니다. 키보드를 통해 입력을 받고 결과와 에러 메시지를 화면에 출력하는 것도 모두 파일을 통해 처리됩니다. 시스템에는 입력을 담당하는 `stdin`, 정상출력을 위한 `stdout`, 에러출력을 위한 `stderr` 가 있는데 터미널을 열면 이 3 개가 터미널에 연결되어 명령의 입, 출력과 에러를 표시하는데 사용됩니다. 우리가 보통 사용하는 파일들은 보기 좋은 이름이 있지만 실제 프로그램 내에서는 FD (file descriptor) 번호에 의해 처리됩니다. 관례에 따라 `stdin` 은 0 번, `stdout` 은 1 번, `stderr` 는 2 번을 사용하며 redirection 에서도 이 번호를 사용합니다.

기본 사용법 과 default value

```
$ cat infile
hello
world

$ wc 0< infile 1> outfile
$ cat outfile
2 2 12
```

위의 `wc` 명령은 redirection 의 기본사용법을 보여줍니다. 데이터 흐름의 방향을 나타내는 redirection 기호 `<` , `>` 를 가운데 두고 왼쪽에는 FD (file descriptor), 오른쪽에는 FD 나 filename 을 위치시키면 됩니다.

위의 예에서는 `infile` 을 FD 0 (`stdin`) 에 연결하여 입력으로 사용하였고 FD 1 (`stdout`) 을 `outfile` 파일에 연결하여 출력이 `outfile` 파일에 저장되게 하였습니다. `<` 기호는 입력을 나타내므로 좌측값 0 을 기본값으로 `>` 기호는 출력을 나타내므로 좌측값 1 을 기본값으로 합니다. 따라서 위의 예는 다음과 같이 작성하여도 결과가 같습니다.

```
$ wc < infile > outfile
$ cat outfile
2 2 12
```

Redirection 기호 사용시 주의할점

`<` , `>` 기호 좌측값은 공백없이 붙여야합니다. 그렇지 않으면 명령의 인수로 인식이 됩니다.

```
$ wc 0 < infile 1> outfile          # 0 을 wc 명령의 인수로 인식.
wc: 0: No such file or directory

$ wc 0< infile 1 > outfile          # 1 을 wc 명령의 인수로 인식.
wc: 1: No such file or directory
```

> 기호의 우측값은 파일이름이 올경우는 괜찮지만 FD 번호가 올경우는 & 기호를 붙여줘야 합니다. 그렇지 않으면 FD 숫자가 파일이름이 됩니다.

```
# '>' 기호에 ' ' 를 붙이지 않으면 FD 번호를 파일이름으로 인식
$ wc asdfgh 2>1
$ cat 1
wc: asdfgh: No such file or directory
```

정리하면 이렇습니다. redirection 기호를 중심으로 왼쪽에 오는 FD 는 명령의 인수로 인식되지 않게 붙여쓰고 오른쪽에 오는 FD 는 파일로 인식되지 않게 & 를 붙여 사용하면 됩니다.

Redirection 기호의 위치

redirection 기호의 명령행상 위치는 어디에 와도 상관없습니다.

```
$ echo hello > /tmp/example

$ echo > /tmp/example hello

$ > /tmp/example echo hello
```

Redirection 기호는 쓰는 순서가 중요하다!

redirection 을 할때 중요한 부분이 순서입니다. 순서가 올바르지 않으면 제대로 redirection 이 되지 않습니다. 다음은 mycomm 실행결과 FD 2 (stderr) 를 FD 1 (stdout) 으로 연결하여 둘 모두를 outfile 에 쓰려고 하지만 이중 하나는 문제가 있습니다.

```
$ mycomm 2>&1 > outfile          # 첫번째

$ mycomm > outfile 2>&1          # 두번째
```

위문장을 exec 명령으로 단계적으로 살펴해보도록 하겠습니다. exec 명령은 현재 shell 을 지정한 명령으로 대체하기 위해 사용하기도 하지만 redirection 설정을 현재 shell 에 적용시키기 위해서도 사용합니다. 터미널을 열어 `ls -l /proc/$$/fd` 을 실행해보면 FD 0, 1, 2 모두 터미널 `/dev/pts/10` 에 (저같은경우) 연결돼 있는것을 볼수있습니다.

```
# '$$' 는 현재 shell process id 를 나타냅니다.
$ ls -l /proc/$$/fd
total 0
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 0 -> /dev/pts/10 # stdin
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 1 -> /dev/pts/10 # stdout
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 2 -> /dev/pts/10 # stderr
```

exec 명령을 실행하는 터미널이 중지될수 있으니 조희용 터미널을 하나 더 열어서 사용하시기 바랍니다. 이때 shell process id 값은 exec 명령을 실행하는 터미널에서 `echo $$` 로 구할수 있습니다.

첫번째 mycomm 2>&1 > outfile

```
$ exec 2>&1
```

FD 2 번을 FD 1 번에 연결 하였습니다. 현재 1, 2 번은 모두 동일한 터미널에 연결돼 있으므로 변경사항이 없습니다. 다만 stderr 가 stdout 으로 변경됩니다.

```
$ ls -l /proc/20040/fd
total 0
lrwx----- 1 mug896 mug896 64 07.06.2015 15:56 0 -> /dev/pts/10 # stdin
lrwx----- 1 mug896 mug896 64 07.06.2015 15:56 1 -> /dev/pts/10 # stdout
lrwx----- 1 mug896 mug896 64 07.06.2015 15:56 2 -> /dev/pts/10 # stdout 으로 변경
```

```
$ exec 1> outfile
```

FD 1 번을 outfile 로 연결하여 1 번 연결이 변경되었습니다. redirection 처리후에 FD 1 번만 outfile 로 연결된것을 볼 수 있습니다. 그러므로 첫번째 명령은 FD 1 번만 outfile 로 출력되고 FD 2 번은 터미널을 통해 stdout 으로 출력 됩니다.

```
$ ls -l /proc/20040/fd
total 0
lrwx----- 1 mug896 mug896 64 07.06.2015 15:56 0 -> /dev/pts/10 # stdin
l-wx----- 1 mug896 mug896 64 07.06.2015 15:56 1 -> /home/mug896/outfile
lrwx----- 1 mug896 mug896 64 07.06.2015 15:56 2 -> /dev/pts/10 # stdout
```

오류메시지는 터미널을 통해 stdout 으로 출력되므로 다음과 같이 하면 errmessage 파일로 저장할 수 있습니다.

```
# { ; } 를 사용해야 합니다 그렇지 않으면 outfile 내용이 errmessage 로 가게 됩니다.
{ mycomm 2>&1 > outfile ; } > errmessage
```

두번째 mycomm > outfile 2>&1

```
$ exec 1> outfile
```

FD 1 번을 outfile 에 연결하여 1 번 연결이 변경되었습니다.

```
$ ls -l /proc/19779/fd
total 0
lrwx----- 1 mug896 mug896 64 07.06.2015 15:33 0 -> /dev/pts/11
l-wx----- 1 mug896 mug896 64 07.06.2015 15:33 1 -> /home/mug896/tmp/outfile
lrwx----- 1 mug896 mug896 64 07.06.2015 15:33 2 -> /dev/pts/11
```

```
$ exec 2>&1
```

FD 2 번을 FD 1 번에 연결하였습니다. 현재 1 번은 outfile 에 연결된 상태이므로 2 번도 따라서 outfile 로 연결됩니다. redirection 처리후에 FD 1, 2 모두 outfile 로 연결된 것을 볼 수 있습니다. 그러므로 두번째 명령은 FD 1, 2 둘다 outfile 로 출력되게 됩니다.

```
$ ls -l /proc/19779/fd
total 0
lrwx----- 1 mug896 mug896 64 07.06.2015 15:33 0 -> /dev/pts/11 # stdin
l-wx----- 1 mug896 mug896 64 07.06.2015 15:33 1 -> /home/mug896/tmp/outfile
l-wx----- 1 mug896 mug896 64 07.06.2015 15:33 2 -> /home/mug896/tmp/outfile
```

mycomm > outfile 2>&1 을 줄여서 mycomm &> outfile 로 쓸수도 있습니다.

Script 작성시 redirection 의 활용

스크립트 실행 시에 FD 0, 1, 2 번은 연결이 터미널에서 파일이나 파이프로 변경될 수 있으므로 사용하기전 백업과 복구 과정을 거쳐야 합니다. 백업 FD 를 삭제할 때는 간단하게 복구 과정에서 뒤에 - 기호를 붙이면 됩니다.

```
exec 3>&1          # FD 3 번을 새로 생성하면서 1번에 연결하였습니다.
                  # 그러므로 1번 연결이 3번에 백업된 것과 같습니다.

exec 1> outfile    # FD 1 번을 outfile 에 연결해 사용
...
...
exec 1>&3-          # FD 1 번을 백업해 두었던 3 번에 다시 연결하고 3 번은 삭제 (3-)
```

스크립트 실행시 stdin 을 redirect 하여 사용후 복구하는 과정입니다.

```
#!/bin/bash

exec 3<&0          # FD 0 번을 사용하기 전에 3 번에 백업
exec 0< infile    # FD 0 을 infile 로 연결 ( 실행하려면 infile 을 준비하세요 )

read var1         # 2 개의 라인을 읽음.
read var2

echo read from infile: $var1
echo read from infile: $var2

exec 0<&3-        # FD 0 번을 3 번 으로부터 복구하고 3 번은 삭제 (3-)

# 프롬프트가 보이면 정상적으로 실행된 것이다.
read -p "enter your favorite number : " var
echo $var
```

스크립트 실행시 stdout 을 redirect 하여 사용후 복구하는 과정입니다.

```
#!/bin/bash

echo start-----

# FD 1 을 outfile 로 연결. 이제 모든 메시지는 outfile 로 저장됨

exec 3>&1          # FD 1 번 백업
exec 1> outfile

echo this message will go to outfile

exec 1>&3-        # FD 1 번 복구

# 마지막 메시지가 보이면 정상적으로 실행된 것이다
echo end-----
```

다음은 스크립트 실행시 stdin, stdout 을 redirect 하여 사용후 복구하는 과정입니다.

```
#!/bin/bash

echo start -----

exec 3<&0 4>&1          # FD 0, 1 번 백업
exec 0< infile         # ( 실행하려면 infile 파일을 준비하세요 )
exec 1> outfile

# stdin(0) 으로 입력을 받아서 'tr' 처리후 stdout(1) 으로 출력.
# 현재 redirection 을 한 상태이므로 infile 에서 읽어들여 outfile 로 출력하게 됩니다.

cat - | tr a-z A-Z

exec 0<&3- 1>&4-        # FD 0, 1 번 복구

# 다음 마지막 메시지가 보이고 outfile 파일 내용이 모두 대문자로 변경 돼 있으면
# 정상적으로 실행된 것입니다.

echo end -----
```

Subshell 에서 실행되는 파이프를 피해 redirection 을 이용해 while 로 파일을 처리하는 예입니다.


```
#!/bin/bash

exec 3<&0
exec 0< infile

lines=0
while read -r line          # 기본적으로 stdin 에서 읽으므로
do
    echo "$line"
    (( lines++ ))
done

exec 0<&3-

echo number of lines read : $lines

-----
#!/bin/bash

exec 3< infile

lines=0
while read -r line
do
    echo "$line"
    (( lines++ ))
done <& 3                    # 'done < &3' 이렇게 사용하지 않습니다.

exec 3<&-

echo number of lines read : $lines

-----
#!/bin/bash

exec 3< infile

lines=0
while read -r line <& 3      # FD 이므로 여기에 놓고 사용해도 된다.
do
    echo "$line"
    (( lines++ ))
done

exec 3<&-

echo number of lines read : $lines
```

동일한 파일을 입, 출력에 사용하면?

하나의 파일을 redirection 을 이용해 동시에 입, 출력에 사용하는 것은 두개의 FD 를 열어서 각각의 파일포지션을 이용하는 것과 같습니다.

입, 출력에 쓰일 파일 x 의 내용

```
$ cat x
000000000000000000
111111111111111111
```

파일 x 로부터 입력을 받고 동일한 파일로 출력을 하여 append 할 경우.

순서대로 ls 명령의 출력이 x 파일에 append 되고 다음에 sed 명령에서는 append 된 내용이 함께 입력으로 사용되는 걸 볼 수 있습니다. (0 이 모두 X 로 바뀜)

```
$ { { ls -l /dev/fd/; sed -e 's/0/X/g' ;} >> x ;} < x

$ cat x
000000000000000000
111111111111111111
total 0
lr-x----- 1 mug896 mug896 64 08.03.2015 10:06 0 -> /home/mug896/tmp/3/x
l-wx----- 1 mug896 mug896 64 08.03.2015 10:06 1 -> /home/mug896/tmp/3/x
lrwx----- 1 mug896 mug896 64 08.03.2015 10:06 2 -> /dev/pts/8
lr-x----- 1 mug896 mug896 64 08.03.2015 10:06 3 -> /proc/15885/fd/
XXXXXXXXXXXXXXXXX # 0 이 모두 X 로 바뀜
111111111111111111
total X
lr-x----- 1 mug896 mug896 64 X8.X3.2X15 1X:X6 X -> /home/mug896/tmp/3/x
l-wx----- 1 mug896 mug896 64 X8.X3.2X15 1X:X6 1 -> /home/mug896/tmp/3/x
lrwx----- 1 mug896 mug896 64 X8.X3.2X15 1X:X6 2 -> /dev/pts/8
lr-x----- 1 mug896 mug896 64 X8.X3.2X15 1X:X6 3 -> /proc/15885/fd/
```

이번에는 명령 실행전에 파일 x 를 삭제합니다. 파일 x 가 먼저 삭제되었기 때문에 ls 명령의 출력이 파일 제일 위에 위치 하게됩니다. 그리고 여기서 눈여겨 볼점은 sed 명령에서 입력으로 사용되는 데이터가 ls 명령의 출력이 아닌 이전 x 파일의 내용 이라는 점입니다.

```
$ { rm -f x; { ls -l /dev/fd/; sed -e 's/0/X/g' ;} >> x ;} < x

$ cat x
total 0
lr-x----- 1 mug896 mug896 64 08.03.2015 10:08 0 -> /home/mug896/tmp/3/x (deleted)
l-wx----- 1 mug896 mug896 64 08.03.2015 10:08 1 -> /home/mug896/tmp/3/x
lrwx----- 1 mug896 mug896 64 08.03.2015 10:08 2 -> /dev/pts/8
lr-x----- 1 mug896 mug896 64 08.03.2015 10:08 3 -> /proc/15951/fd/
XXXXXXXXXXXXXXXXX
111111111111111111
```

그러므로 다음과 같이 하면 파일 x 의 내용을 sed 명령으로 바꾸어 저장할 수 있습니다. 결론적으로 `sed -i 's/0/X/g' x` 한 결과와 같습지만 이 방법은 `sed -i` 와 달리 실행중에 임시파일을 만들지 않습니다. 그러므로 사이즈가 큰 파일을 처리중에 문제가 발생하여 프로세스가 중단된다면 처리되지 않은 나머지 뒷부분은 모두 잃게 됩니다.

```
$ { rm -f x; sed -e 's/0/X/g' > x ;} < x

$ cat x
XXXXXXXXXXXXXXXXX
1111111111111111
```

Redirection 우선순위

`{ ;}` 은 명령 grouping 외에 [메타문자 우선순위](#) 조절에도 사용됩니다. redirection 기호도 shell 메타문자 중에 하나로 `{ ;}` 를 이용하면 우선순위를 조절할 수 있습니다. 아래와 같은 명령이 실행될 경우 제일 바깥쪽 메타문자가 먼저 실행되고 차례로 안쪽에 있는 메타문자가 실행됩니다.

```
$ { command 2번 ;} 1번
```

```
# FD 3 번이 생성되어있지 않아 오류 발생.
$ date >&3
bash: 3: Bad file descriptor

$ date >&3 3>&1
bash: 3: Bad file descriptor

# 3>&1 가 먼저 실행되므로 FD 3 번이 생성되어 오류가 발생하지 않는다.
$ { date >&3 ;} 3>&1
Sat Aug 8 03:31:05 KST 2015
```

앞선 예제 `{ rm -f x; sed -e 's/0/X/g' > x ;} < x` 에서 파일 x 가 삭제되었음에도 불구하고 기존의 데이터를 읽어들이 수 있었던 것도 외부에 있는 `< x` 가 먼저 실행되어 open 되었기 때문입니다. 사실 내부에 있는 x 는 삭제되어 다시 생성된 파일로 외부 x 와는 inode 번호가 다릅니다.

```
$ stat -c %i x; { rm -f x; sed -e 's/0/X/g' > x ;} < x; stat -c %i x
1709909
1709910
```

만약에 두 메타문자의 위치를 바꾸어 실행한다면 rm 명령으로 삭제된 파일을 입력으로 사용하게 되므로 오류가 발생합니다.

```
$ { rm -f x; sed -e 's/0/X/g' < x ;} > x
bash: x: No such file or directory
```

함수를 정의할때 redirection 사용

함수를 정의할때 redirection 을 사용하면 함께 정의가 됩니다. 다음 color 함수는 명령 실행결과 stderr 로 출력되는 메시지를 stdout 으로 출력되는 메시지와 분류하여 빨간색으로 표시합니다.

```
color() (
    set -o pipefail;
    "$@" 2>&1 1>&3 | sed 's/./\e[31m&\e[m/' >&2
) 3>&1
```

1. 함수를 정의할때 { ;} 대신에 () subshell 을 사용한것은
pipefail 옵션설정을 함수에 국한하기 위해서 입니다.
2. "\$@" 는 color 함수명 뒤에 오는 스트링을 명령문으로 실행합니다.
3. 2>&1 : stderr 가 stdout 으로 redirect 됐으니 파이프를 통해 sed 입력으로 들어갑니다.
1>&3 : stdout 는 fd 3번 (함수밖에서 제일먼저 stdout 으로 연결됐죠) 으로 연결되므로
sed 입력으로 들어가지 않고 바로 stdout 으로 출력됩니다.
4. sed 입력으로 들어간 stderr 메시지는 color escape 문자처리가 되어 출력됩니다.
'\e' escape 문자적용을 위해 '\$' ' quote 이 사용되었습니다.
& 문자는 앞에 매칭된 라인으로 치환됩니다.

테스트 해보기

```
f1() {
    echo 111;
    echo AAA >&2;
}

$ color f1
111      # stdout 은 흰색으로 출력
AAA      # stderr 는 빨간색으로 출력

$ color date -%Y
date: invalid option -- '%'      # 빨간색으로 출력
```

문제점 : 명령의 출력이 "정상메시지; 에러메시지; 정상메시지; 에러메시지;" 순서로 출력될경우 파이프라인으로 인해 메시지 순서가 유지되지 않고 "정상메시지; 정상메시지; 에러메시지; 에러메시지;" 로 출력

Quiz

FD 1번 stdout 과 FD 2번 stderr 를 서로 바꾸기

```

# 초기 FD 상태
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 0 -> /dev/pts/10 # stdin
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 1 -> /dev/pts/10 # stdout
lrwx----- 1 mug896 mug896 64 07.06.2015 15:04 2 -> /dev/pts/10 # stderr

# 먼저 FD 1 을 FD 3 으로 백업
$ exec 3>&1

0 -> /dev/pts/10 # stdin
1 -> /dev/pts/10 # stdout
2 -> /dev/pts/10 # stderr
3 -> /dev/pts/10 # stdout 생성

# FD 1 을 FD 2 에 연결
$ exec 1>&2

0 -> /dev/pts/10 # stdin
1 -> /dev/pts/10 # stderr 변경
2 -> /dev/pts/10 # stderr
3 -> /dev/pts/10 # stdout

# FD 2 을 백업해둔 FD 3 에 연결
$ exec 2>&3

0 -> /dev/pts/10 # stdin
1 -> /dev/pts/10 # stderr
2 -> /dev/pts/10 # stdout 변경
3 -> /dev/pts/10 # stdout

# FD 3 삭제
$ exec 3>&-

0 -> /dev/pts/10 # stdin
1 -> /dev/pts/10 # stderr
2 -> /dev/pts/10 # stdout

##### 테스트 #####

$ ( exec 3>&1 1>&2 2>&3 3>&-; date ) 2> out

$ cat out
Sat Aug  1 15:51:45 KST 2015

$ ( exec 3>&1 1>&2 2>&3 3>&-; date -%Y ) 1> out

$ cat out
date: invalid option -- '%'
Try 'date --help' for more information.

```

Pipe

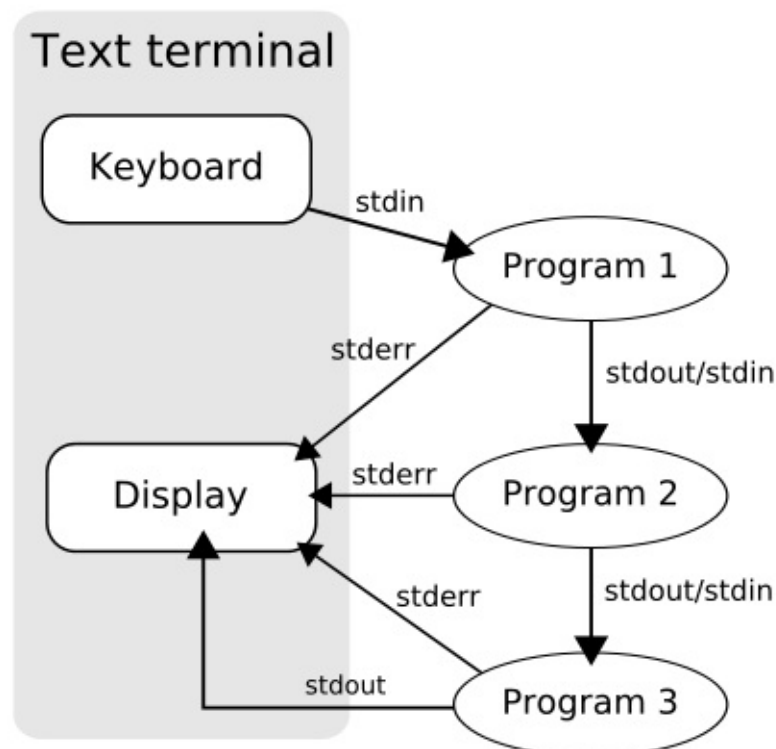
프롬프트 명령행 상에서 모듈 프로그래밍을 할 수 있습니다. 바로 파이프를 이용하는 것 인데요. 가령 : 문자로 구분된 필드를 가진 inventory.db 라는 파일을 읽어들이 3번째 필드를 선택해서 그 중 첫자가 c 로 시작하는 항목들만 뽑아내어 알파벳 순서로 정렬하여 프린트한다면 일반적인 프로그래밍 언어로도 그렇게 간단한 작업이 아닐 수 있습니다. 하지만 명령행상에서 파이프를 이용한다면 다음과 같이 간단하게 처리할 수 있습니다.

```
cat inventory.db | cut -d ':' -f 3 | grep '^c' | sort
```

먼저 필요한 명령들을 고르고, 적절한 옵션을 준 후에 파이프를 연결을 하면 프로그래밍을 한 것과 같이 훌륭하게 결과를 만들어 냅니다. 유닉스의 기본철학을 보통 모듈화 라고 합니다. 다시말해 "각기 독립적인 역할을 하는 프로그램을 만들어놓고 필요에 따라 선택해서 서로 조합하여 전체를 완성한다" 는것은 바로 이 파이프라는 기능이 있기에 가능하다고 할 수 있습니다.

파이프의 stdin, stdout, stderr

아래 도식은 파이프를 연결된 명령들이 실행될때 표준입력 (stdin), 표준출력 (stdout) , 표준에러 (stderr) 의 관계를 보여주는데 program1 의 표준출력 은 program2 의 표준입력이 되고 program2 의 출력은 program3 의 입력이 되며 마지막으로 program3 출력이 터미널로 디스플레이 됩니다. 기본적으로 표준에러는 모두 터미널로 연결돼 있는 것을 볼 수 있습니다.



파이프로 연결된 명령은 subshell 에서 실행된다.

아래는 프롬프트 상에서 `{ echo; sleep 10 ;} | { echo; sleep 10 ;} | { echo; sleep 10 ;}` 명령을 실행했을때의 프로세스 상태인데 파이프로 연결된 세 명령 모두 subshell 이 생성된후에 그 아래에서 실행되는것을 볼수있습니다. 그러므로 마지막 명령에서 결과를 어떤변수에 할당한다면 그값은 파이프 실행이 종료되면 사라지게 됩니다.

| PID | TTY | STAT | TIME | COMMAND |
|-------|--------|------|------|-------------|
| 27160 | pts/13 | Ss | 0:00 | bash |
| 2479 | pts/13 | S+ | 0:00 | _ bash |
| 2486 | pts/13 | S+ | 0:00 | _ sleep 10 |
| 2480 | pts/13 | S+ | 0:00 | _ bash |
| 2484 | pts/13 | S+ | 0:00 | _ sleep 10 |
| 2483 | pts/13 | S+ | 0:00 | _ bash |
| 2485 | pts/13 | S+ | 0:00 | _ sleep 10 |

`shopt -s lastpipe` 옵션을 설정하면 마지막 명령이 현재 shell 에서 실행되게 할수있습니다. 이 옵션은 job control 이 disable 되어있어야 사용할수 있는데 non-interactive shell 인 script 실행시에는 기본적으로 disable 됩니다.

파이프로 연결된 shell 의 FD

다음은 파이프로 연결된 shell 의 FD (File Descriptor) 상태입니다. 현재 shell pid 를 나타내는

`$$` 변수는 subshell 에서도 동일한 값을 가지므로 subshell 에서의 FD 상태를 보기 위해서는

`$BASHPID` 변수를 이용해야 합니다. 첫번째 명령은 `$$` 변수를 사용했으므로 현재 shell 의 FD 상태에 해당하고 나머지는 파이프에 연결된 명령 순서대로 입니다. 첫번째 명령은 stdout 이 파이프에 연결돼 있고, 두번째 가운데에 위치한 명령은 stdin, stdout 둘다 연결되어 있습니다. 마지막 명령은 stdin 에 파이프가 연결돼 있는 걸 볼 수 있습니다.

```

E$ { ls -l /proc/$$/fd >&2 ;} | { : ;} | { : ;}
total 0
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 0 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 00:32 255 -> /dev/pts/13

E$ { ls -l /proc/$BASHPID/fd >&2 ;} | { : ;} | { : ;}
total 0
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 0 -> /dev/pts/13
l-wx----- 1 mug896 mug896 64 04.07.2015 21:08 1 -> pipe:[1026629]
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 255 -> /dev/pts/13

E$ { : ;} | { ls -l /proc/$BASHPID/fd >&2 ;} | { : ;}
total 0
lr-x----- 1 mug896 mug896 64 04.07.2015 21:08 0 -> pipe:[1026640]
l-wx----- 1 mug896 mug896 64 04.07.2015 21:08 1 -> pipe:[1026642]
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 255 -> /dev/pts/13

E$ { : ;} | { : ;} | { ls -l /proc/$BASHPID/fd >&2 ;}
total 0
lr-x----- 1 mug896 mug896 64 04.07.2015 21:08 0 -> pipe:[1026653]
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 1 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 2 -> /dev/pts/13
lrwx----- 1 mug896 mug896 64 04.07.2015 21:08 255 -> /dev/pts/13

```

명령이 파이프에서 실행되는지 구분하기

다음은 grep 명령이 파이프에서 실행될 경우 옵션설정을 달리하여 실행하기 위한 함수입니다.

```

# 함수이름이 우선순위가 높으므로 외부명령 지정을 위해 command 명령을 사용해야 합니다.
# /dev/fd 는 /proc/self/fd 의 심볼릭 링크로 process 자신을 나타냅니다.

```

```

grep() {
    # FD 1번 (stdout) 이 터미널에 연결되어 있는지 테스트
    if [ -t 1 ]; then
        command grep -n "$@"

    # stdout 이 (named, unnamed) pipe 에 연결되어 있는지 테스트
    elif [ -p /dev/fd/1 ]; then
        command grep "$@"

    # stdout 이 일반 파일에 연결되어 있는지 테스트
    elif [ -f /dev/fd/1 ]; then
        command grep ...

    fi
}

```


파이프로 연결된 명령들은 순서대로 실행될까?

보통 파이프에 대해 설명할때 `command1 | command2` 가 있을 경우 `command1` 의 실행결과가 `command2` 의 입력으로 들어간다고 말합니다. 그렇다면 `command1` 이 종료된 후에 `command2` 가 실행될것 같지만 실은 그렇지 않습니다. 다음을 한번 보시죠

```
$ ps | grep ".*"
PID TTY          TIME CMD
3773 pts/0        00:00:00 bash
3784 pts/0        00:00:00 ps
3785 pts/0        00:00:00 grep
```

만약에 `ps` 실행이 종료되고 그 결과가 `grep` 명령의 입력으로 들어간다면 최종 결과에는 `grep` 명령 이 보이면 안되겠지만 `ps` 와 `grep` 명령이 동시에 나타나고 있습니다. 파이프로 연결된 프로그램들이 실행될때는 순서대로 실행되지 않고 모두 동시에 실행됩니다. 또한 `command2` 의 상태에 따라 `command1` 의 실행이 완료되기 전에 종료할 수도 있습니다.

다음과 같은 경우 `grep` 명령의 실행이 완료되기 전에 결과가 나오는 것을 볼 수 있습니다.

```
grep pattern very-large-file | tr a-z A-Z
```

다음과 같은 경우는 `head` 명령의 실행이 먼저 종료됨에 따라 `grep` 명령이 실행을 완료하기 전에 종료하게 됩니다.

```
grep pattern very-large-file | head -n 1
```

파이프로 연결된 명령들의 종료값은?

파이프로 연결된 명령들 간에는 `$?` 변수로 이전 명령의 종료값을 확인할 수 없습니다. 그리고 파이프 실행이 종료됐을 때의 종료값은 마지막 명령의 종료값이 사용됩니다. 그래서 중간에 `false` 로 종료된 명령이 있더라도 마지막 명령이 `true` 로 종료되면 파이프 종료값은 `true` 가 됩니다. `set -o pipefail` 옵션을 설정하면 중간에 `false` 로 종료된 명령이 있을 경우 파이프 종료값은 `false` 가 됩니다. `shell` 변수로 `PIPESTATUS` 라는 `array` 변수가 있는데 이변수는 파이프로 연결된 모든 명령들의 종료값을 담고 있습니다.

파이프로 연결된 명령들은 process group 을 형성한다.

파이프를 이용해 여러 명령을 동시에 실행시키면 `process group` 이 만들어지는데 이때 파이프로 연결된 명령들 중에 첫번째 명령이 `process group id (pgid)` 가 됩니다. 이후 `jpspec` 을 이용하여 `job control` 을 하게되면 동일한 `process group` 에 속한 명령들이 모두 같이 적용을 받게 됩니다.

파이프에서 복수의 명령 사용하기

파이프로 명령을 연결할때 꼭 하나의 명령만 사용할 수 있는것은 아닙니다.

```
# 다음의 경우는 첫번째 명령의 결과를 중간의 date 명령이 전달하지 못하고 있습니다.
```

```
$ echo "What date is it today?" | date | cat
```

```
Mon Jul 27 10:51:34 KST 2015
```

```
$ echo "What date is it today?" | { cat ; date ;} | cat
```

```
What date is it today?
```

```
Mon Jul 27 11:01:57 KST 2015
```

```
$ echo "What date is it today?" | { date; cat ;} | cat
```

```
Mon Jul 27 11:02:05 KST 2015
```

```
What date is it today?
```

Named Pipe

| 파이프를 이용해 명령들을 연결하여 사용하거나 process 치환을 사용하면 명령 실행 중에 pipe 가 자동으로 생성되어 사용된 후 사라지게 되는데요. 이때 생성되는 파이프를 이름이 없다고 해서 unnamed pipe 또는 anonymous pipe 라고 합니다. 이에 반해 named pipe 는 직접 파이프를 파일로 만들어 사용합니다.

named pipe 는 파일과 동일하게 사용될수 있는데 파일과 다른 점은 redirection 을 이용해 데이터를 출력했을 때 파일은 데이터를 저장하는 반면 pipe 는 저장하지 않는다는 점입니다. 그래서 만약에 디스크 용량이 부족한 상태에서 용량이 큰 파일을 다루고자 할때 pipe 를 이용하면 프로세스 중간에 임시파일을 만들지 않아도 되므로 디스크 사용을 피할 수 있습니다. 다음은 gzip 으로 압축돼 있는 mysql 데이터 파일을 압축 해제하여 mypipe 로 출력하고 mysql 프롬프트 상에서 named pipe 를 이용해 테이블에 로드 하는 예입니다.

```
$ mkfifo /tmp/mypipe
$ gzip --stdout -d dbfile.gz > /tmp/mypipe

# 다음은 mysql 프롬프트 상에서 실행하는 명령입니다.
mysql> LOAD DATA INFILE '/tmp/mypipe' INTO TABLE tableName;
```

또한가지 pipe 는 데이터를 저장하지 않기 때문에 파일 내용을 random access 할 수 없습니다. 그러므로 위에서처럼 파일을 open 한 후에는 처음부터 끝까지 한번에 읽거나 써야 합니다.

다음은 named pipe 를 이용해서 일종의 프록시를 만드는 예인데요. 첫번째 nc 명령은 localhost 8080 포트를 리스닝하고 있다가 브라우저가 접속하면 받은 request 를 그대로 stdout 으로 출력합니다. 두번째 nc 는 www.naver.com 80 에 연결한 상태입니다. 두 명령은 | 로 연결돼 있으므로 브라우저의 request 가 naver 에 전달되게 됩니다. 이때 naver 에서 받은 결과는 두번째 nc 명령의 stdout 으로 출력되는데 지금은 > mypipe 로 연결돼 있습니다. 그러므로 naver 에서 받은 결과는 > mypipe 로 보내지고 다시 첫번째 nc 명령의 < mypipe 를 통해 입력으로 들어가서 브라우저까지 도달하게 됩니다.

```
$ mkfifo mypipe
$ nc -l localhost 8080 < mypipe | nc www.naver.com 80 > mypipe
```

이렇게 named pipe 를 이용하면 기존의 | 파이프를 통해서 하지 못하는 작업들을 할수있습니다.

Pipe 는 FIFO

Pipe 는 FIFO (First In First Out) 형식으로 데이터가 전달됩니다. 그러니까 제일 먼저 pipe 로 들어간 데이터가 pipe 를 읽게 되면 제일 먼저 값으로 나오게 됩니다.

```
$ mkfifo mypipe
# FD 를 연결하면 버퍼가 차기 전까지 block 되지 않는다.
$ exec 3<> mypipe

$ echo 111 > mypipe
$ echo 222 > mypipe
$ echo 333 > mypipe

$ read var < mypipe; echo $var
111
$ read var < mypipe; echo $var
222
$ read var < mypipe; echo $var
333

$ exec 3>&-
```

Pipe 는 block 된다.

위의 FIFO 예에서 데이터를 mypipe 로 입력할때 `exec 3<> mypipe` 를 사용한걸 볼 수 있습니다. 이것은 pipe 로 데이터를 전달할 때 데이터를 읽는 상대방이 없으면 block 되기 때문입니다. 앞서 이야기했지만 pipe 는 파일과 달리 데이터를 저장하지 않기 때문에 읽는 상대방이 없으면 작업이 중단됩니다. 또한 writer 가 없는 상태에서 읽기를 시도할 때도 block 됩니다. 다시 말해 pipe 는 writer 와 reader 가 서로 연결되어 있어야 작업이 진행될수 있습니다.

Broken pipe 에러

writer 와 reader 가 서로 pipe 에 연결되어 writer 가 지속적으로 데이터를 쓰고 있는 상태에서 reader 가 종료 하게 되면 Broken pipe 에러로 writer 가 종료됩니다.

```
# writer 쓰기시작, 하지만 reader 가 읽기 전까지 block 된다.
$ while ;; do echo $(( i++ )); sleep 1; done > mypipe &
[1] 17103

$ cat mypipe # reader 읽기 시작
0
1
2
3
4
^C          # ctrl-c 로 강제 종료

$          # Broken pipe 에러로 writer 종료됨
[1]+  Broken pipe          while ;; do
    echo $(( i++ )); sleep 1;
done > mypipe
```

Reader 의 자동 연결 해제

writer 와 reader 가 서로 pipe 에 연결되어 reader 가 지속적으로 데이터를 읽고 있는 상태에서 writer 가 종료하면 에러는 발생하지 않지만 읽을 데이터가 없는 reader 는 파이프로부터 연결이 해제됩니다.

```
# terminal 1
# writer 시작. 현재 reader 가 없기 때문에 block
$ while ;; do echo $(( i++ )); sleep 1; done > mypipe

# terminal 2
$ cat mypipe # reader 실행
0
1
2
3
...

# terminal 1
# 현재 실행되고 있는 writer 를 ctrl-c 로 강제 종료

# terminal 2
# reader 는 읽을 데이터가 없으므로 파이프 연결이 해제되고 다음 프롬프트가 뜬다.
```

FD 를 PIPE 에 연결해서 사용

위에서 살펴본 바와 같이 파이프는 상대방의 연결 상태에 따라 broken pipe 오류로 writer 가 종료되거나 아니면 읽을 데이터가 없는 reader 는 파이프에서 자동으로 연결이 해제됩니다. 이와 같은 파이프의 특성은 파이프에서 읽어들이 데이터가 없더라도 지속적으로 연결을 유지하고 있다가 데

이터가 들어올 경우 처리하고자 할때나 또는 writer 가 지속적으로 데이터를 쓰고 있는 상태에서 reader 가 일시적으로 연결을 해제하고자 할때 장애가 됩니다. 이와 같은 경우에 FD 를 pipe 에 연결하여 사용하면 문제를 해결할 수 있습니다.

```
# FD 3 번을 mypipe 에 연결
# named pipe 는 외부에서 모두 사용할 수 있는 파일이기 때문에
# 이 명령은 아무 프로세스에서 한번만 실행하면 됩니다.
# 그리고 설정한 프로세스가 종료하면 연결도 해제됩니다.

$ exec 3<> mypipe
```

writer 가 지속적으로 쓰고 있는 상태에서 reader 를 강제 종료하기

```
$ while ;; do echo $(( i++ )); sleep 1; done > mypipe &
[1] 21951

$ cat mypipe
0
1
2
3
4
^C    # ctrl-c 강제 종료

# reader 를 강제 종료 시켰지만 writer 는 broken pipe 로 종료되지 않는다.
# 그리고 다시 파이프를 읽으면 다음 데이터가 이어진다.

$ cat mypipe
5
6
7
8
^C
```

reader 가 지속적으로 읽고 있는 상태에서 writer 를 종료하기

```

# terminal 1
$ while ;; do echo $(( i++ )); sleep 1; done > mypipe # writer 실행

# terminal 2
$ cat mypipe # reader 실행
0
1
2
3
...

# terminal 1
# ctrl-c 로 writer 강제 종료

# terminal 2
# writer 가 종료되어 읽어들이 데이터가 없지만 파이프에 연결을 지속하고 있음.

# terminal 1
# 다시 writer 연결
$ while ;; do echo $(( i++ )); sleep 1; done > mypipe

# terminal 2
# writer 가 다시 연결하여 쓴 데이터가 연이어서 읽혀진다.
4
5
6
7
...

```

Pipe buffer size

Reader 가 없는 상태에서 writer 가 파이프에 쓰기를 하면 커널에서 사용하는 pipe buffer size 를 알아볼 수 있습니다. `ulimit -a` 명령으로 조회해 보면 pipe size 가 $8 * 512 \text{ bytes} = 4096$ 로 나오지만 커널에서 16 개까지 버퍼를 할당해 사용하므로 $4096 * 16 = 65,536$ 까지 사용할 수 있게 됩니다. 여기서 4096 은 리눅스에서 여러 프로세스가 동시에 같은 파일에 write 할때 데이터가 서로 겹치지 않고 atomic 하게 쓸 수 있는 크기에 해당합니다.

```

/usr/src/linux-headers-3.19.0-25/include/linux/pipe_fs_i.h
#define PIPE_DEF_BUFFERS 16

```

```

$ mkfifo mypipe
$ exec 3<> mypipe

# ctrl-c 로 종료
$ while ;; do echo -n '1'; done > mypipe
^Cbash: echo: write error: Interrupted system call

# outfile 로 버퍼 내용을 출력후 ctrl-c 로 종료
$ cat mypipe > outfile
^C

# outfile 파일 사이즈가 65,536 으로 나옴
$ ls -l outfile
-rw-rw-r-- 1 mug896 mug896 65536 08.03.2015 15:33 outfile

```

Multiple writers and readers

Pipe 는 데이터 손실 없이 multiple writers, readers 를 가질 수 있습니다. 하지만 입, 출력시에 데이터 분배는 일정하지 않습니다. atomic 하게 write 할수있는 크기는 ulimit 명령의 pipe size 로 알 수 있습니다. 다음은 multiple writers, single reader 의 예입니다.

```

$ while ;; do echo $(( i++ )); sleep 1; done > mypipe &
[1] 23084

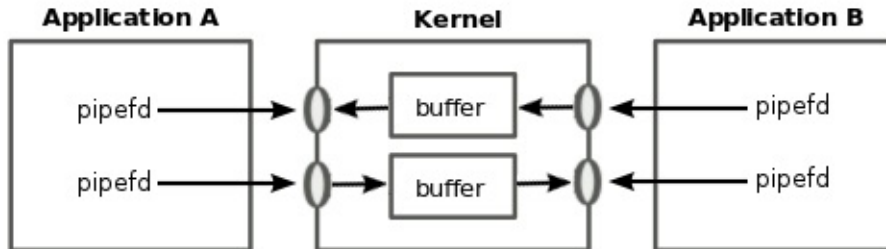
$ while ;; do echo --- $(( i++ )); sleep 1; done > mypipe &
[2] 23097

$ cat mypipe
0
1
2
--- 0
3
--- 1
4
--- 2
5
--- 3
6
--- 4
7
--- 5
...

```

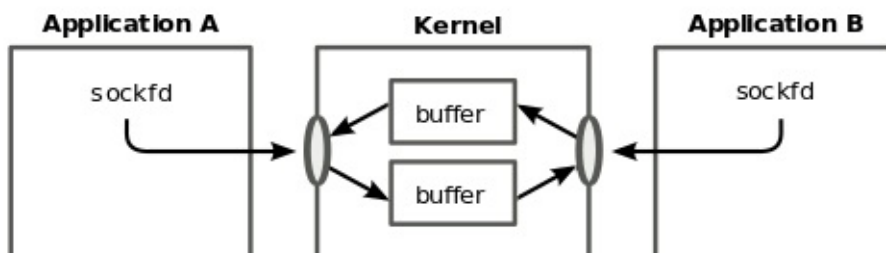
양방향 통신

Pipe 는 단방향이기 때문에 두 프로세스가 서로 대화를 나누기 위해서는 두개의 파이프가 필요합니다. 이와같은 방법을 이용하는 것이 keyword 명령중에 하나인 **coproc** 입니다. coproc 는 입, 출력 용으로 두개의 파이프를 만들어서 입력 파이프를 통해 외부 프로세스로부터 입력을 받고 연산결과를 출력 파이프를 통해 전달합니다.



Socket

파이프의 양방향 통신과 비슷한 것이 unix domain socket 입니다. unix domain socket 은 파이프와 같이 디렉토리에 **socket** 파일을 만들어서 시스템 내의 프로세스와 통신을 하는데 이때 생성되는 **socket** 파일은 다른 프로세스가 접속할때 사용하는 ip 주소와 같은 역할을 합니다. 커널이 교통정리를 해주므로 socket 에 연결된 하나의 FD 만으로도 양방향 통신을 할수있습니다.



터미널에서 **nc** 명령을 이용하여 직접 테스트해볼 수 있습니다. 먼저 터미널에 입, 출력 메시지가 함께 표시되므로 2개의 터미널이 필요합니다. **terminal 1** 에서 **nc -lU mysocket** 명령을 실행하면 디렉토리에 **mysocket** 파일이 생성되고 접속대기 상태가 됩니다. 이어 **terminal 2** 에서 **nc -U mysocket** 명령을 실행하면 두 프로세스가 연결되는데 이후에 서로 메시지를 주고받을 수 있습니다.

- terminal 1

```
$ nc -lU mysocket
hello
socket
```

- terminal 2

```
$ nc -U mysocket
hello
socket
```

두 프로세스가 연결된 후 FD 상태

```
E$ ls -l /proc/27509/fd/
total 0
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 0 -> /dev/pts/11
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 1 -> /dev/pts/11
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 2 -> /dev/pts/11
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 4 -> socket:[5879565]

E$ ls -l /proc/27510/fd/
total 0
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 0 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 1 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 2 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 08.20.2015 18:19 3 -> socket:[5885903]
```

unix domain socket 을 이용하는 대표적인 프로그램 중에 하나가 xwindows 서버 인데요

/tmp/.X11-unix/ 디렉토리에 socket 파일이 위치합니다. 또한 `lsuf -u` 명령을 실행하면 시스템 내에서 unix domain socket 을 사용하는 프로그램들을 볼수있습니다. (pipe 를 이용하는 프로그램은 `lsuf | awk '$5 == "FIFO"'` 로 볼수있습니다.)

unix domain socket 은 파일시스템에 socket 파일을 만들어 통신하기 때문에 같은 컴퓨터에서 실행되는 프로세스들에 한해서만 통신이 가능하다는 단점이 있습니다. internet domain socket 은 서로 떨어져 있는 컴퓨터에서 실행되는 프로세스들과도 통신을 할수있습니다. 이때는 데이터를 상자에 담아 송, 수신자 주소를 적어 보내는 packet 과 protocol 개념이 필요하게 됩니다.

`nc` 명령을 이용해 internet domain socket 도 테스트해볼 수 있습니다. 먼저 접속을 받는 computer1 에서 `ifconfig` 명령으로 ip 주소를 확인한 다음 `nc -l 12.34.56.78 8080` 명령을 실행하면 접속대기 상태가 됩니다. 이어 computer2 에서 computer1 의 ip 주소와 포트번호를 이용해 `nc 12.34.56.78 8080` 명령을 실행하면 두 프로세스가 연결되는데 이후부터 서로 메시지를 주고받을 수 있습니다.

- computer1

```
$ nc -l 12.34.56.78 8080
hello
internet socket
```

- computer2

```
$ nc 12.34.56.78 8080
hello
internet socket
```

named pipe 사용예)

Job Control 메뉴에 보면 wait 명령과 파일을 이용해 background process 들의 exit 값을 구하는 예가 있습니다. 그걸 named pipe 를 이용해 바꾼 것인데요. 여기서 눈여겨볼 점은 FD 를 named pipe 에 연결해 사용하지 않으면 `while read -r res` 문에서 읽을 데이터가 없을경우 바로 종료가 됩니다.

```
#!/bin/bash

trap 'rm -f $pipe_name' EXIT

pipe_name="/tmp/mypipe_$$"
mkfifo "$pipe_name"
exec {FD}<> "$pipe_name" # FD 를 named pipe 에 연결

doCalculations() {
    echo start job $i...
    sleep $((RANDOM % 5))
    echo ...end job $i
    exit $((RANDOM % 10))
}

number_of_jobs=10

for i in $( seq 1 $number_of_jobs )
do
    ( trap "echo job$i ----- exit: \ $? > $pipe_name" EXIT
      doCalculations ) &
done

i=1
while read -r res; do
    echo "$res"
    [ $i -eq $number_of_jobs ] && break
    let i++
done < "$pipe_name"

echo $i jobs done !!!
```

File Descriptors

프로그램이 실행중에 데이터 입, 출력에 사용하는 파일이나 `pipe`, `socket`, 외부장치 같은 리소스들은 처음 사용될때 OS 에 의해 리소스 번호가 할당됩니다. 이 번호는 양의 정수로 각 프로세스마다 가지고 있는 `file descriptor table` 에 등록되고 이후부터는 이 FD 번호를 이용해 리소스를 사용하게 됩니다.

실제 프로그래밍에서 FD 는 `low-level` 적인 개념으로 FD 자체가 어떤 기능을 하는것은 아니지만 `shell` 에서는 FD 를 파일에 연결하면 파일포지션을 이용할수 있고 `pipe` 나 `socket` 의 경우에는 읽어들일 데이터가 없어도 연결을 유지할 수 있으며 버퍼를 사용할 수 있습니다.

그리고 `parent process` 에 설정되어있는 FD 들은 `child process` 에게 상속됩니다.

File descriptor 의 생성, 복사, 삭제

터미널을 열면 자동으로 생성되는 FD 0, 1, 2 외에 필요하면 다른 번호의 FD 를 생성, 복사, 삭제할수 있습니다. 이와같은 작업은 `exec builtin` 명령으로 하며 사용 가능한 번호의 범위는 `ulimit -n` 과 같습니다.

FD 를 설정할때 사용하는 메타문자

- 입력 : `<`
- 출력 : `>`
- 입, 출력 : `<>`
- append : `>>`

이 메타문자의 방향성은 실제 리소스에 연결할 때만 의미를 갖습니다. 이후에 FD 끼리 서로 복사하거나 삭제할 때는 무시할 수 있습니다 (방향에 상관없이 원본 FD 의 설정이 그대로 복사되기 때문에). FD 설정을 할때 FD 와 리소스를 위치시키는 방법은 다음과 같습니다.

새로 생성, 복사, 삭제되는 FD 는 왼쪽에, 실제 리소스, 원본 FD, `-` 는 오른쪽에

그러므로 FD 3 번을 생성하면서 `infile` 에 입력으로 연결할 경우

```
# 왼쪽 : 새로 생성되는 FD, 오른쪽 : 리소스
# 실제 리소스에 연결하므로 방향을 올바르게 설정해야 합니다.

$ exec 3< infile
```

FD 4 번을 생성하고 FD 0 번에 연결하기

```
# 왼쪽 : 복사되는 FD, 오른쪽 : 원본 FD
# 실제 리소스와의 설정이 아니므로 방향성을 무시할수 있습니다.
# ( 다시말해 exec 4>&0 로 할수도 있습니다. )

$ exec 4<&0
```

FD 3 번을 삭제하기

```
# 왼쪽 : 삭제되는 FD, 오른쪽 : '-'
# 실제 리소스와의 설정이 아니므로 방향성을 무시할수 있습니다.
# ( 다시말해 exec 3>&- 로 할수도 있습니다. )

$ exec 3<&-
```

FD 3 번을 만들고 outfile 파일을 출력으로 연결

출력이므로 이때 outfile 파일에 내용이 있다면 삭제됩니다. 삭제를 방지하고 append 되게 하려면 `>>` 을 사용합니다. `>` 는 데이터를 파일의 처음부터 쓰기 시작하며, `>>` 는 기존의 내용에 뒤이어서 쓰게 됩니다.

```
$ exec 3> outfile

# write 으로 연결 하였으므로 퍼미션은 `l-wx-----` 가 됩니다.
$ ls -l /proc/$$/fd/3
l-wx----- 1 mug896 mug896 64 04.07.2015 10:56 /proc/9363/fd/3 -> /home/mug896/tmp/outfil

$ echo hello FD >&3

$ cat outfile
hello FD
```

FD 3 번을 사용후 삭제하기

```
$ exec 3>&-

$ ls -l /proc/$$/fd/3
ls: cannot access /proc/9363/fd/3: No such file or directory
```

FD 4 번을 만들고 infile 파일을 입력으로 연결

```
$ exec 4< infile

# read 로 연결 하였으므로 퍼미션은 `lr-x-----` 가 됩니다.
$ ls -l /proc/$$/fd/4
lr-x----- 1 mug896 mug896 64 04.07.2015 10:56 /proc/9363/fd/4 -> /home/mug896/tmp/infile
```

파일에서 라인을 읽어 들일때 FD 를 사용하는것과 단순히 < 를 사용하는것은 차이가 있습니다.

```
$ cat infile
111
222
333

$ read var < infile; echo $var
111
$ read var < infile; echo $var      # 읽을때 마다 새로 실행되는 명령과 같아 동일한 값이 출력된다
111

$ exec 4< infile                    # FD 4 번을 infile 파일에 입력으로 연결.

$ read var <&4; echo $var             # FD 이므로 한번 읽으면 안의 파일 포지션이 이동한다.
111
$ read var <&4; echo $var
222
$ read var <&4; echo $var
333
```

FD 4 번을 사용후 삭제하기

```
exec 4<&-
```

FD 5 번을 만들고 outfile 파일을 입, 출력 으로 연결

```
$ exec 5<> iofile

# read, write 로 연결 하였으므로 퍼미션은 `lrwx-----` 가 됩니다.
$ ls -l /proc/$$/fd/4
lrwx----- 1 mug896 mug896 64 04.07.2015 10:56 /proc/9363/fd/5 -> /home/mug896/tmp/iofile

$ cat <&5
hello world

$ echo this is a test >&5

$ cat iofile
hello world
this is a test

$ exec 5>&-
```

<> 로 파일을 연결하게 되면 기존의 데이터는 삭제되지 않습니다. 하지만 쓰기작업은 기존의 데이터를 **overwrite** 하면서 첫라인 부터 쓰게됩니다. 또한 읽기와 쓰기시에 파일 포지션 값을 공유하므로 유의해야 합니다.

```
$ cat iofile
111
222
333
444
555

$ exec 3<> iofile

$ echo XXX >&3      # 기존의 데이터는 삭제되지 않지만 첫라인부터 overwrite 된다.
$ cat iofile
XXX
222
333
444
555

$ read var <&3; echo $var  # 쓰고난후 파일 포지션이 이동하여 222 값이 출력된다.
222
$ read var <&3; echo $var
333

$ echo YYY >&3      # 읽고난후 파일 포지션이 이동하여 444 자리에 overwrite 된다.
$ cat iofile
XXX
222
333
YYY
555
```

<> 는 읽거나 쓰게되면 계속해서 파일 포지션이 아래로 이동합니다. 그러므로 윗부분의 데이터를 다시 읽으려면 `exec` 으로 다시 FD 를 연결해야 합니다. 하나의 파일에 두개의 FD 를 연결하면 각각 독립적으로 파일 포지션을 사용할 수 있습니다.

pipe 나 socket 의 연결을 유지할 수 있다.

Shell 에서 FD 를 pipe 나 socket 에 연결하여 사용하면 읽어들일 데이터가 없더라도 연결을 유지할 수 있습니다.

바로 `socket` 에 메시지를 보내면 연결이 유지되지 않는다.

```
$ echo hello > /dev/tcp/www.google.com/80

$ ls -al /proc/$$/fd
total 0
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 0 -> /dev/pts/18
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 1 -> /dev/pts/18
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 2 -> /dev/pts/18
```


FD 를 사용하면 socket 의 연결이 유지된다.

```
$ exec 3<> /dev/tcp/www.google.com/80

$ ls -al /proc/$$/fd
total 0
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 0 -> /dev/pts/18
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 1 -> /dev/pts/18
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 2 -> /dev/pts/18
lrwx----- 1 mug896 mug896 64 08.19.2015 18:37 3 -> socket:[5641570]

$ echo hello >&3

$ cat <&3
HTTP/1.0 400 Bad Request
Content-Type: text/html; charset=UTF-8
Content-Length: 1419
Date: Wed, 19 Aug 2015 09:36:39 GMT
Server: GFE/2.0
...
```

버퍼를 사용할 수 있다.

```
$ mkfifo mypipe

$ echo hello > mypipe      # reader 가 없으므로 block 된다.
^C                        # Ctrl-c 종료

$ exec 3<> mypipe          # FD 를 연결하면 버퍼가 사용되므로 block 되지 않는다.
$ echo hello > mypipe      # ( 버퍼가 차기 전까지 )
$
```

Named file descriptor

FD 번호를 생성할때 현재 어떤 번호가 사용되고 있는지 체크할 필요 없이 자동으로 FD 번호를 생성해서 변수에 할당해 주는 기능입니다. 생성할때와 삭제할때 변수 이름에 `{ }` 를 사용해야 합니다.

```
$ exec {myFD}> outfile

$ echo $myFD
10

$ ls -l /proc/$$/fd/$myFD
l-wx----- 1 mug896 mug896 64 04.07.2015 10:56 /proc/9363/fd/10 -> /home/mug896/tmp/outfi

$ exec {myFD}>&-

$ ls -l /proc/$$/fd/$myFD
ls: cannot access /proc/9363/fd/10: No such file or directory
```

FD 는 child process 에게 상속된다.

shell script 가 실행되어 child process 가 생성될때 parent 에 설정되어있는 FD 를 물려받습니다. 그러므로 parent 와 동일하게 터미널로부터 입력을 받고, 출력을 할 수 있습니다.

```
$ exec 3> outfile

$ ls -l /proc/$$/fd
total 0
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 0 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 1 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 2 -> /dev/pts/14
l-wx----- 1 mug896 mug896 64 Jul  5 09:45 3 -> /home/mug896/tmp/outfile

$ bash -c 'ls -l /proc/$$/fd' # child process 생성
total 0
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 0 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 1 -> /dev/pts/14
lrwx----- 1 mug896 mug896 64 Jul  5 09:45 2 -> /dev/pts/14
l-wx----- 1 mug896 mug896 64 Jul  5 09:45 3 -> /home/mug896/tmp/outfile
```

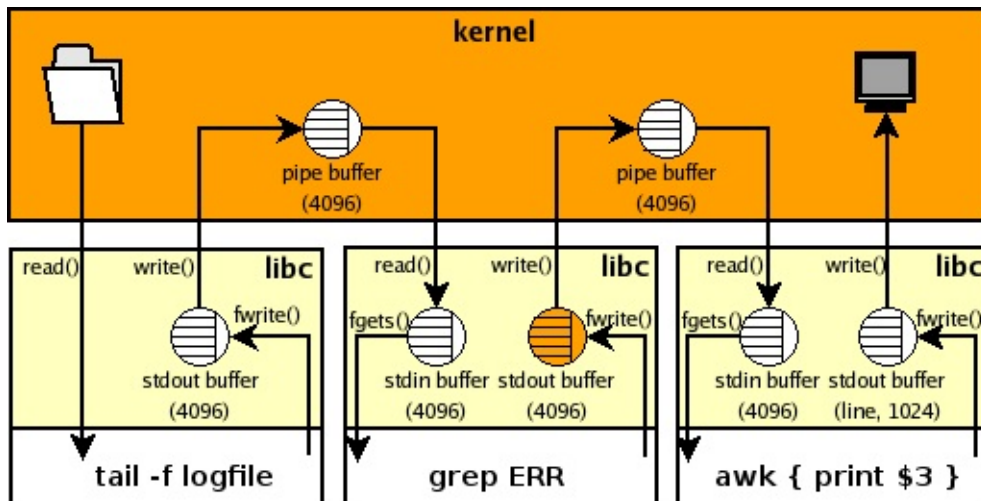
Buffering

Buffering 은 shell 에서 명령을 사용할때 뿐만 아니라 java, python 같은 언어에서 입,출력을 할 때 도 동일하게 적용되는 개념입니다. stream 으로 데이터를 입력받고 연산결과를 출력하는 명령들은 내부적으로 버퍼를 이용합니다 (grep, sed, awk ...). 명령 실행이 바로 종료되면 버퍼에 있던 내용 도 모두 출력되므로 문제가 없지만 프로세스가 종료되지 않은 상태에서 출력을 지속한다면 버퍼와 관련해서 문제가 생길 수 있습니다.

다음은 프로세스 A 가 logfile 에 로그를 append 하면 tail 명령으로 실시간으로 데이터를 추출해서 프린트하는 명령인데 실행해보면 정상적으로 동작하지 않습니다. 프로세스 A 가 ERR 로그를 logfile 에 append 했으므로 파일에는 로그가 존재함에도 불구하고 grep, awk 명령을 거치면서 출 력이 되지 않고 있는 것입니다. 나중에 로그가 쌓여서 출력할 데이터가 4096 bytes 가되면 그때 한 번에 출력이 됩니다.

```
$ tail -f logfile | grep ERR | awk '{ print $3 }'
```

이와 같은 상황을 그림으로 나타내면 다음과 같습니다. 먼저 kernel 쪽 버퍼는 kernel 에서 사용하 는 것으로 사용자가 컨트롤할 수 있는 부분이 아니고 버퍼링과 관련해서 문제가 되지는 않는다고 합니다. 사용자가 컨트롤할 수 있는 부분은 프로그램이 사용하는 버퍼인데 tail 명령에서는 -f 옵션을 주어 출력이 버퍼링 되지 않지만 grep 명령에서 출력이 버퍼링 되고 있는 것을 볼 수 있습니 다.



[출처] http://www.pixelbeat.org/programming/stdio_buffering/

Buffering modes

Buffering modes 는 다음 3 가지로 구분해 볼 수 있습니다.

- **line buffered** : newline 을 만나면 출력합니다.
- **full buffered** : 버퍼가 차면 출력합니다.
- **unbuffered** : 버퍼링을 하지 않고 바로 출력합니다.

명령이 실행되면 자동으로 `stdin`, `stdout`, `stderr` 세 개의 `stream` 이 생성되는데 버퍼와 관련해서 각각 다음과 같은 특징이 있습니다. (기본적으로 새로 open 되는 `stream` 은 `full buffered` 입니다.)

- **stdin** : 터미널에서 입력을 받으면 `line buffered` or `unbuffered` 이고 그외는 `full buffered` 입니다.
- **stdout** : 터미널에 연결되어 있으면 `line buffered` 이고 그외는 `full buffered` 입니다.
- **stderr** : 항상 `unbuffered` 입니다. 그러므로 `stderr` 로 메시지를 출력하면 바로 표시가 됩니다.

위 예제의 경우를 적용해보면 `tail` 명령은 `-f` 옵션을 주었으므로 `unbuffered` 에 해당되고 `grep` 명령은 출력이 파이프에 연결되어 있으므로 `full buffered` 가되며 `awk` 는 터미널에 연결되어 있으므로 `line buffered` 가 됩니다. 그러므로 문제를 해결하기 위해서는 `grep` 명령의 출력 버퍼링을 수정해 주어야 합니다.

문제 해결 하기

버퍼를 이용하는 대부분의 명령들은 버퍼링으로 인해 발생할 수 있는 문제를 해결하기 위해 별도의 옵션이나, 함수를 제공합니다. 위에서 예들든 `grep`, `sed`, `awk` 같은 명령 외에도 `python`, `java`, `perl` 같은 인터프리터 언어들도 모두 동일하게 적용됩니다. 이런 언어들은 보통 내부적으로 버퍼링을 컨트롤할 수 있는 함수를 제공합니다.

- `grep` : `--line-buffered`
- `sed` : `-u,--unbuffered`
- `tail` : `-f`
- `tcpdump` : `-l`
- `gawk` : `fflush()`, `close()`
- `perl` : `flush()`, `close()`, `$|`
- `python` : `-u` , `flush()`, `close()`
- `java` : `flush()`, `close()`

그러므로 위의 문제를 해결하기 위해서는 다음과 같이 작성하면 됩니다.

```
$ tail -f logfile | grep --line-buffered ERR | awk '{ print $3 }'
```

버퍼와 관련된 옵션을 제공하지 않는 경우

모든 명령이 버퍼와 관련된 옵션이나 함수를 제공하는 것은 아닙니다. 다음의 경우 `cut`, `tr` 명령은 별도로 버퍼를 컨트롤할 수 있는 옵션을 제공하지 않아 문제를 해결할 수 없습니다.

```
$ tail -f access.log | cut -d ' ' -f1 | tr A-Z a-z | uniq
```

이럴 때에는 `gnu coreutils` 명령 중에 하나인 `stdbuf` 명령을 이용하여 해결할 수 있습니다.

```
$ tail -f access.log | stdbuf -oL cut -d ' ' -f1 | stdbuf -oL tr A-Z a-z | uniq
```

stdbuf

Usage: `stdbuf OPTION... COMMAND`

- Options

- `-i`, `--input=MODE` : stdin stream buffering 을 조정
- `-o`, `--output=MODE` : stdout stream buffering 을 조정
- `-e`, `--error=MODE` : stderr stream buffering 을 조정

- Modes

mode **L** : line buffered

mode **O** : unbuffered

full buffered 는 KB 1000, K 1024, MB 1000*1000, M 1024*1024 ... G, T, P, E, Z, Y 단위를 이용해서 버퍼 사이즈를 설정할 수 있습니다.

`tee` 명령같이 자체에서 버퍼링을 설정하는 경우는 `stdbuf` 설정이 override 되고, `dd`, `cat` 명령같이 i/o 에 stream 을 사용하지 않는 경우는 적용되지 않습니다.

sed 명령의 출력 버퍼링 문제

다음은 `coprocess` 를 사용하는 예인데 `sed` 명령이 background 로 실행을 지속하면서 stdout 이 파이프에 연결되어 출력이 버퍼링 되고 있습니다. 그러므로 `read` 명령으로 연산 결과를 읽을 수가 없습니다.

```
$ coproc sed 's/^/foo/'
[1] 5907

$ echo bar >& ${COPROC[1]}

$ read -r res <& ${COPROC[0]}
^C
```

다음과 같이 `-u` | `--unbuffered` 옵션을 사용하여 해결할 수 있습니다.

```
$ coproc sed --unbuffered 's/^/foo/'
[1] 5993

$ echo bar >& ${COPROC[1]}

$ read -r res <& ${COPROC[0]}

$ echo $res
foobar
```

sed 명령의 입력 버퍼링 문제

`sed 1q` 는 입력받은 데이터에서 첫번째 라인을 출력하고 종료하는 명령입니다. `echo` 명령으로 3개의 라인을 파이프를 통해 전달하였으므로 각 `sed` 명령에서 한 라인씩 프린트할것 같지만 실제 결과는 첫번째 라인만 출력되고 있습니다. 원인은 첫번째 `sed` 명령의 input 버퍼로 3개의 라인이 모두 저장되었기 때문입니다.

```
$ echo -e "ONE\nTWO\nTHREE\n" | { sed 1q ; sed 1q ; sed 1q ; }
ONE
```

다음과 같이 `-u` | `--unbuffered` 옵션을 사용하면 해결할 수 있습니다.

```
# 첫번째 sed 명령에 -u 옵션사용
$ echo -e "ONE\nTWO\nTHREE\n" | { sed -u 1q ; sed 1q ; sed 1q ; }
ONE
TWO

# 첫번째, 두번째 sed 명령에 -u 옵션사용
$ echo -e "ONE\nTWO\nTHREE\n" | { sed -u 1q ; sed -u 1q ; sed 1q ; }
ONE
TWO
THREE
```

sort 명령 주의할점

`sort` 명령은 결과를 sort 하기 위해서 모든 출력을 buffering 합니다. 그러므로 명령 실행이 바로 종료되지 않는 경우에는 사용할 수 없습니다.

Buffer sizes

Default Buffer sizes

- `ulimit -p (8 * 512 = 4096)`

- stdin, stdout 이 터미널에 연결되어 있으면 default size = 1024, 그외는 4096

버퍼가 자동으로 flush 되는 경우

- 버퍼가 full 되었을때
- stream 이 close 될때
- 프로그램이 종료될때
- line buffered 모드에서 newline 을 만났을때
- input stream 에서 파일로부터 데이터를 읽어들이때

왜 버퍼링을 사용하나?

상대적으로 속도가 느린 장치에 접근 횟수를 줄일 수 있습니다.

가령 디스크에 있는 데이터를 읽어들이 처리한다고 할때 4096 번 접속해서 읽어들이는 것보다는 한번에 4096 bytes 를 메모리로 읽어들이 후에 메모리에서 읽어들이 처리하는게 효율적이고 속도가 빠릅니다. 데이터를 쓸때도 마찬가지로 속도가 느린 디스크에 4096 번 접속해서 쓰는것보다는 메모리에 4096 bytes 데이터가 찼을때 한번에 디스크에 쓰는것이 효율적입니다. 그리고 실질적으로 1 byte 를 쓰는 속도나 4096 bytes 를 쓰는 속도는 차이가 없다고 합니다.

처리 속도가 다른 프로세스 간에 block 되는 횟수를 줄일 수 있습니다.

두 프로세스 간에 데이터를 주고 받을때 한쪽이 처리속도가 느리면 다른쪽 프로세스는 더이상 진행하지 못하고 block 됩니다. 하지만 버퍼를 사용하면 이렇게 block 되는 횟수를 줄일 수 있습니다.

System call 횟수를 줄일 수 있습니다.

외부 장치에서 데이터를 입, 출력할 때나 프로세스 간에 데이터를 주고받을 때는 사용자 프로그램이 단독으로 할수없고 kernel 에서 제공하는 system call 함수를 이용해야 합니다. system call 을 하면 사용자 프로그램이 실행되는 user mode 에서 kernel mode 로 context 가 switching 되는데 이것 또한 cpu 연산을 소요하는 작업입니다. 버퍼를 이용하면 4096 번 system call 을 하는 대신에 1번으로 횟수를 줄일수 있습니다.

<<< , <<

Here string (<<<) , here document (<<) 는 실행 중에 임시 파일을 만들어 사용합니다. 그러므로 `echo` 명령 대신에 `cat` 명령을 사용합니다. 아래는 `bash -c` 명령의 인수로 here document 를 사용하였는데 stdin 입력으로 sh-thd-4029673544 임시 파일이 사용된 후 삭제된 것을 볼수가 있습니다.

```
$ bash -c 'ls -l /proc/$$/fd' <<EOF          # here document
hello
EOF

total 0
lr-x----- 1 mug896 mug896 64 Jul  5 00:08 0 -> /tmp/sh-thd-4029673544 (deleted)
lrwx----- 1 mug896 mug896 64 Jul  5 00:08 1 -> /dev/pts/11
lrwx----- 1 mug896 mug896 64 Jul  5 00:08 2 -> /dev/pts/11

-----

$ bash -c 'ls -l /proc/$$/fd' <<< "hello"      # here string
total 0
lr-x----- 1 mug896 mug896 64 Jul  5 00:20 0 -> /tmp/sh-thd-4029677356 (deleted)
lrwx----- 1 mug896 mug896 64 Jul  5 00:20 1 -> /dev/pts/11
lrwx----- 1 mug896 mug896 64 Jul  5 00:20 2 -> /dev/pts/11
```

<<< (here string)

스트링을 명령의 입력으로 사용하려고 할때 파이프를 사용하면 subshell 로 인해 설정된 변수의 값을 유지할 수가 없습니다. 이때 here string 을 이용하면 현재 shell 에서 실행되어 문제를 해결할 수 있습니다.

```
# read 명령이 '|' 로인해 subshell 에서 실행되어 값이 표시되지 않는다
$ echo "here string test" | read var
$ echo $var

# '<<<' 는 현재 shell 에서 실행되어 정상적으로 값을출력.
$ read var <<< "here string test"
$ echo $var
here string test
```

<< (here document)

Here string 과 같은 역할을 하지만 구분자를 이용해 여러줄을 입력할 수 있습니다. 구분자명은 임의로 만들어 사용할 수 있으며 마지막 줄은 구분자 외에 다른 문자가 와서는 안됩니다. 본문에서 escape 할 수 있는 문자는 double quote 과 같으며 (단 " 는 제외, 구분자를 사용하므로), 구분자에 single quote 을 하면 변수확장, 명령치환이 일어나지 않습니다.

```
# 기본적으로 본문에서 변수확장, 명령치환이 일어난다.
$ AA=100
$ cat <<ABC      # 구분자 ABC
> here $AA
> document $(date +%D)
> ABC

here 100
document 07/23/15

# 구분자에 single quote 을 하면 변수확장, 명령치환이 안된다.
$ AA=100
$ cat <<'END'    # 구분자 END
> here $AA
> document $(date +%D)
> END

here $AA
document $(date +%D)
```

Here document 를 파일로 쓰기

```
$ cat <<END > outfile
> here
> document
> END
```

Here document 를 파이프로 전달하기

```
$ cat <<END | tr a-z A-Z
here
document
END

HERE
DOCUMENT
```

Here document 를 변수에 넣기

```
$ AA=$(cat <<END
> here
> document
> END           # 마지막 줄은 구분자 외에 다른 문자가 오면 안된다.
> )

$ echo "$AA"
here
document
```

while 문의 입력으로 사용하기

```
$ while read -r line; do
>     echo "$line"
> done <<END
> here
> document
> END
here
document
```

Leading tab 을 출력에서 제거하기

here document 기호로 `<<-` 를 이용하면 출력시에 leading tab 을 제거할 수 있습니다.

```
$ if true; then
>     cat <<END
>     here
>     document
>     with tab
> END
> fi
    here          # 출력에 leading tab 이 포함됨
    document
    with tab

# '<<' 기호 대신에 '<<-' 를 사용

$ if true; then
>     cat <<-END
>     here
>     document
>     without tab
> END
> fi
here          # 출력에서 leading tab 이 제거됨
document
without tab
```

Job Control

터미널에서 단순히 한번에 하나의 명령만 실행시킬 수 있는 것이 아니라 `&` 메타문자를 이용해 `background job` 을 생성함으로써 멀티태스킹을 할 수 있습니다. 가령 인터넷에서 파일을 다운로드 하는 `job` 을 `background` 로 실행시켜놓고 동시에 `vi` 에디터로 파일 수정 작업을 할 수 있습니다.

`background process` 는 `main process` 와 독립적으로 실행되기 때문에 실행중인 스크립트를 `Ctrl-c` 로 종료하여도 `background` 로 실행을 계속합니다. 그러므로 종료상태 값을 알 수 없는 단점이 있습니다.

job id 와 job specification

```
$ wget -o wget.log http://fly.srk.fer.hr/jpg/flyweb.jpg &  
[3] 26558
```

명령을 `&` 메타문자를 이용해 `background` 로 실행시키면 결과로 `job id` 와 `process id` 를 보여줍니다. 위의 예에서 `[3]` 부분이 `job id` 에 해당되고 `26558` 은 `process id (pid)` 에 해당됩니다.

만약에 `kill` 명령을 사용해 `job` 에 신호를 보낼때 `job id` 를 사용한다면 `pid` 와 구분할 수 없게 됩니다 (둘 다 숫자이므로). 그래서 `job id` 대신 `job specification (줄여서 jobspec)` 을 사용하는데 이때 `jobspec` 은 `job id` 앞에 `%` 문자를 붙여서 만듭니다. (예: `%3`)

이 `jobspec` 은 `job control` 에 사용되는 명령들 (`jobs`, `bg`, `fg`, `wait`, `disown`, `kill`) 에서 사용됩니다. `jobspec` 을 이용해 `kill` 명령으로 신호를 보내면 같은 `pgid (process group id)` 를 갖는 프로세스들에게 모두 전달되므로 만약에 `child process` 가 생성되어 실행 중이라면 함께 종료하게 됩니다.

jobspec 과 pid 가 다른점

`pid` 는 개별 프로세스를 나타내지만 `jobspec` 은 파이프로 연결된 모든 프로세스를 포함합니다.

```
$ sleep 10 | sleep 10 | sleep 10 &
[1] 12782

# jobspec 은 파이프로 연결된 3 개의 프로세스를 모두 포함.
$ jobs
[1]+  Running                  sleep 10 | sleep 10 | sleep 10 &

# pid 는 개별 프로세스를 나타냄.
$ ps af
  PID TTY          STAT       TIME COMMAND
...
 1643 pts/10    Ss        0:00 bash
12780 pts/10    S         0:00 \_ sleep 10 # pid
12781 pts/10    S         0:00 \_ sleep 10 # pid
12782 pts/10    S         0:00 \_ sleep 10 # pid
...
```

jobs

`jobs [-lnprs] [jobspec ...] or jobs -x command [args]`

현재 job table 목록을 보여줍니다. `-l` 옵션을 주면 process id 도 함께 보여줍니다. job id 옆에 보이는 `+`, `-` 기호는 jobspec 에 사용되며 `%+` 는 current job 그러니까 가장 최근에 background 상태가된 job 을 나타내고 `%-` 는 previous job 을 나타냅니다. `fg` 명령을 사용해 이 동함에 따라 `+`, `-` 위치도 바뀌게 됩니다.

```
$ jobs                                     # vi 로 3 개의 파일을 열고난 후의 상태
[1]  Stopped                  vi 111
[2]- Stopped                  vi 222 # previous job
[3]+ Stopped                  vi 333 # current job ( 가장 최근 job )

$ fg %1                                    # %1 로 이동

$ jobs
[1]+ Stopped                  vi 111 # current
[2]  Stopped                  vi 222
[3]- Stopped                  vi 333 # previous

$ fg %2                                    # %2 로 이동

$ jobs
[1]- Stopped                  vi 111 # previous
[2]+ Stopped                  vi 222 # current
[3]  Stopped                  vi 333
```

jobspec `%%` 은 `%+` 와 동일한 의미를 가집니다.

fg

```
fg [jobspec]
```

해당 jobspec 을 foreground 로 실행하고 current job 으로 만듭니다.

그러므로 이후에 ctrl-z 로 stopped 되었을때 job table 에는 + 로 표시됩니다.

jobspec 을 인수로 주지 않으면 current job (+ 표시된 job) 이 사용됩니다.

bg

```
bg [jobspec ...]
```

현재 stopped 상태에 있는 jobs 을 running 상태로 만듭니다.

jobspec 을 인수로 주지 않으면 current job 이 사용됩니다.

suspend

```
suspend [-f]
```

suspend 명령을 실행하는 shell 은 이후에 SIGCONT 신호를 받기 전까지 중단됩니다.

login shell 에서는 사용할 수 없으나 -f 옵션을 이용하면 override 할 수 있습니다.

disown

```
disown [-h] [-ar] [jobspec ...]
```

disown 명령은 단어에서 알 수 있듯이 "이 job 은 내것이 아니다" 라고 선언합니다. job 이 종료되는 것은 아니고 결과로 job table 목록에서 삭제되어 더이상 control 할 수 없게 됩니다. 터미널 프로그램 종료시킬때, 또는 login shell 에서 exit 시에 HUP 시그널에 의해 job 이 종료되는 것을 방지할 수 있습니다 (shopt -s huponexit 옵션이 설정되어 있을 경우). -h 옵션도 동일한 역할을 하지만 job table 에는 계속 남아있으므로 control 할 수 있습니다.

wait

```
wait [-n] [jobspec or pid ...]
```

background 로 실행되는 job 이 종료될 때까지 기다립니다. 인수로 jobspec 이나 pid 를 주게되면 해당 job, pid 가 종료될 때까지 기다린 후 종료값을 리턴합니다. 인수 없이 실행하면 모든 프로세스를 기다리며 종료 값으로는 0 을 리턴합니다. 특정 작업이 완료된후에 다음단계로 진행해야 할경우 사용할수 있습니다.

```
$! 변수는 가장 최근에 background 로 실행된 pid 값을 나타냅니다.
```

```
#!/bin/bash

( sleep 1; exit 3 ) &

wait $!
echo $?

##### output #####

3

-----
#!/bin/bash

( echo start process 1...; sleep 4; echo end process 1.; exit 1 ) &
( echo start process 2...; sleep 3; echo end process 2.; exit 2 ) &
( echo start process 3...; sleep 5; echo end process 3.; exit 3 ) &

wait                                     # 3 개의 background process 를 모두 기다림.
echo exit status: $?

##### output #####

start process 1...
start process 2...
start process 3...
end process 2.
end process 1.
end process 3.
exit status: 0                          # 종료 상태값이 0 이 됨
```

위의 예에서 보는 것과 같이 여러개의 background job 을 생성하면 각 프로세스의 종료 상태 값을 알수없습니다. 그럴땐 다음과 같은 방법을 사용할 수 있습니다.

```
#!/bin/bash

trap 'rm -f $tmpfile' EXIT

tmpfile=$(mktemp)

doCalculations() {
    echo start job $i...
    sleep $((RANDOM % 5))
    echo ...end job $i
    exit $((RANDOM % 10))
}

number_of_jobs=10

for i in $( seq 1 $number_of_jobs )
do
    ( trap "echo job$i : exit value : \ $? >> $tmpfile" EXIT
      doCalculations ) &
done

wait

i=0
while read -r res; do
    echo "$res"
    let i++
done < "$tmpfile"

echo $i jobs done !!!
```

Job control 관련 키조합

- **Ctrl-c**

interrupt 신호 (SIGINT) 를 foreground job 에 보내 종료시킵니다.

- **Ctrl-z**

suspend 신호 (SIGTSTP) 를 foreground job 에 보내 suspend 시키고 background 에 있던 shell 프로세스를 foreground 로 하여 명령을 입력받을 수 있게 합니다.

Input and Output

- **Input**

입력은 foreground job 에서만 받을 수 있습니다. background job 에서 입력을 받게되면 SIGTTIN 신호가 전달되어 suspend 됩니다.

- **Output**

출력은 기본적으로 현재 session 에서 실행되고 있는 모든 job 들이 공유합니다. 그러므로 background job 을 실행할때 제대로 redirection 처리를 하지 않으면 터미널로 출력되는 메시지들이 서로 섞이게 됩니다.

`stty tostop` 명령을 사용하면 background job 에서 출력이 발생할시 SIGTTOU 신호가 전달되어 suspend 시킬 수 있습니다.

Shell 이 종료되면 background job 들은 어떻게 될까?

프롬프트 상에서 exit 이나 logout 명령으로 종료할 경우

background job 은 두가지 상태를 가집니다. stopped 와 running 인데요. shell 을 exit 할때 stopped 상태에 있는 job 이 있으면 메시지를 통해 프롬프트 상에 알려줍니다. 하지만 stopped job 을 처리하지 않고 다시 exit 명령을 하게되면 shell 이 종료되는데 이때 stopped job 도 함께 종료됩니다.

running 상태에 있는 job 은 기본적으로 shell 이 종료되어도 background 로 실행을 계속합니다. 그러나 바뀌는게 하나 있는데요, 바로 parent process id (ppid) 입니다. background job 들은 shell 에서 실행이 됐기 때문에 ppid 가 shell 이 되는데요. shell 이 종료가 됐기때문에 ppid 가 1 번 그러니까 init process (or upstart) 로 바뀌게 됩니다. 이와같은 동작은 script 실행 시에도 동일하게 적용됩니다.

login shell 일 경우 `shopt -s huponexit` 옵션을 설정하게 되면 logout 시에 모든 running background job 들이 HUP 신호를 받고 종료하게 됩니다.

윈도우 상에서 터미널 프로그램을 종료시키거나 시스템이 종료될 경우

session leader 인 shell process 에 `kill -HUP` 신호를 주는 경우도 해당되며 이때는 shell 의 자손 프로세스들이 모두 HUP 신호를 받고 종료합니다.

Ctrl-c 로 스크립트 실행을 종료시키는 경우

exit 명령으로 shell 이 종료될 때와 같이 ppid 가 바뀌어 실행을 계속합니다.

Controlling Terminal

[TTY](#) 메뉴 참조.

Double forks

터미널에서 실행시킨 background 프로세스가 job table 에 잡히지 않는 경우가 있습니다. 스크립트 A.sh 에서 B.sh 을 background 로 실행 시킨후 먼저 종료하게되면 B.sh 은 orphaned process 가 되고 ppid 가 init (or upstart) 로 바뀌어 실행됩니다. 이때 B.sh 은 controlling terminal 에서 떨어져 나오게되므로 터미널이 종료되어도 HUP 신호를 받지 않습니다. init 프로세스는 child process 가 종료되었을때 좀비 프로세스가 생기지 않도록 항상 wait system call 을 실행하므로 이와같은 절차는 daemon 프로세스를 생성하는 방법으로도 사용됩니다.

```

----- A.sh -----
#!/bin/bash
echo A.sh ... start

./B.sh &

echo A.sh ... end

----- B.sh -----
#!/bin/bash
echo B.sh ... start

ls -l /dev/fd/

echo B.sh ... end

##### 실행결과 #####

$ ./A.sh
A.sh ... start
A.sh ... end
B.sh ... start
total 0
lr-x----- 1 mug896 mug896 64 Aug 12 13:37 0 -> /dev/null # stdin 이 /dev/null
lrwx----- 1 mug896 mug896 64 Aug 12 13:37 1 -> /dev/pts/14 # 에 연결되어 실행
lrwx----- 1 mug896 mug896 64 Aug 12 13:37 2 -> /dev/pts/14
B.sh ... end

```

Script 실행시에는 job control 이 기본적으로 disable 됩니다.

non-interactive shell 인 script 실행시에는 기본적으로 job control 이 disable 됩니다. 그렇다고 해서 & 메타문자를 이용해 background process 를 생성하지 못한다는건 아니고 다만 bg, fg, suspend 명령을 사용할 수 없습니다. 하지만 jobs, wait, disown 명령들은 사용할 수 있습니다. 필요에 따라 set -o monitor 옵션 설정을 통해 enable 할수도 있습니다.

Session 과 Process group

리눅스 에서 실행되는 모든 프로세스들을 process id (pid) 와 parent process id (ppid) 관계로만 관리하기엔 부족한점이 있는데요. 그래서 session 과 process group 을 만들어 사용합니다. 예를 들어 터미널을 열면 shell 이 실행되는 이때 shell pid 가 session id (sid) 가 되며 session leader 가 됩니다. 이후 프롬프트를 통해 명령을 실행시켜 생성되는 자손 process 들은 같은 sid 를 갖게됩니다.

shell script 가 실행되면 process group 이 만들어 지는데 이때 script pid 가 process group id (pgid) 가 되며 process group leader 가 됩니다. 이후 script 내에서 실행되는 process 들은 모두 같은 pgid 를 갖게됩니다. 다음은 프롬프트 상에서 AA.sh script 를 실행한 예입니다. 모두 bash shell pid 를 sid 로 가지고 있고 AA.sh pid 가 pgid 로 사용된 것을 볼 수 있습니다.

| PPID | PID | PGID | SID | COMMAND |
|-------|-------|-------|-------|----------|
| 9111 | 29471 | 29471 | 29471 | bash |
| 29471 | 4606 | 4606 | 29471 | _ AA.sh |
| 4606 | 4607 | 4606 | 29471 | _ BB.sh |
| 4607 | 4608 | 4606 | 29471 | _ BB.sh |
| 4608 | 4610 | 4606 | 29471 | _ sleep |
| 4607 | 4609 | 4606 | 29471 | _ BB.sh |
| 4609 | 4611 | 4606 | 29471 | _ sleep |

'|' 파이프를 통해 여러 명령을 동시에 실행시킬 때도 process group 이 만들어지는데 이때는 파이프로 연결된 명령들중에 첫번째 명령이 pgid 와 process group leader 가 됩니다. (script 내에서 실행될 경우는 script pgid 를 따릅니다.) 다음은 프롬프트 상에서 { echo; sleep 10 ;} | { echo; sleep 10 ;} 명령을 실행한 예입니다. 모두 bash shell pid 를 sid 로 가지고 있고 파이프로 연결된 명령 중에 첫번째 명령의 pid 가 pgid 로 사용된 것을 볼 수 있습니다.

| PPID | PID | PGID | SID | COMMAND |
|-------|-------|-------|-------|----------|
| 9111 | 29471 | 29471 | 29471 | bash |
| 29471 | 4631 | 4631 | 29471 | _ bash |
| 4631 | 4634 | 4631 | 29471 | _ sleep |
| 29471 | 4632 | 4631 | 29471 | _ bash |
| 4632 | 4633 | 4631 | 29471 | _ sleep |

이와같이 process group 은 job control 을할때 기본단위 (job) 가 됩니다. 하나의 session 에서는 하나의 process group 만 foreground 가 될수있고 나머지는 background 가 됩니다.

실행되는 모든 프로세스는 (단 하나의 프로세스라도) 프로세스 그룹에 속하게되고, 또한 프로세스 그룹은 세션에 속하게 됩니다.

조회하고 신호보내기

현재 실행되고 있는 process 들의 ppid, pid, pgid, sid 관계는 다음과 같이 ps 명령을 통해 알아볼 수 있고 pgrep , pkill 명령을 사용하면 process group 이나 session 별로 조회하거나 시그널을 보낼 수 있습니다.

```
$ ps axfo user,ppid,pid,pgid,sid,comm
```

```
$ ps axjf
```

실행중인 스크립트를 종료하는 방법

스크립트를 종료할 때는 jobspec 을 이용하거나 pgid 를 이용해 process group 에 신호를 보내야 합니다. 또는 Ctrl-c 키를 누르는 것도 process group 에 신호를 보내는 방법 중에 하나입니다. 그렇지 않고 스크립트 pid 에만 신호를 보내면 child process 는 종료되지 않고 남아 있게 됩니다. 다음은 ping 외부 명령을 실행하고 있는 스크립트 pid 에 종료신호를 보냈으나 child process 인 ping 명령은 종료되지 않은채 남아 있습니다.

```
----- test.sh -----
#!/bin/bash
echo test.sh ... start

ping 192.168.1.1

echo test.sh ... end
-----

# test.sh 스크립트 실행후 ps 조회
$ ps j -C test.sh
  PID  PGID   SID TTY          TIME CMD
13056 13056 12676 pts/16   00:00:00 test.sh

# test.sh 스크립트 pid 를 이용해 종료
$ kill 13056
[1]+  Terminated                  ./test.sh

# child process 인 ping 명령은 종료되지 않고 ppid 가 upstart 로 바뀌어 남아있다.
$ ps jf -C ping
  UID      PID  PPID  PGID   SID  C  STIME TTY          TIME CMD
mug896   13057  1091 13056 12676   0  10:39 pts/16   00:00:00 ping 192.168.1.1
```

다음과 같이 pgid 를 이용해 스크립트를 종료합니다.

- pkill -g 13056
- kill -- -13056 (pgid 앞에 - 문자를 붙인다.)

```
$ ps j -C test.sh
  PID  PGID   SID TTY          TIME CMD
13056 13056 12676 pts/16    00:00:00 test.sh

# test.sh 스크립트 pgid 를 이용해 종료
$ pkill -g 13056
[1]+  Terminated                  ./test.sh

# 같은 pgid 를 갖는 ping 명령도 함께 종료되었다.
$ ps jf -C ping
UID          PID  PPID  PGID   SID  C STIME TTY          TIME CMD
```

새로운 session id 로 실행하기

스크립트를 background 로 실행시킬때 `setsid` 명령을 이용하면 새로운 sid, pgid 가 할당되고 ppid 도 init (or upstart) 으로 바뀌어 실행됩니다. sid 가 바뀌므로 controlling terminal 에서도 떨어져 나가고 parent 도 init 이 되므로 스크립트를 daemon 으로 실행시키는 결과를 갖습니다.

```
setsid daemon.sh > /dev/null 2>&1 < /dev/null &
```

Process State Codes

`ps ax` 명령을 실행하면 STAT 컬럼에 현재 프로세스의 상태를 나타내는 기호들이 표시됩니다. 아래 첫번째 테이블의 상태 기호가 제일 앞에 오고 그다음에 두번째 테이블의 추가적인 기호가 표시됩니다.

| 기호 | 설명 |
|----------|--|
| D | uninterruptible sleep (usually IO) |
| R | running or runnable (on run queue) |
| S | interruptible sleep (waiting for an event to complete) |
| T | stopped, either by a job control signal or because it is being traced. |
| X | dead (should never be seen) |
| Z | defunct ("zombie") process, terminated but not reaped by its parent. |

다음은 BSD 포맷의 추가적인 정보를 나타냅니다.

| 기호 | 설명 |
|-------------|---|
| < | high-priority (not nice to other users) |
| N | low-priority (nice to other users) |
| L | has pages locked into memory (for real-time and custom IO) |
| s | is a session leader |
| I | is multi-threaded (using CLONE_THREAD, like NPTL pthreads do) |
| + | is in the foreground process group. |

다음은 현재 실행중인 linux OS 의 ps 정보를 발췌한 것입니다.

| TTY | STAT | TIME | COMMAND |
|--------|------|-------|---|
| pts/14 | Ss+ | 0:00 | bash |
| pts/13 | Ss | 0:00 | bash |
| pts/13 | R+ | 0:00 | _ ps af |
| pts/1 | Ss+ | 0:01 | bash |
| pts/1 | T | 0:00 | _ /bin/bash ./test.sh |
| pts/1 | Z | 0:00 | _ [sub2.sh] <defunct> |
| pts/8 | Ss+ | 0:00 | bash |
| pts/7 | Ss+ | 0:00 | bash |
| pts/6 | Ss | 0:00 | bash |
| tty7 | Ssl+ | 36:11 | /usr/bin/X -core :0 -seat seat0 -auth /var/run/ |
| ? | S<l | 20:01 | /usr/bin/pulseaudio --start --log-target=syslog |
| ? | SNsl | 0:05 | /usr/lib/rtkit/rtkit-daemon |
| ? | SLl | 47:34 | /home/mug896/S.opt/google/chrome/chrome |
| ? | D | 0:24 | [iprt-VBoxTscThr] |

- **S** : 대부분의 프로세스들이 이벤트 처리를 위해 현재 대기중에 있는것을 볼 수 있습니다.
- **R** : ps af 명령은 현재 실행 중이므로 R 로 표시됩니다.
- **+** : 터미널에 연결돼 있는 interactive bash shell 프로세스들과 ps af 명령은 현재 foreground 로 실행중인걸 알 수 있습니다.
- **s** : 터미널에 연결돼 있는 interactive bash shell 프로세스들은 session leader 인것을 알 수 있습니다.
- **T** : test.sh 명령은 현재 suspend job control 명령에 의해 stop 상태입니다. ctrl-z 에 의해 프로그램 실행이 중단됐을때도 이 상태가 됩니다.
- **Z** : sub2.sh 은 test.sh 에서 실행된 background process 인데 현재 종료된 상태 (defunct) 이 나 parent 프로세스가 stop 되어있어 좀비 상태로 남아 있습니다. parent 프로세스가 종료되면 삭제됩니다.

좀비 프로세스는 이미 종료된 프로세스로 프로세스 테이블에서 pid 만 차지하고 있는 상태입니다. 그러므로 실행 중에 사용했던 메모리나 리소스들은 모두 반환된 상태입니다. child process 가 종료하게 되면 parent process 에서는 종료상태 값을 얻기 위해 wait system call 을 실행하는데 (보통 SIGCHLD 신호 핸들러를 통해서) 이때 child process 가 프로세스 테이블에서 삭제됩니다. 하지만 위의 경우는 parent process 가 stop 상태에 있어서 프로세스 테이블에서 삭제되지 못하고 좀비 상태로 남아있습니다.

- **<** : pulseaudio 는 현재 높은 우선순위로 실행되고 있습니다.
- **N** : rtkit-daemon 은 현재 낮은 우선순위로 실행되고 있습니다.
- **I** : X, pulseaudio, chrome 같은 프로그램들은 멀티 스레드를 사용하고 있습니다.

- **L** : 메모리 locking 은 프로세스가 사용하는 가상 주소 공간을 물리적인 메모리에 lock 합니다. lock 된 페이지는 paging, swapping 에서 제외되고 unlock 될때까지 메모리에 존재하는 것이 보장되므로 주로 실시간 처리가 요구되는 프로그램에서 latency 를 줄이기 위해 사용됩니다.
- **D** : D (uninterruptible sleep) 은 S (interruptible sleep) 과는 다르게 sleep 하는 동안 signal을 처리하지 않습니다. 심지어 kill 신호로 프로세스를 종료시킬 수도 없습니다. 주로 디바이스 드라이버에서 디스크나 네트워크 I/O 를 기다릴때 사용됩니다

[명령어] 으로 표시되는 경우 : ps 는 COMMAND 컬럼의 값을 표시할때

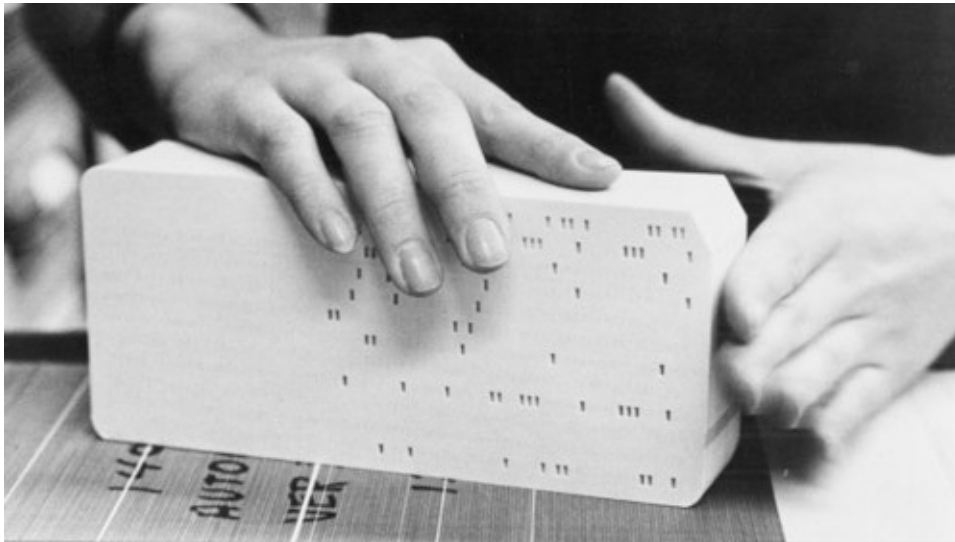
/proc/<pid>/cmdline 에 있는 내용을 사용하는데 여기에 값이 없을 경우 /proc/<pid>/stat 에 있는 명령 이름에 [] 를 더해서 표시한다고 합니다.

TTY

컴퓨터는 기본적으로 연산을 위한 입력장치와 출력장치를 가집니다. 지금은 기술이 좋아져서 노트북같은 경우 연산장치, 디스플레이 출력장치, 키보드 입력장치가 모두 같이 있지만 초기에는 대형 연산장치가 큰 방 하나를 가득 채우고 있었고 입, 출력장치는 따로 분리되어 운영되었습니다

리눅스의 `/dev` 디렉토리에서 볼수있는 `/dev/tty*` , `/dev/pts/*` 파일들이 입, 출력에 관련된 파일들이며 외부 터미널 장치를 연결할때, 리눅스에서 가상콘솔을 제공할때, `xterm` 이나 `gnome-terminal` 같은 터미널 emulation 프로그램, `telnet`, `ssh` 같은 리모트 로그인 프로그램등에 의해 사용됩니다.

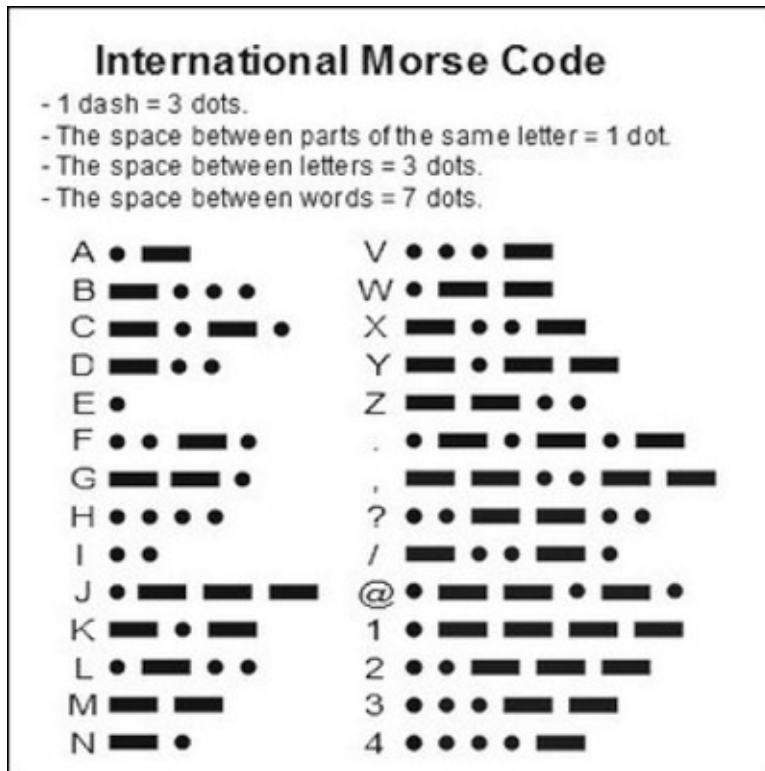
간단히 입, 출력 장치의 변천 과정을 살펴보면 초기에는 `punch card` 를 입,출력에 사용하였습니다. 아래 그림에서 여성 오퍼레이터 앞에 있는 기계에 `punch card` 를 넣고 작업하게 됩니다.





Morse 부호를 사용하는 전신 기술은 뒤에 이어지는 teletypewriter 의 기반이 됩니다.





Teletypewriter 는 입력으로 키보드와 출력으로 라인프린터를 가지고 있으며 통신회선이 연결되어 있습니다. 문자를 치면 자동적으로 전신 부호로 번역되어 송신되고, 수신측에서는 반대로 수신된 전신 부호가 문자로 번역되어 출력됩니다. 컴퓨터에 연결되어 입,출력에 사용되기도 하였으며 command line interface 의 원조라고 할수있습니다. 리눅스의 `/dev` 디렉토리에서 볼수있는 `tty` 파일의 이름은 이 TeleTYpe 에서 유래한 것입니다.

TELETYPE MODEL 33

TWX or
COMPUTER INTERFACE

\$840⁰⁰



TWX AS SHOWN
\$1,400.00

- 33ASR PRIVATE-LINE
- FRICTION FEED
- COPYHOLDER & STAND
- ANSWERBACK
- MANUAL READER
- GUARANTEED 30 DAYS
- F.O.B. NEW JERSEY
- CRATING INCLUDED
- NOTHING ELSE TO BUY

Options:

- * AUTOMATIC READER ADD \$50
- * READER RUN CARD (DEC) ADD \$75
- * SPROCKET (PIN) FEED ADD \$100
- * TAPE WINDER (ELECT.) \$55 - WINDUP \$22
- * EIA INTERFACE \$110
- * TAPE UNWINDER (MCM ELECT.) \$33
- * PAPER WINDER (ELECTRIC) \$50

--- NEW FREE CATALOG AVAILABLE NOW ---



**TELETYPEWRITER
COMMUNICATIONS
SPECIALISTS, INC.**
550 SPRINGFIELD AVENUE
BERKELEY HEIGHTS, N. J. 07922

**BUY * SELL
SERVICE * LEASE**

- * OVERHAULING & MODIFICATIONS
- * REPLACEMENT PARTS
- * PAPER-TAPE-RIBBONS
- * VIDEO TERMINALS
- * DECRYPTERS
- * ACOUSTIC COUPLERS

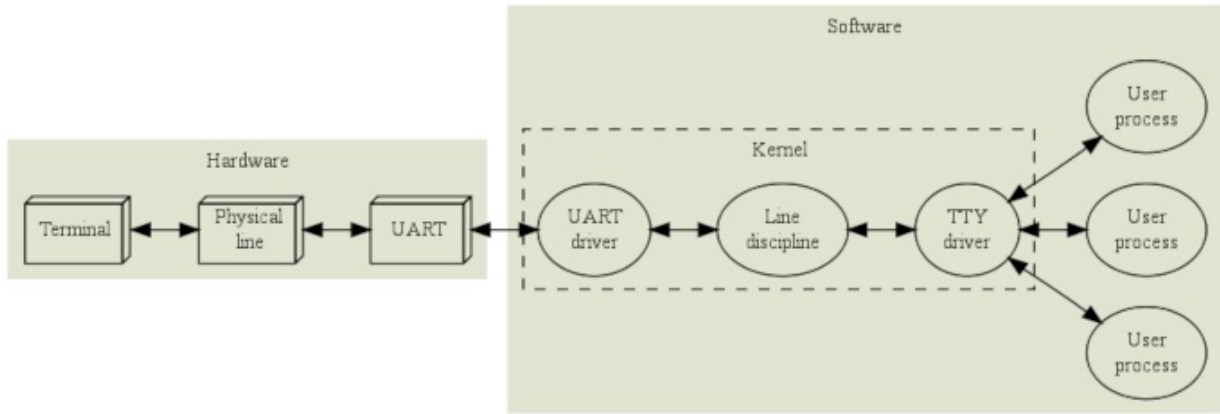
PHONE 201-464-5300 TWX 710-986-3006 TELEX 13-6479

Teletype Corporation 의 teletype model 33



2차대전 당시 teletypewriter 앞에서 작업하고 있는 오퍼레이터

출력에 사용되던 라인프린터가 CRT 모니터로 바뀌었습니다. 입, 출력을 위한 키보드와 디스플레이만으로 구성된 dumb terminal 입니다 (cpu, ram, disk 같은것은 들어있지 않습니다). 이때는 ethernet 같은 컴퓨터 네트워크가 생기기 전이라 메인프레임에 시리얼라인으로 터미널이 연결되어 사용되었습니다.



[출처] <http://www.linusakesson.net/programming/tty/>

위의 VT100 같은 외부 터미널 장치가 시리얼라인을 통해 연결되는 경우입니다. `getty` 프로세스가 백그라운드에서 라인을 모니터링 하고있다가 터미널에서 접속을하면 login prompt 를 보여줍니다. `/dev/ttyS[번호]` 파일이 사용됩니다.

- **UART driver**

물리적으로 bytes 를 전송하는 역할을하며 parity checks 이나 flow control 을 수행합니다.

- **Line discipline**

텍스트를 입력할때 한번에 오류없이 입력할수는 없죠. 그래서 OS에서는 기본적인 backspace, erase word, clear line, reprint 같은 기능을 default discipline 을 통해 제공합니다. (하지만 readline 이나 curses 같은 고급기능을 제공하는 프로그램은 raw mode 를 통해서 자신이 직접 모 든기능을 제어합니다.)

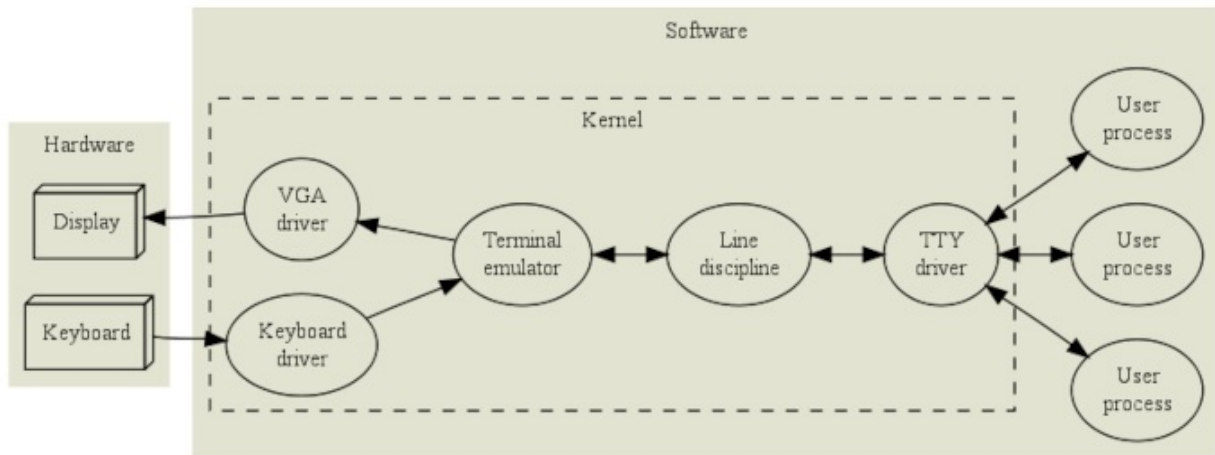
line editing 을 제공하는 default discipline 외에 커널에서는 목적에 따라 여러종류의 line discipline 을 제공하며 (예를 들면 managing packet switched data (ppp, IrDA, serial mice)) 이 들중에서 하나만 장치파일에 적용됩니다.

- **TTY driver**

Session management 기능, 다시말해서 job control 과 관련된 기능을 제공합니다. Ctrl-z 를 누르 면 실행중인 job 을 suspend 시키고, Ctrl-c 를 누르면 foreground job 에 SIGINT 신호를 보내 종 료시키고, 사용자 입력은 foreground 프로세스에게만 전달되게 하고, background 프로세스가 입 력받기를 하면 SIGTTIN 신호를 보내 suspend 시키는 등은 모두 TTY driver 에서 제공하는 기능 입니다.

UART driver, Line discipline, TTY driver 를 합쳐서 TTY device 라고 합니다.

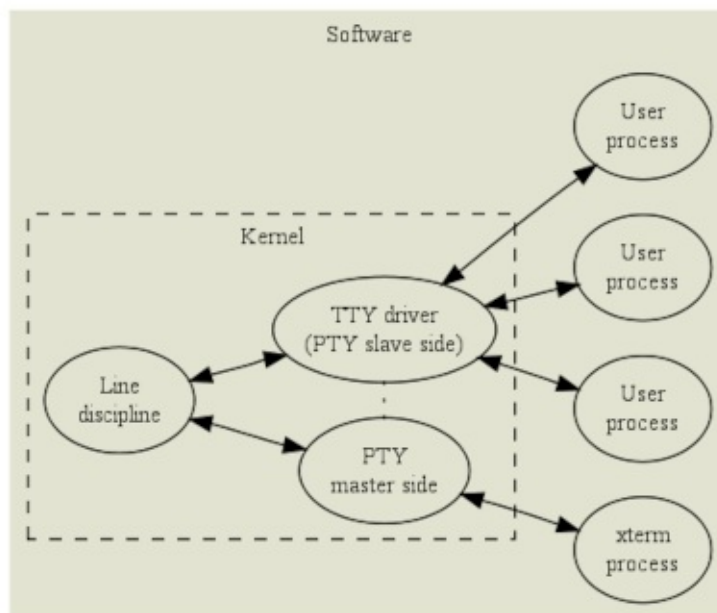
2. 리눅스 virtual console



[출처] <http://www.linusakesson.net/programming/tty/>

Ctrl-Alt-F1 ~ F6 키조합으로 사용할수있는 OS 에서 제공하는 가상콘솔 입니다. 실제 물리적인 터미널 장치가 연결된것이 아니기 때문에 커널에서 터미널을 emulation 합니다. Line discipline, TTY driver 의 기능은 위와같고 마찬가지로 백그라운드 `getty` 프로세스에의해 login prompt 가 제공됩니다 (`ps ax | grep getty` 로 볼수있습니다). `/dev/tty[번호]` 파일이 사용됩니다.

3. Pseudo TTY (xterm, gnome-terminal, telnet, ssh ...)



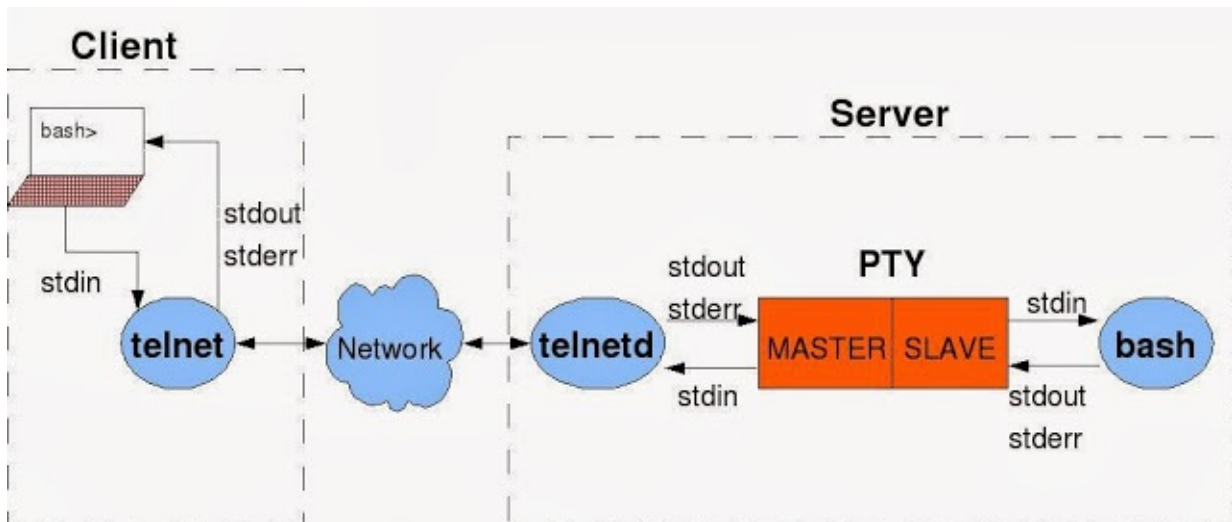
[출처] <http://www.linusakesson.net/programming/tty/>

앞서 virtual console 에서는 커널에서 터미널을 emulation 했다면, TTY driver 가 제공하는 session management 기능과 line discipline 을 그대로 사용하면서 사용자 프로그램에서 터미널을 emulation 하는것이 PTY (Pseudo TTY) 입니다.

PTY 는 master/slave pair 로 이루어지는데 /dev/ptmx 파일을 open 하면 pseudo terminal master (PTM) 에 해당하는 file descriptor 가 생성되고 pseudo terminal slave (PTS) 에 해당하는 device 가 /dev/pts/ 디렉토리에 생성됩니다. 일단 두 ptm 과 pts 가 open 되면 /dev/pts/[번호] 는 실제 터미널과 같은 인터페이스를 프로세스에 제공합니다. ptm 에 write 한 데이터는 pts 의 input 으로, pts 에 write 한 데이터는 ptm 의 input 으로 사용됩니다. 어떤 점에서 kernel 이 처리하는 레이어가 중간에 들어간 named pipe 와 비슷하다고 할 수 있습니다.

(터미널 프로그램을 실행하거나 외부에서 ssh client 가 접속하면 /dev/pts/ 디렉토리에 device 파일이 새로 생성되는 것을 볼 수 있습니다.)

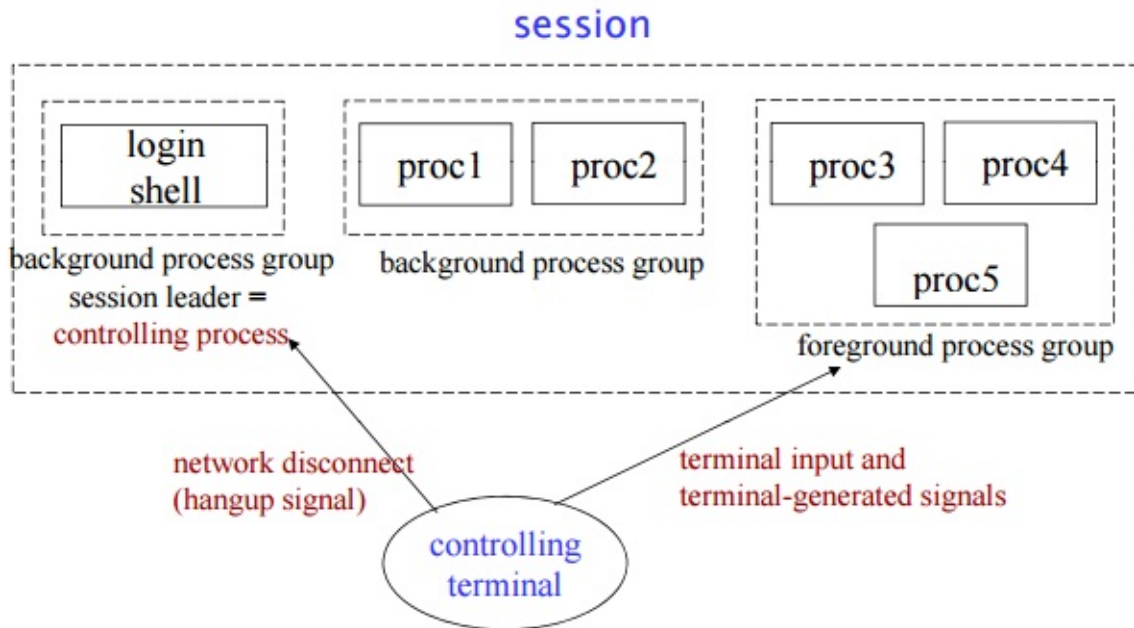
위 그림은 xterm 의 경우를 예로 든것으로 사용자가 xterm 에서 ls 명령을 입력하면 ptm -> line discipline -> pts 를 거쳐서 bash (user process) 에 전달되고 명령의 실행 결과가 pts -> line discipline -> ptm 을 통해서 xterm 에 전달되면 xterm 은 실제 터미널과 같이 화면에 표시하게 됩니다.



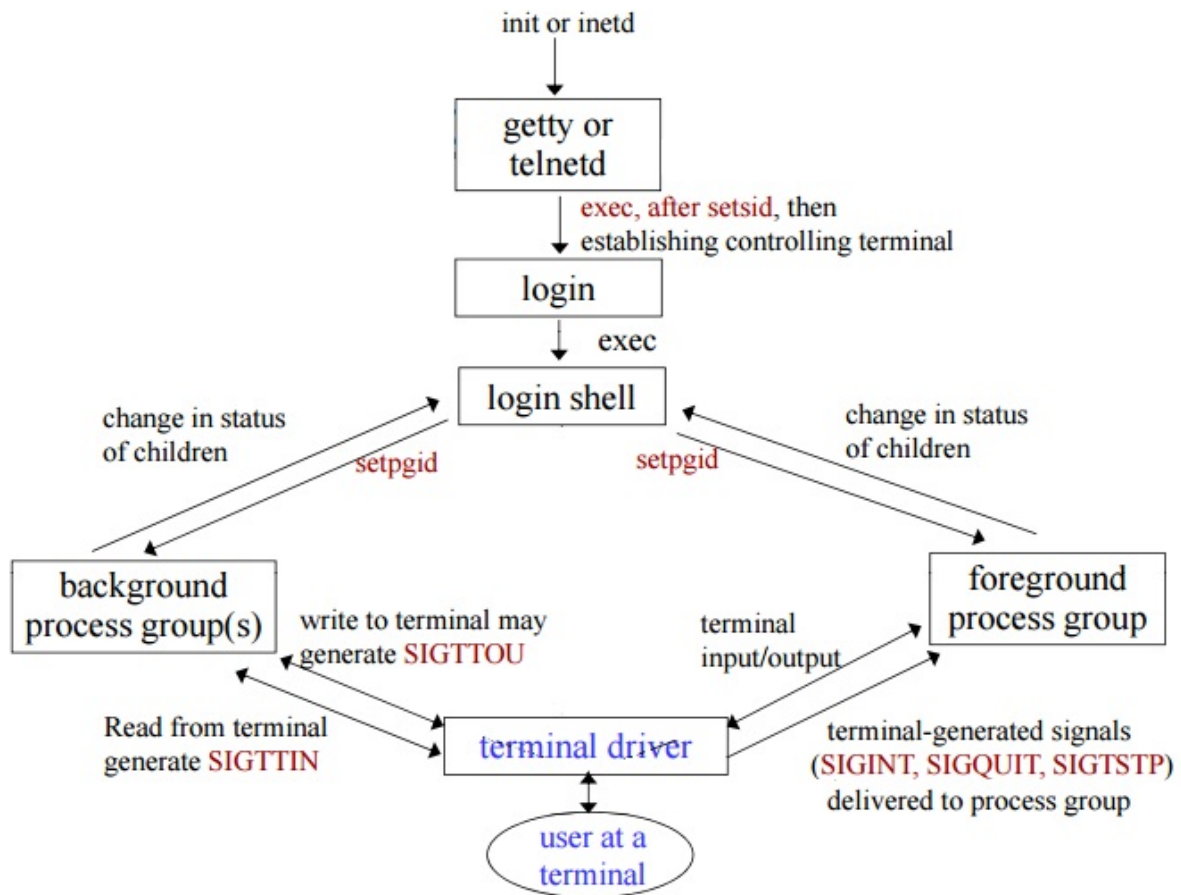
[출처] http://rachid.koucha.free.fr/tech_corner/pty_pdip.html

위 그림은 telnet 또는 ssh 의 경우인데 client 에서 ls 명령을 입력하면 네트워크를 거쳐 telnetd 에 전달되고 ptm, pts 를 거쳐 bash 프로세스에 전달됩니다. 명령의 실행결과는 다시 pts, ptm 을 거쳐서 telnetd 에 전달되고 네트워크를 거쳐 client 에 전달되게 됩니다.

Controlling Terminal



- 하나의 session 은 하나의 controlling terminal 을 가질 수 있습니다. 이것은 보통 `/dev/tty*` 나 `/dev/pts/*` 와 같은 터미널 device 를 말합니다.
- controlling terminal 과 연결된 session leader 를 controlling process 라고 부릅니다.
- 세션은 하나의 foreground process group 과 여러개의 background process groups 로 구성 됩니다.
- Ctrl-c 를 누르면 SIGINT 신호가 foreground process group 에 전달됩니다.
- modem (or network) 연결이 끊기면 SIGHUP 신호가 controlling process (session leader) 에 전달됩니다.
- `ps ax` 명령을 실행하였을때 두번째 TTY 컬럼에 나오는 내용이 controlling terminal (ctty) 입니다. ctty 를 갖지 않는 프로세스는 `?` 로 표시 됩니다.

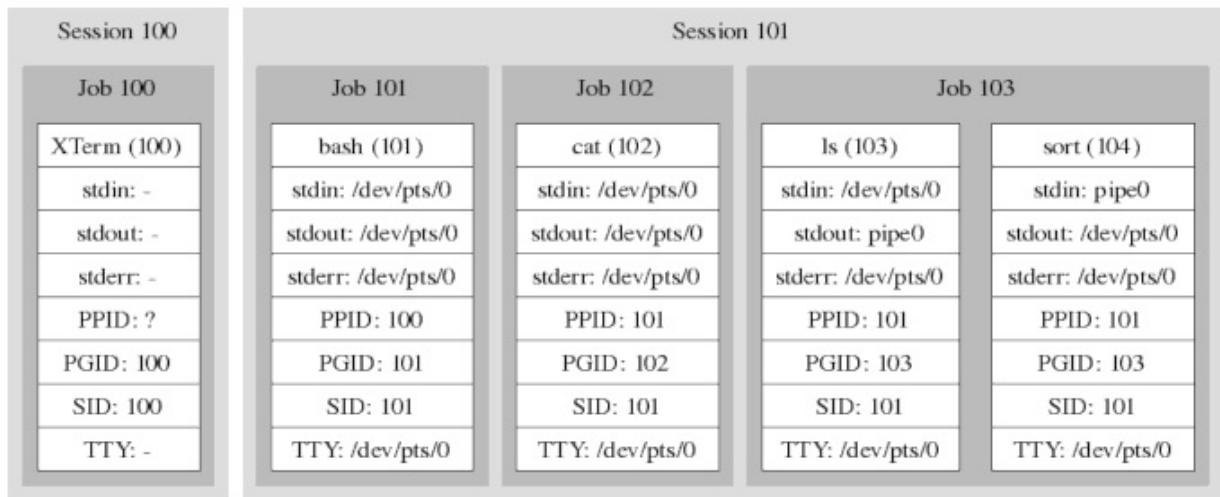


다음은 터미널에서 job control 을 이용해 여러개의 명령을 실행하는 예 입니다.

```

Terminal
lft@shizuku:~$ cat
hello
hello
^Z
[1]+  Stopped                  cat
lft@shizuku:~$ ls | sort

```



Xterm 에서 실행된 bash (101) 프로세스가 세션리더가 되고 /dev/pts/0 (controlling terminal) 과 연결된 것을 볼 수 있습니다. 그러므로 controlling process 가 됩니다. 이후 실행된 job 들은 모두 같은 sid (101) 와 controlling terminal 을 가지고 있는것을 볼 수 있습니다.

Xterm (100) 프로세스가 PTM 에 연결되고 나머지 job 프로세스들이 /dev/pts/0 (PTS) 에 연결되는 형태로 볼 수 있습니다.

/dev/tty

/dev/tty 는 특수 파일로 프로세스의 controlling terminal 과 동일합니다. 다시말해 현재 ctty 가 /dev/pts/1 이라면 /dev/tty 도 /dev/pts/1 와 같다고 할 수 있습니다. 예를들면 stdout 이 redirect 되어있다고 하더라도 /dev/tty 로 출력하면 터미널로 출력할 수 있습니다.

```
$ echo hello          # 정상적으로 출력이 됨
hello

$ exec 1> /dev/null

$ echo hello          # stdout 이 /dev/null 로 redirect 되어있어 터미널로 출력이 안됨

$ echo hello > /dev/tty # 하지만 /dev/tty 로 출력하면 터미널로 출력이 됨
hello
```

또한 어떤 프로세스가 /dev/tty 를 open 하는데 실패하였다면 ctty 를 갖고있지 않다고 할 수 있습니다.

Configuring TTY device

tty 명령으로 현재 shell 의 tty device 를 조회할수 있고, stty 명령을 이용해 설정값을 변경할 수 있습니다.

```
$ tty
/dev/pts/1

$ stty -a
speed 38400 baud; rows 13; columns 93; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = M
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc ixan
imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echok
```

위에서 출력된 설정값들을 보면 UART parameters, line discipline, TTY driver 에 해당되는 값들이 모두 섞여 있습니다. 가령 첫번째 라인의 speed 값은 UART parameters 에 해당되는 값인데 pseudo terminal 에서는 필요가 없으므로 무시됩니다. 다음의 rows, columns 값은 TTY driver 에 해당되는 값으로 터미널 프로그램의 윈도우 사이즈를 조절하면 값이 변경되고 TTY driver 는 foreground job 에 SIGWINCH 신호를 보내게 됩니다. line 은 line discipline 값으로 0 은 line editing 을 제공하는 default discipline 에 해당됩니다.

이어지는 설정값들을 몇가지 살펴보면 다음줄의 `intr = ^C` 는 인터럽트 키를 Ctrl-c 로 설정합니다. 단어앞에 dash (-) 가 붙은 것은 switch off 상태를 나타냅니다. icanon 은 line discipline 에 해당하는 값으로 canonical (line-based) 모드를 활성화합니다. `stty -icanon` 명령으로 off 상태로 변경하고 cat[enter] 한후에 타이핑을 해보면 backspace, ^U (Ctrl-u) 같은 키가 작동하지 않고 라인단위로 프린트되는 대신에 문자단위로 프린트되는 것을 볼 수 있습니다. 프롬프트에서 문자를 타입 할 때 타입한 문자가 보이는것도 line discipline 의 echo 옵션에 해당합니다. 그러므로 마지막 줄의 `echo` 값을 `stty -echo` 로 변경하면 echo 가 되지 않아 보이지 않게 됩니다.

job control 에 관련된 tostop 은 background job 에서 출력이 발생할시에 suspend 할지를 결정합니다. 현재는 off 상태이기 때문에 다른 프로세스의 출력과 겹치던 상관없이 터미널에 출력이 되지만 `stty tostop` 으로 on 설정을 해주면 출력이 발생할시에 suspend 시킬수 있습니다.

각 항목에 대한 설명은 man 페이지나, info 페이지를 통해 조회해볼 수 있습니다.

stdin, stdout, stderr

터미널 프로그램을 실행하면 /dev/pts/ 디렉토리에 device 가 새로 생성되고 controlling process 에 해당하는 shell 프로세스와 연결이 됩니다. 이때 shell 에서 /proc/\$\$/fd/ 를 조회해보면 stdin, stdout, stderr 에 해당하는 file descriptor 번호 0, 1, 2 번이 모두 새로 생성된 device 파일에 심볼릭 링크 되어있는 것을 볼 수 있습니다.

```

E$ ls -al /proc/$$/fd/
total 0
dr-x----- 2 mug896 mug896 0 12.03.2015 09:41 ./
dr-xr-xr-x 9 mug896 mug896 0 12.03.2015 09:41 ../
lrwx----- 1 mug896 mug896 64 12.04.2015 17:27 0 -> /dev/pts/15
lrwx----- 1 mug896 mug896 64 12.04.2015 17:27 1 -> /dev/pts/15
lrwx----- 1 mug896 mug896 64 12.03.2015 09:41 2 -> /dev/pts/15
lrwx----- 1 mug896 mug896 64 12.04.2015 17:27 255 -> /dev/pts/15

```

terminal 1 의 shell pid 가 123 이라고 한다면 또 하나의 터미널을 열어 terminal 2 에서 terminal 1 으로 데이터를 보낼때 /proc/123/fd/0 (stdin) 이나 /proc/123/fd/1 (stdout) 로 구분해서 보낼수 있을까요?

```

# terminal 1
# device      : /dev/pts/1
# shell pid   : 123
# file descriptor : /proc/123/fd/0 -> /dev/pts/1

# terminal 1
$ read line    # stdin 에서 입력을 받음.

# terminal 2
$ echo hello > /proc/123/fd/0

# terminal 1
$ echo $line    # 출력되는 데이터가 없다.

```

위의 예제를 실행해보면 terminal 2 에서 /proc/123/fd/0 로 데이터를 보냈지만 terminal 1 의 화면에만 hello 가 표시될뿐 read line 명령의 입력으로 들어가지 않습니다. stdout 도 마찬가지 입니다. 가령 terminal 1 에서 stdout 출력을 aaa 라는 파일로 쓰고 있다고 한다면 terminal 2 에서 /proc/123/fd/1 로 데이터를 보낸다고 해서 그 데이터가 aaa 파일에 쓰여지지 않습니다. (마찬가지로 terminal 1 의 화면에만 hello 가 표시됩니다.)

이렇게 되는 이유는 stdin, stdout, stderr 를 구분해서 처리하는 것은 shell 프로세스가 하는 것이기 때문입니다.

다음은 하나의 터미널에서 아래의 f1 함수를 실행하는 경우입니다. echo, date 명령으로 구성된 첫 번째 명령그룹은 stdout 으로 출력을 하고 있고 이어지는 두번째 명령그룹은 스트링을 빨간색으로 만들어주는 color escape 코드를 stderr 로 출력하고 있습니다. 각각 stdout, stderr 로 출력을 하고 있으니 첫번째 명령그룹의 출력이 빨간색으로 나올일은 없을까요?

```

f1() {
    { echo 111; date; echo 222 ;} &          # background 로 실행
    { echo -en "\e[31m"; sleep 1; echo -en "\e[m";} >&2
}

```

위의 f1 함수를 실행해 보면 첫번째 명령그룹의 출력이 빨간색으로 나옵니다. 다시말해 **stdout**, **stderr** 구분 없이 명령의 실행 순서가 `echo -en "\e[31m"; echo 111; date; echo 222; echo -en "\e[m";` 와 같이 되어 **terminal device** 로 출력된다고 할 수 있습니다.

두 개의 프로세스가 동시에 하나의 파일이나, 파이프에 쓰기를 한다면 쓰여지는 데이터의 순서를 예측할 수 없습니다. 위의 경우도 두 개의 프로세스가 동시에 하나의 **device** 파일에 쓰기를 하여 소위 말하는 **race condition** 이 일어나는 상황입니다.

두 명령그룹의 출력을 각각 **redirection** 을 통해 파일로 쓰기를 한다면 **shell** 이 구분하여 처리하므로 위와같은 상황은 발생하지 않게 됩니다.

Signals and Traps

Signal 은 프로세스에 전달되는 software interrupt 입니다. OS 내에 정의되어 있고 여러가지 예외적인 오류상황이 발생했을때 프로세스에 알리기위해 사용합니다. 터미널에서 실행중인 프로그램을 종료하기위해 Ctrl-c 키를 누를때도 signal 이 사용되며 프로세스 A 가 프로세스 B 를 종료시킬때, 또는 두 프로세스 간에 서로 통신을 할때도 signal 을 사용할 수 있습니다.

신호 보내기

신호는 비동기적으로 전송이 됩니다. 다시말해 보내는 쪽에서 임의대로 아무때나 상대 프로세스에 게 보낼 수 있습니다. 신호를 보낼 때 특정 메시지를 함께 전달하는 것은 아니며 신호를 받는 쪽에서는 누가 신호를 보냈는지조차 알 수 없습니다. 신호는 자기 자신에게 보낼 수도 있고 다른 프로세스에게 보낼 수도 있습니다.

신호 받기

신호를 받으면 프로세스는 실행을 중단하고 signal handler 를 실행합니다. shell 에서는 trap 명령을 이용하여 자체 제작한 signal handler 를 등록해 사용할수 있으며 또한 신호를 ignore 하여 signal handler 가 실행되는 것을 막을 수도 있습니다.

signal 은 비동기적으로 전송되므로 signal handler 가 실행 중에 있는 상태에서 동일한 신호가 전달되면 동시에 실행될 수 있습니다. 하지만 shell 에서는 실행 중인 signal handler 가 종료될 때까지 신호를 ignore 합니다. 그러므로 가령 signal handler 에서 전달받은 신호 개수를 카운트한다면 올바른 결과를 얻을 수 없습니다.

스크립트에서 background 로 여러 프로세스를 실행시킨 후에 SIGCHLD 신호를 trap 해서 종료상태값을 구한다고 했을 경우에도 동시에 프로세스가 종료하게 되면 중복되는 신호는 잃어버리게 됩니다.

Signal handler

Signal 이 프로세스에 전달되었을때 실행되는 코드를 signal handler 라고 합니다. 모든 signal 은 default action 을 위한 signal handler 를 가지고 있습니다. 그러므로 SIGSTOP 신호가 전달되면 프로세스가 중단되고 SIGKILL 신호를 받으면 종료되는 것입니다.

kill

```
kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
```

Signal table 을 보면 알 수 있겠지만 대부분의 신호들은 치명적인 오류를 나타내고 default action 으로 프로세스가 종료됩니다. 그러므로 왜 신호를 보내는 명령 이름이 kill 인지 짐작을 할 수 있습니다. kill 명령으로 신호를 보낼때 신호값을 주지 않으면 기본값 또한 TERM 신호가 됩니다.

다음은 kill 명령 사용 예입니다. 신호명 앞에 SIG 를 제외하고 사용할수 있고 소문자로 쓸수도 있습니다. 또한 pid 대신에 jobspec 을 사용할수 있습니다.

```
kill -TERM 4287

kill -SIGTERM 4287

kill -s TERM 4287

kill -s SIGTERM 4287

kill -15 4287

kill -n 15 4287

kill 4287          # 신호값을 주지 않으면 기본값은 TERM 신호가 된다.
```

kill 명령을 이용해 process group 에 신호를 보낼수 있습니다. 이때는 pgid 앞에 - 문자를 붙여 사용합니다.

```
kill -TERM -4287
```

kill -0

kill -0 은 실질적으로 신호를 보내지 않지만 프로세스가 아직 살아있는지, 살아있다면 신호를 보낼수있는 권한이 되는지 테스트 하는데 사용합니다.


```
$ kill -0 32630; echo $?
0                                     # process 가 살아있고 시그널을 보낼수있는 권한이 됨

$ kill -0 32631; echo $?
bash: kill: (32631) - No such process  # process 가 존재하지 않음
1

$ kill -0 1; echo $?
bash: kill: (1) - Operation not permitted  # process 가 존재하나 시그널을 보낼수있는 권한이 안됨
1
```

trap

```
trap [-lp] [[arg] signal_spec ...]
```

스크립트를 작성할때 임시파일 이나 파이프같은 리소스를 생성해 사용하거나 필요한 사전 설정작업을 할수있습니다. 또한 스크립트 실행중에 임시 결과물이 발생할수도 있습니다. 스크립트가 정상적으로 종료되면 뒤처리 작업을 거치게 되므로 문제가 없겠지만 실행도중에 사용자가 Ctrl-c 로 종료를 시도한다던지, 아니면 터미널 프로그램을 종료시킨다면 문제가 될수있습니다.

trap 명령은 시스템에서 비동기적으로 발생하는 신호를 잡아서 필요한 작업을 수행할수 있게 해줍니다.

- signal handler 를 등록하여 실행

```
myhandler() { ... ;}

trap 'myhandler' EXIT
```

- 기존에 설정했던 handler 를 default 값으로 reset

```
trap - INT
```

- signal 을 ignore 하여 handler 가 실행되지 못하게 함

```
trap '' INT
```

- 아무 설정도 하지 않음

default handler 가 실행됨

trap 에 사용할수 없는 신호들

다음 3 개의 신호는 trap 명령을 이용해 ignore 하거나 signal handler 를 등록해 사용할 수 없습니다. 다시말해 무조건 default handler 가 실행됩니다.

- SIGKILL
- SIGSTOP
- SIGCONT

Pseudo signals

스크립트를 작성할때 `trap` 명령에서 주로 사용되는 신호들을 [signal table](#) 에서 찾아보면 [Termination Signals](#), [Job Control Signals](#) 카테고리 에 있는 신호들 정도 일것입니다. `table` 을 보면 알수있듯이 스크립트가 종료될때 실행되는 `handler` 를 등록하려면 `HUP`, `INT`, `QUIT`, `TERM` 신호를 모두 등록해야 합니다. 그리고 정상 종료될 경우에도 실행이 돼야 하므로 다음과 같이 작성해야 합니다.

```
#!/bin/bash

myhandler() { ... ;}

trap 'myhandler' HUP, INT, QUIT, TERM
...
...
...
myhandler # 정상종료시 처리를 위해
```

이와 같은 불편을 없애기 위해 `shell` 에서는 `EXIT` pseudo-signals 을 제공합니다. `trap` 에 `EXIT` 신호를 사용하면 어떤 종료 상황에서도 마지막에 `handler` 가 실행됩니다. 다시말해 정상종료, `Ctrl-c` 에 의한 종료, 터미널 프로그램의 종료, 시스템 `shutdown` 에 의한 종료 상관없이 마지막에 스크립트가 `exit` 될때 실행이 됩니다. 그러므로 위의 예를 `EXIT` 신호를 이용해 다시 작성하면 다음과 같이 할수있습니다.

```
#!/bin/bash
myhandler() { ... ;}

trap 'myhandler' EXIT
...
...
```

Shell 에서 제공하는 pseudo-signals 에는 `EXIT` 외에 함수나 `source` 한 파일에서 `return` 할때 사용할수 있는 `RETURN` 신호, 명령이 0 이 아닌값으로 종료했을때 사용할수있는 `ERR` 신호, 스크립트 디버깅에 사용할수있는 `DEBUG` 신호를 제공합니다. 이들 신호의 설정은 `subshell` 까지 적용이 되며 `child process` 에는 적용되지 않습니다. 또한 `EXIT` 신호 설정은 개별 프로세스에 해당하는 것으로 `subshell` 에도 상속되지 않습니다.

| Signal | Description |
|--------|-----------------------------------|
| EXIT | shell 이 exit 할때 발생. |
| ERR | 명령이 0 이 아닌값을 리턴할경우 발생. |
| DEBUG | 명령 실행전에 매번 발생. |
| RETURN | 함수에서 리턴할때, source 한 파일에서 리턴할때 발생. |

RETURN

```
#!/bin/bash

f1() {
    trap 'echo f1 return trap' RETURN

    echo 111
    echo 222
}

echo start---

f1
f1

echo end---
```

output

```
start---
111
222
f1 return trap
111
222
f1 return trap
end---
```

ERR, DEBUG

[Debugging](#) 메뉴 참조

EXIT handler 와 다른 신호 handler 를 같이 등록하면?

만약에 EXIT handler 와 INT handler 를 함께 등록했다면 종료시 INT handler 가 먼저 실행되고
마지막에 EXIT handler 가 실행됩니다.

Ctrl-c

Ctrl-c 키의 SIGINT 신호를 이용한 trap 은 다음 세 가지로 활용할 수 있습니다.

- 사용자가 Ctrl-c 로 종료하지 못하게 ignore 하기
- child process 만 종료시키기
- 전체 process group 을 종료시키기

기본적으로 Ctrl-c 키를 입력했을때 발생하는 SIGINT 신호는 process group 에 전달되므로 같은 process group 에 속한 명령들은 모두 종료하게 됩니다. 다음은 A.sh 에서 B.sh 을 실행하고 B.sh 에서는 sleep 외부명령을 실행하고 있는 예입니다. sleep 명령에 의해 실행이 중단됐을때 Ctrl-c 키를 입력하면 A.sh, B.sh 모두 종료하는것을 볼수있습니다.

```

----- A.sh -----
#!/bin/bash
echo A.sh --- start
./B.sh
echo A.sh --- end

----- B.sh -----
#!/bin/bash
echo B.sh ----- start
sleep 100
echo B.sh ----- end
-----

$ ./A.sh
A.sh --- start
B.sh ----- start
^C                               # Ctrl-c 키를 입력했을때 A.sh, B.sh 모두 종료된다.

```

이번에는 child process 만 종료시키는 예입니다. 먼저 B.sh 에 trap 설정을 하는데 마지막 명령으로 exit 을 사용하였습니다. 프롬프트에서 A.sh 을 실행시킨후 Ctrl-c 를 입력한 결과는 A.sh 의 child process 인 B.sh 만 종료가 되고 이후에 A.sh 스크립트의 나머지 부분이 실행되어 A.sh --- end 메시지를 볼수있습니다.

```

----- B.sh -----
#!/bin/bash

trap 'echo trap INT in B.sh; exit' INT

echo B.sh ----- start
sleep 100
echo B.sh ----- end
-----

$ ./A.sh
A.sh --- start
B.sh ----- start
^Ctrap INT in B.sh
A.sh --- end                # B.sh 만 종료되고 A.sh 의 나머지 부분이 실행된다.

```

이와같은 예는 ping 명령을 통해서도 확인할수 있습니다. 아래의 경우 B.sh 에서 ping 명령을 실행시킨후 Ctrl-c 로 종료하였으나 ping 명령 자체만 종료가 되고 B.sh, A.sh 의 나머지 명령들은 모두 실행되는것을 볼수있습니다. 다시말해 ping 명령 내에서의 trap 설정은 위의 예와같이 trap

'command ... ; exit' INT 형식으로 되어있다고 할수있습니다.

```

----- B.sh -----
#!/bin/bash
echo B.sh ----- start
ping 1.1.1.1
echo B.sh ----- end
-----

$ ./A.sh
A.sh --- start
B.sh ----- start
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
^C
--- 1.1.1.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 999ms

B.sh ----- end    # ping 명령만 종료되고 B.sh, A.sh 나머지 명령이 모두 실행된다.
A.sh --- end

```

위의 ping 명령과 같은 경우에 사용자가 Ctrl-c 키를 입력하였을때 ping 명령 뿐만아니라 A.sh, B.sh 모두 같이 종료되어야할 경우가 있습니다. 이럴때는 다음과 같이 trap 설정을 하면 해결할수 있습니다. 기존의 trap 설정에서 exit 을 빼고 trap 설정을 reset 한다음 자기 자신에게 다시 INT 신호를 보내면 parent process 인 B.sh, A.sh 에서도 terminate default action 이 실행되어 process group 에 있는 명령들이 모두 종료하게 됩니다.

```

----- B.sh -----
#!/bin/bash

trap 'echo trap INT in B.sh; trap - INT; kill -INT $$' INT

echo B.sh ----- start
ping 1.1.1.1
echo B.sh ----- end
-----

$ ./A.sh
A.sh --- start
B.sh ----- start
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
^C
--- 1.1.1.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 999ms

trap INT in B.sh # ping 명령과 함께 A.sh, B.sh 이 모두 같이 종료가 되었다.

```

다음은 B.sh 스크립트 파일 대신에 subshell 을 이용한 예입니다.
subshell 이므로 \$\$ 대신에 \$BASHPID 가 사용된걸 볼수있습니다.

```
----- A.sh -----
#!/bin/bash
echo A.sh --- start
(
trap 'trap - INT; kill -INT $BASHPID' INT
ping 1.1.1.1
)
echo A.sh --- end
-----

$ ./A.sh      # Ctrl-c 키를 입력했을때 전체 스크립트가 종료됩니다.
A.sh --- start
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
^C
--- 1.1.1.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 999ms
```

Signal Table

Program Error Signals

| Signal | Num | DefaultAction | Description |
|---------|-----|--------------------------|---|
| SIGFPE | n/a | Terminate (core dump) | 산술연산 에러를 나타냅니다. 이름은 floating-point exception 에서 왔지만 실질적으로 모든 산술연산 에러를 포함합니다. (Floating/Integer division by zero, modulo operation by zero, Floating/Integer overflow ...) |
| SIGILL | n/a | Terminate (core dump) | 프로세스가 illegal, malformed, unknown, or privileged instruction 을 실행하려 하면 전달됩니다. |
| SIGSEGV | n/a | Terminate (core dump) | 프로세스에 할당된 메모리 주소 범위를 벗어나 참조하게 되면 segmentation violation 이 발생하고 신호가 프로세스에 전달됩니다. (또는 read-only memory 에 쓸경우) |
| SIGBUS | n/a | Terminate (core dump) | 정렬되지 않은 메모리 주소를 사용하거나 존재하지 않는 물리적 주소를 사용하게 되면 프로세스에 전달됩니다. |
| SIGABRT | 6 | Terminate (core dump) | 프로그램 코드에서 비정상적인 종료를 할 경우를 나타내며 abort function 을 이용해 종료할때 발생합니다. |
| SIGTRAP | n/a | Terminate (core dump) | 디버거에서 사용하는 신호로 특정함수 실행, 변수 값 변경 같은 상황이 발생하면 전달됩니다. |
| SIGEMT | n/a | Terminate (core dump) | software 로 emulation 되지 않은 instruction 이 실행되거나 OS 에서 emulation 실패가 발생하면 프로세스에 전달됩니다. |
| SIGSYS | n/a | Terminate (core dump) | system call 을 할때 인수전달에 오류가 있으면 발생합니다. system call 을 하기위해 주로 libc 를 이용하므로 실질적으로 거의 발생하지 않습니다. |

Termination Signals

| Signal | Num | DefaultAction | Description |
|---------|-----|--------------------------|---|
| SIGTERM | 15 | Terminate | 프로세스에 종료를 요청할때 사용합니다. SIGKILL 과 달리 trap 하거나 ignore 할 수 있습니다. signal handler 를 이용하여 종료하기 전에 필요한 뒤처리 작업을 할 수 있습니다. |
| SIGINT | 2 | Terminate | 실행중인 프로세스를 interrupt 할때 사용되는 신호로 터미널에서 Ctrl-c 키를 입력하면 프로세스에 전달되고 실행중인 프로세스는 종료됩니다. |
| SIGQUIT | 3 | Terminate (core dump) | 터미널에서 Ctrl-\ 키를 입력하면 발생하며 프로세스가 종료할때 추가로 core dump 합니다. 사용자에게 의한 에러 발견으로 볼 수 있습니다. |
| SIGKILL | 9 | Terminate | 이 신호를 받으면 프로세스는 즉시 종료합니다. SIGTERM, SIGINT 와 달리 trap 하거나 ignore 할 수 없으므로 필요한 뒤처리 작업을 할 수 없습니다. |
| SIGHUP | 1 | Terminate | 터미널과 network, 모뎀 연결이 끊기거나 또는 터미널 프로그램이 종료될경우 모든 자손 프로세스들에 전달됩니다. 이 신호는 용도가 하나 더 있는데 daemon 프로세스에서 사용하는 설정파일을 수정할 경우 restart 할 필요 없이 이 신호를 보내면 설정파일을 reload 합니다. |

Alarm Signals

| Signal | Num | DefaultAction | Description |
|-----------|-----|---------------|--|
| SIGALRM | 14 | Terminate | real or clock 시간을 나타내며 alarm or setitimer function 에서 설정한 값이 초과되면 발생합니다. |
| SIGVTALRM | n/a | Terminate | 프로세스에서 사용한 CPU 시간이 alarm or setitimer function 에서 설정한 값을 초과했을 경우 발생합니다. |
| SIGPROF | n/a | Terminate | 프로세스 에서 사용한 cpu 시간과 프로세스를 대신해 system call 을 하는데 소비된 cpu 시간이 alarm or setitimer function 에서 설정한 값을 초과했을 경우 전달됩니다. code profiling 하는 프로그램에서 사용됩니다. |

Asynchronous I/O Signals

| Signal | Num | DefaultAction | Description |
|---------|-----|---------------|---|
| SIGIO | n/a | Ignore | socket 이나 터미널 같은 FD (file descriptor) 에서 input 또는 output 할 준비가 되면 발생합니다. kernel 이 caller 를 대신해 FD 를 조사해 신호를 전달하므로 비동기 I/O requests 만드는데 사용할 수 있습니다. |
| SIGURG | n/a | Ignore | socket 에서 urgent or out-of-band 데이터가 도착하면 전달됩니다. |
| SIGPOLL | n/a | Ignore | System V 신호이름으로 SIGIO 와 같습니다. (호환성 유지를 위해 정의됨) |

Job Control Signals

| Signal | Num | DefaultAction | Description |
|---------|-----|---------------|---|
| SIGCHLD | n/a | Ignore | child 프로세스가 terminate, stop, continue 상태 변경이 되면 parent 프로세스에 전달됩니다. |
| SIGCONT | n/a | Continue | SIGSTOP, SIGTSTP 신호로 중단된 프로세스를 다시 시작합니다. (실행중인 프로세스에서는 무시됩니다.) |
| SIGSTOP | n/a | Stop | 이 신호를 받으면 프로세스는 즉시 정지합니다. trap 하거나 ignore 할 수 없습니다. |
| SIGTSTP | n/a | Stop | 터미널에서 Ctrl-z 키를 입력할때 발생하며 SIGSTOP 과는 달리 trap 하거나 ignore 할 수 있습니다. |
| SIGTTIN | n/a | Stop | background process 가 tty 로부터 읽기를 시도하면 전달됩니다. |
| SIGTTOU | n/a | Stop | background process 가 tty 로 쓰기를 시도하면 전달됩니다. 터미널 옵션 tostop 가 enable 되어 있어야 합니다. (stty -a 로 값을 볼수있음) |

Operation Error Signals

| Signal | Num | DefaultAction | Description |
|---------|-----|---------------|---|
| SIGPIPE | n/a | Terminate | broken pipe 를 나타냅니다. writer 가 파이프로 데이터를 쓰고 있는중에 reader 가 종료하거나 또는 connect 되지 않은 socket 에 데이터를 전송하면 발생합니다. |
| SIGLOST | n/a | Terminate | Resource lost 를 의미합니다. 가령 NFS 에서 파일에 lock 을 가지고 있는데 NFS 서버가 reboot 된다면 lock 을 잃어버리게 되어 신호가 프로세스에 전달됩니다. |
| SIGXCPU | n/a | Terminate | 사용가능한 CPU 시간 (soft limit) 을 초과했을 경우 전달되며 신호를 받은 프로세스는 필요한 뒤처리 작업을 할수있고 이후에 OS 의 SIGKILL 신호에 의해 종료됩니다. |
| SIGXFSZ | n/a | Terminate | 시스템에서 사용가능한 파일크기 (soft limit) 을 초과하려 할때 발생합니다. |

Miscellaneous Signals

| Signal | Num | DefaultAction | Description |
|----------|-----|---------------|---|
| SIGUSR1 | n/a | Terminate | 사용자가 정의하여 사용할 수 있는 신호입니다 1. |
| SIGUSR2 | n/a | Terminate | 사용자가 정의하여 사용할 수 있는 신호입니다 2. |
| SIGWINCH | n/a | Ignore | 터미널 윈도우 사이즈가 변경되었을때 프로세스에 전달됩니다. |
| SIGINFO | n/a | Ignore | 터미널의 foreground process group 에 모두 전달되며 group leader 일경우 시스템 상태정보와 프로세스가 실행중인 작업에 대한 정보를 표시합니다. |

Real time Signals

SIGRTMIN ~ SIGRTMAX 는 real time signal 입니다. 유닉스 표준 시그널에서는 시그널이 블럭 될 경우 하나의 시그널만 유지하고 나머지는 모두 잃어 버리지만 RTS는 블럭되더라도 시그널의 queue를 유지합니다.

RTS는 비동기 이벤트를 전달하기 위한 목적으로 만들어 졌으며, 주로 네트워크 애플리케이션 작성시 소켓 이벤트를 통보하기 위해서 사용합니다. RTS는 네트워크 입출력에 있어서 polling에 비해 월등한 성능 향상을 보장해 줍니다. 시그널의 장점인 실시간성을 유지하면서 단점인 queue 부재의 문제를 해결한 향상된 시그널 도구라고 할 수 있습니다.

Num

신호이름 대신에 번호를 사용할 수 있는데요. POSIX standard 로 정해진 것은 몇개 안되고 나머지는 OS 마다 틀리다고 합니다.

DefaultAction

- **Terminate**
프로세스를 종료합니다.
- **Terminate (core dump)**
프로세스를 종료하고 추가적으로 core 파일을 생성합니다.
- **Ignore**
신호를 무시합니다.
- **Stop**
프로세스를 stop 합니다. (종료하는 것은 아닙니다.)
- **Continue**
프로세스가 stopped 상태일 경우 실행을 재개합니다. (실행 중에는 무시됩니다.)

Shell Options

Shell 이 동작하는 방식을 여러 옵션을 통해 컨트롤할 수가 있습니다. 옵션을 설정할때 사용할수 있는 명령은 `set` 과 `shopt` 두가지가 있는데 `set` 명령의 설정값은 `SHELLOPTS` 변수에 `shopt` 명령의 설정값은 `BASHOPTS` 변수에 각각 저장됩니다. `set` 옵션의 경우에는 보통 짧은 옵션도 함께 제공합니다.

sh 에서는 `shopt` 을 사용할 수 없습니다.

Set

Set 명령으로 설정할 수 있는 옵션들의 목록과 현재 상태 값은 `set -o` 으로 볼수가 있습니다. 값을 `enable` 할때는 `set -o 옵션명` 을 사용하고 `disable` 할때는 `set +o 옵션명` 을 사용합니다. `[-o 옵션명]` 으로 현재 값을 테스트해볼 수 있으며 `enable` 일경우 `0` 을 `disable` 일경우 `1` 을 반환합니다. 또한 설정된 옵션은 `$-` 특수변수에 `flags` 으로 저장됩니다.

--

- `set --` 은 현재 설정되어있는 인수들을 전부 `unset` 합니다.
- `set -- 11 22 33` 은 11 22 33 을 각각 `$1` `$2` `$3` 에 할당합니다.

-

- `xtrace`, `verbose` 옵션을 turn off 시킵니다.
- `set -` 은 `set --` 과 달리 설정되어있는 인수들을 `unset` 하지 않습니다.
- `set - 11 22 33` 은 11 22 33 을 각각 `$1` `$2` `$3` 에 할당합니다.

```
#!/bin/sh

line="11:22:33"

set -f; IFS=:          # globbing 을 disable
set -- $line           # IFS 값에 따라 필드를 분리하여 positional parameters 에 할당
set +f; IFS=$' \t\n'

echo number of fields = $#
echo field1 = "$1"
echo field2 = "$2"
echo field3 = "$3"
```

-a | allexport

설정된 functions, variables 들이 자동으로 export 되어 child process 에서 사용할 수 있게됩니다.

-B | braceexpand

Brace 확장을 사용합니다. 기본값입니다.

emacs

emacs 스타일 line editing 을 사용합니다.

-e | errexit

script 실행중에 명령이 에러로 종료하면 exit 합니다. if, while, until, ||, && 와 함께 사용되는 경우에는 적용되지 않습니다.

-E | errtrace

명령치환, subshell, shell function 에서도 ERR trap 을 가능하게 합니다.

-T | functrace

명령치환, subshell, shell function 에서도 DEBUG 와 RETURN trap 을 가능하게 합니다.
RETURN trap 은 기본적으로 source 한 파일이 return 했을때 발생합니다.

-h | hashall

명령이 사용될때 위치를 기억합니다. 기본값입니다.

-H | histexpand

! 문자를 이용한 history 확장 기능을 제공합니다. `set -o history` 이 설정돼있어야 사용할 수 있습니다.

history

명령 history 기능을 enable, disable 할 수 있습니다.

ignoreeof

ctrl-d 키의 명칭은 `end of file` 로 터미널로부터 직접 입력을 받는 프로그램에서 입력을 마칠때 사용합니다. 또한 프롬프트 상에서 ctrl-d 를 입력하면 `exit` 명령을 한것과 같이 터미널이 종료되는데 이 옵션을 설정하면 `$IGNOREEOF` 변수에 설정된 값을 따릅니다. (가령 설정값이 10 이면 10 회 ctrl-d 입력을 해야 exit 합니다.)

interactive-comments

프롬프트에서 `#` 코멘트를 사용할수 있습니다.

-k | keyword

`command var1=val1 var2=val2` 와같이 사용하며 `command` 에만 적용되는 일시적인 변수설정을 가능하게 해줍니다.

-m | monitor

job control 기능을 사용합니다. script 실행시 이옵션은 disable 됩니다. CHLD signal 을 trap 하기 위해서는 이 옵션이 활성화 되어있어야 합니다.

-C | noclobber

redirection 에 의해 파일이 overwriting 되지 못하게 합니다. 하지만 `>|` 을 사용하면 overwrite 할 수 있습니다.

-n | noexec

script 내의 명령들을 실행하지 않으므로 syntax errors 를 checking 하는 용도로 사용할수 있습니다.

-f | noglob

globbing 사용을 disable 합니다. globbing 을 회피하고자 할 때 사용할 수 있습니다

nolog

현재 사용안함.

-b | notify

background process 가 종료하면 기본적으로 다음 프롬프트가 표시될때 종료메시지를 함께 보여줍니다. 하지만 이 옵션을 설정하면 프로세스 종료시 바로 프롬프트상에 종료메시지를 표시해 줍니다.

-u | nounset

변수 확장을 할 때 존재하지 않는 변수일 경우 에러로 간주하여 exit 합니다.

-t | onecmd

옵션설정 뒤에 따라오는 명령을 실행하고 다음 프롬프트 전에 자동으로 exit 합니다.

```
$ set -o onecmd; command1;
```

-P | physical

cd 할때 심볼릭 링크 디렉토리 구조를 따르지 않고 물리적 디렉토리 구조를 따릅니다.

```
# 예를들어 /usr/sys 가 /usr/local/sys 를 symbolic link 한다면

$ cd /usr/sys; echo $PWD
/usr/sys
$ cd ../; pwd
/usr

# 옵션을 설정 할경우는 :

$ cd /usr/sys; echo $PWD
/usr/local/sys
$ cd ../; pwd
/usr/local
```

pipefail

파이프로 연결된 명령들이 실행될때는 마지막 명령의 종료값이 true, false 를 판단하는데 사용됩니다. 하지만 이 옵션을 설정하면 연결된 명령들 중에 하나라도 false 이면 false 가 됩니다.

```
$ ( true | false | true ) && ( true ); echo $?
0

$ set -o pipefail
$ ( true | false | true ) && ( true ); echo $?
1
```


posix

POSIX 모드로 실행합니다.

-p | privileged

shell script 파일에는 set uid 비트를 설정해도 소용이 없기 때문에 bash 프로그램을 복사하여 set uid 비트를 설정해 사용하는 경우를 말합니다. 실행시에는 보안을 위해 해당파일 소유자 (effective user) 의 스타트업 파일, \$BASH_ENV, \$ENV 파일은 처리되지 않고 현재 export 되어있는 함수들도 사용할수 없으며 SHELLOPTS, BASHOPTS, CDPATH, GLOBIGNORE 같은 변수들은 무시됩니다. suid 비트가 설정되어 있더라도 이 옵션을 주어야지 effective user id 로 실행이 되며 그렇지 않으면 real user id 로 실행됩니다.

```
### alice 계정 ###

alice@box:/tmp/priv$ cp /bin/bash bash
alice@box:/tmp/priv$ chmod u+s bash      # suid 비트 설정
alice@box:/tmp/priv$ id
uid=1004(alice) gid=1005(gamers) groups=1004(alice),1005(gamers)

### bob 계정 ###

# suid 비트가 있지만 euid 가 설정되지 않는다.
bob@box:/tmp/priv$ ./bash
bash-4.3$ id
uid=1002(bob) gid=1005(gamers) groups=1002(bob),1005(gamers)

# -p 옵션을 주어야 euid 가 설정된다.
bob@box:/tmp/priv$ ./bash -p
bash-4.3$ id
uid=1002(bob) gid=1005(gamers) euid=1004(alice) groups=1002(bob),1005(gamers)
```

-v | verbose

실행을 위해 읽은 명령을 화면에 표시해 줍니다.

vi

vi 스타일 line editing 을 사용합니다.

-x | xtrace

명령 실행전에 매개변수확장, 명령치환, 산술확장이 완료된 결과를 보여줍니다. 디버깅에 유용하게 사용할 수 있습니다.

Shopt

설정된 옵션값은 `shopt` or `shopt -p` 명령으로 볼수가 있습니다. 옵션값을 `enable` 할때는 `shopt -s` 옵션명 , `disable` 할때는 `shopt -u` 옵션명 을 사용합니다.

autocd

디렉토리 이동을 위해 `cd` 명령을 사용할 필요없이 디렉토리명만 입력하면 됩니다.

cdable_vars

가령 Music 디렉토리가 있고 AA 변수값이 Music 이면 `cd AA` 할수 있습니다.

cdspell

`cd` 명령을 사용할때 디렉토리 이름의 `transposed characters`, `a missing character`, `one character too many` 같은 오류를 자동으로 수정해줍니다.

checkhash

프롬프트에서 한번 실행한 명령은 다음에 재실행시 다시 명령을 `$PATH` 에서 검색할 필요없이 빠르게 실행하기 위해 내부적으로 `hash table` 에 경로를 저장해둡니다. 현재 저장되어 있는 내용은 `hash` 명령으로 볼 수 있는데요. 만약에 저장되어 있는 명령이 다른 위치로 옮겨지게 되면 (`$PATH` 내로 옮겨지더라도) 다음에 명령 사용시 `No such file or directory` 오류가 납니다. 이 옵션을 사용하면 `hash table` 에있는 명령이 존재하지 않을 경우 다시 `$PATH` 검색을 하게 합니다.

checkjobs

프롬프트에서 `exit` 명령을 실행했을때 `background job table` 에 `stopped` 상태에 있는 `job` 이 있으면 `There are stopped jobs` 메시지를 통해 알려줍니다. 하지만 `running` 상태에 있는 `job` 은 제외 되는데요. 이 옵션을 설정하면 `running` 상태에 있는 `job` 들도 메시지를 통해 알려줍니다. 연이어 `exit` 명령을 내리면 세션을 종료하는데 이때 `stopped` 상태에 있는 `job` 들은 함께 종료되고 `running` 상태에 있는 `job` 들은 `background` 로 실행을 계속하게 됩니다.

```
$ ping 192.168.1.1 &
[1] 1736
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.

$ shopt -s checkjobs

$ exit
exit
There are running jobs.      # running job 도 메시지를 통해 알려준다.
[1]+  Running                ping 192.168.1.1 &
```

checkwinsize

명령실행후 window size 를 체크하여 `$LINES` `$COLUMNS` 변수값을 갱신합니다.

cmdhist

multiple-line 명령을 작성할 경우 명령 줄들이 각각 다른 history 번호로 할당돼서 다음에 재사용하기가 어려운데 이 옵션을 사용하면 newline 을 `;` 로 치환해서 저장해 줍니다. `lithist` 옵션과 같이 사용하면 newline 도 그대로 유지됩니다.

compat31

Bash 3.1 호환모드

compat32

Bash 3.2 호환모드

compat40

Bash 4.0 호환모드

compat41

Bash 4.1 호환모드

compat42

Bash 4.2 호환모드

complete_fullquote

direxpend

경로명에 변수를 사용하면 자동완성 시에 확장됩니다.

dirspell

경로명 자동완성시 철자가 틀릴 경우 수정해 줍니다. `direxpend` 옵션과 같이 사용.

dotglob

이 옵션을 사용하면 `.filename` 와 같이 점으로 시작하는 파일도 매칭에 포함시킵니다.

execfail

script 실행중에 `exec` 명령이 실패하면 에러메시지와 함께 script 실행이 종료됩니다. 이 옵션을 사용하면 에러메시지는 기존과 같이 출력되나 script 종료는 되지 않습니다.

expand_aliases

alias 의 사용여부를 설정합니다. 기본적으로 interactive shell 에서는 `on` 이고 non-interactive shell 에서는 `off` 입니다.

extdebug

debugging 모드를 활성화 합니다.

- `declare -F` 함수이름 을 사용하면 해당 함수가 위치한 파일이름과 라인번호를 알려줍니다.
- `DEBUG trap` 함수가 0 이 아닌 값을 리턴하면 다음에 오는 명령을 스킵하고 실행하지 않습니다.

```

shopt -s extdebug

debug_trap() {
    echo DEBUG MESSAGE
    return 0          # return 값을 0, 1 로 바꾸어가며 테스트
}

trap 'debug_trap' DEBUG

echo HELLO 1
echo HELLO 2
echo HELLO 3

##### return 0 실행 #####
DEBUG MESSAGE
HELLO 1          # 정상적으로 debug message 와함께 다음명령이 실행됨
DEBUG MESSAGE
HELLO 2
DEBUG MESSAGE
HELLO 3

##### return 1 실행 #####
DEBUG MESSAGE
DEBUG MESSAGE   # 다음에 오는 명령이 스킵됨
DEBUG MESSAGE

```

- DEBUG trap 함수에서 2 를 리턴하면 현재 실행 중인 함수나 source 한 스크립트에서 리턴합니다.
- BASH_ARGC 와 BASH_ARGV 값을 사용할 수 있습니다.

extglob

Globbering 에서 extended pattern matching 을 사용할 수 있습니다.

extquote

Parameter expansion 에서 `'$string'` and `"$string"` 를 사용할수 있게합니다. 디폴트 입니다.

```
$ shopt -u extquote ; var=

$ echo "${var:-$"translate me"}"
me
$ echo "${var:-$'ab\ncd'}"
$'ab\ncd' # 맨앞에 $ 는 프롬프트가 아님

$ shopt -s extquote

$ echo "${var:-$"translate me"}"
translate me
$ echo "${var:-$'ab\ncd'}"
ab
cd
```

failglob

globbing 매칭 실패시 오류메시지와 함께 `$?` 값으로 `1` 을 리턴합니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class     쉘 스크립트 테스트.txt WriteObject.java

$ echo *.sh
*.sh

$ echo $?
0

$ shopt -s failglob

$ echo *.sh
bash: no match: *.sh

$ echo $?
1
```

force_ignore

`$FIGIGNORE` 변수에 파일 확장자를 `:` 로 분리해서 등록해 놓으면 filename completion 시에 제외됩니다.

globstar

****** 문자를 이용하여 하위 디렉토리 까지 recursive 매칭을 할수있습니다.

```
$ echo **          # 모든 파일, 디렉토리
$ echo **/         # 모든 디렉토리
$ echo **/*.sh     # 확장자가 .sh 인 모든파일
```

globasciiranges

C locale 이 아닐 경우 `[a-c]` 와 같이 `-` 문자를 이용한 range 매칭시에 기본적으로 대, 소문자 구분을 하지 않습니다. 다시 말해 `[aAbBcCc]` 와 같은 의미가 되는데요. 이 옵션을 설정하면 대, 소문자 구분을 합니다.

gnu_errfmt

shell 에러 메시지를 "standard GNU error message format" 으로 표시합니다.

histappend

shell 을 exit 할때 `HISTFILE` 변수에 설정돼 있는 파일에 현재 history list 를 append 합니다. off 이면 overwrite 합니다.

histreedit

프롬프트 에서 `!` 문자를 이용한 history 확장을 사용할때 확장이 실패할 경우 입력했던 내용이 없어지지 않고 다시 수정할수 있는 기회를 줍니다.

histverify

프롬프트 에서 `!` 문자를 이용한 history 확장을 사용할때 확장된 명령을 바로 실행하지 않고 필요할시 수정할수 있게 enter 를 칠 기회를 줍니다.

hostcomplete

/etc/hosts 파일에 등록되어있는 호스트이름 들의 completion 을 제공합니다.

huponexit

interactive login shell 이 exit 할때 모든 jobs 들에 `HUP` 시그널을 보냅니다.

interactive_comments

프롬프트에서 `#` 코멘트를 사용할수 있습니다.

lastpipe

| 를 사용해서 명령을 실행할때 단점은 연결된 명령들이 각각의 subshell 에서 실행이 되어 명령이 종료된 후에는 설정한 변수값을 읽을수가 없습니다. lastpipe 옵션은 | 로 연결된 명령들중에 마지막 명령을 현재 shell 에서 실행하게 해줍니다. 이옵션을 사용하기 위해서는 job control 이 disable 되어있어야 합니다. non-interactive shell 인 script 실행시에는 기본적으로 disable 됩니다.

```
# echo hello, read var 가 subshell 에서 실행되어 종료후 echo $var 로 값을 읽을수 없다.
$ echo hello | read var
$ echo $var
$

$ set +o monitor          # job control 을 disable
$ shopt -s lastpipe

# 마지막 명령인 read var 는 현재 shell 에서 실행되어 echo $var 로 값을 읽을수 있다.
$ echo hello | read var ; echo $var
hello
```

lithist

명령 history 옵션으로 multiple-line 명령을 작성할 경우 newline 유지해 줍니다.

login_shell

readonly 옵션으로 login shell 일경우 설정됩니다.

mailwarn

시스템에서 메일도착을 체크한 이레로 메일을 읽은적이 있으면 "The mail in mailfile has been read" 메시지를 표시해 줍니다.

no_empty_cmd_completion

empty line 에서 자동완성 시도시 \$PATH 를 검색하지 않습니다.

nocaseglob

globbing 시에 대, 소문자 구분을 하지 않습니다.

nocasematch

패턴매칭 시에 대, 소문자 구분을 하지 않습니다.

nullglob

globbing 시에 매칭되는 파일이 없을 경우 null 을 리턴합니다.

```
$ ls
2013-03-19 154412.csv  ReadObject.java      WriteObject.class
ReadObject.class      셸 스크립트 테스트.txt  WriteObject.java

$ echo *.sh
*.sh                # 매칭되는 파일이 없을 경우 패턴을 그대로 리턴

$ shopt -s nullglob

$ echo *.sh
$                  # 매칭되는 파일이 없을 경우 null 을 리턴
```

progcomp

명령 자동완성 기능을 사용 가능하게 합니다. 디폴트 설정입니다.

promptvars

프롬프트 스트링에 사용된 특수문자 들의 디코딩이 끝나고 나서 변수확장, 명령치환, 산술확장, quote 제거 적용을 받습니다. 기본 설정입니다.

restricted_shell

bash 가 restricted shell 일 경우 설정되며 readonly 옵션입니다.

shift_verbose

shift 명령이 사용될때 shift count 가 positional parameters 수를 넘어서면 에러메시지를 프린트 합니다.

sourcepath

source 명령이 파일을 찾을때 `$PATH` 를 사용하게 합니다. 디폴트 설정입니다.

xpg_echo

echo 명령이 single, double quote 모두에서 백슬래시를 이용한 escape sequences 을 사용하게 합니다.

Bourne Shell Variables

CDPATH

자주 사용하는 디렉토리를 `:` 로 분리해 등록해 놓으면 `cd` 할 때 전체 경로를 입력할 필요가 없고 `tab` 키를 이용한 자동완성을 사용할 수 있습니다.

```
$ echo $CDPATH
.: /home/razvan/school/current: /home/razvan/projects

$ pwd
/home/razvan

$ cd so
/home/razvan/school/current/so

$ cd p2p-next
/home/razvan/projects/p2p-next
```

HOME

현재 사용자 홈디렉토리

IFS

Internal Field Separator. 단어분리 (word splitting) 시에 이 값을 기준으로 단어가 분리됩니다. `read` 명령으로 읽어들이는 라인을 필드로 분리할때, 매개변수 확장을 통해 `array` 에 원소들을 입력할때도 사용됩니다. 기본값은 `space`, `tab`, `newline` 3 개로 `IFS` 변수가 `unset` 됐을때도 적용됩니다. `IFS` 값이 `null` 이면 단어분리가 일어나지 않습니다.

```
$ echo -n "$IFS" | od -a
00000000 sp ht nl
```

MAIL

`$MAILPATH` 변수가 설정되었지 않을때 `mailbox` 파일이나 `maildir` 디렉토리를 등록해 놓으면 메일이 왔을때 알려줍니다.

MAILPATH

`mailbox` 파일들을 `:` 로 분리하여 등록해놓으면 메일이 왔을때 알려줍니다.

OPTARG

`getopts` 명령에서 사용되는 변수로 현재 옵션값을 나타냅니다.

OPTIND

`getopts` 명령에서 사용되는 변수로 다음에 처리할 옵션 index 를 나타냅니다.

PATH

명령이 위치한 디렉토리를 `:` 로 분리하여 등록해 놓으면 순서대로 명령을 찾습니다.

PS1

Default interaction prompt 입니다. 프롬프트 모양을 변경하기 위해 여러가지 특수문자를 사용할 수 있습니다.

```
# \u - 아이디
# \h - 호스트이름
# \w - 현재 디렉토리의 전체경로

bash-3.2$ export PS1="\u@\h \w> "

ramesh@dev-db ~-> cd /etc/mail
ramesh@dev-db /etc/mail>
```

PS2

Continuation interactive prompt 입니다. 여러줄의 명령을 입력할때 나타납니다.

```
# 다음줄부터 PS2 값으로 설정된 '>' 문자가 보입니다.

ramesh@dev-db ~-> myisamchk --silent --force --fast --update-state \
> --key_buffer_size=512M --sort_buffer_size=512M \
> --read_buffer_size=4M --write_buffer_size=4M \
> /var/lib/mysql/bugs/*.MYI
```

Bash Variables

BASH

현재 실행된 shell 프로그램의 전체경로 입니다.

BASHOPTS

`shopt` 명령으로 설정한 옵션들이 `:` 로 분리되어 저장됩니다.

BASHPID

현재 bash process id 입니다. subshell 에서 다른값을 가집니다.

BASH_ALIASES

`alias` 명령에 의해 설정된 associative array 입니다.

BASH_ARGC

`shopt -s extdebug` 옵션이 설정돼 있을 경우에 사용되며 현재 실행되는 함수의 인자수 call stack 을 나타내는 array 변수입니다. 가령 `func1 11 22` 에서 `func2 33 44 55` 를 호출하여 실행중이라면 `${BASH_ARGC[@]}` 값은 `3 2` 가 됩니다.

BASH_ARGV

`shopt -s extdebug` 옵션이 설정돼 있을 경우에 사용되며 현재 실행되는 함수의 인자값 call stack 을 나타내는 array 변수입니다. 주의할 것은 표시되는 인자의 순서인데 가령 `func1 11 22` 에서 `func2 33 44 55` 를 호출하여 실행중이라면 `${BASH_ARGV[@]}` 값은 `55 44 33 22 11` 가 됩니다.

BASH_CMDS

`hash` builtin 명령에서 사용하는 associative array 입니다. 값을 추가하거나 삭제하면 동일하게 hash 테이블에 반영됩니다.

BASH_COMMAND

현재 명령을 나타냅니다.

```
$ trap 'echo \"${BASH_COMMAND}\" 명령이 실패하였습니다. 에러코드: $?' ERR
$ asdfgh
asdfgh: command not found
"asdfgh" 명령이 실패하였습니다. 에러코드: 127
```

BASH_COMPAT

Bash compatibility level 을 설정할수 있습니다. 설정을 안했을 경우는 현재버전으로 됩니다.

BASH_ENV

터미널을 새로 열때마다 `interactive shell` 이 시작되면서 `.bashrc` 파일이 실행됩니다. 비슷하게 `non-interactive shell` 인 스크립트 파일이 실행될때 마다 실행되는 파일을 이 변수에 등록합니다. 스크립트 파일용 `.bashrc` 인 셈입니다.

BASH_EXECUTION_STRING

`bash -c` 형식으로 실행시 사용된 명령구문을 나타냅니다.

BASH_LINENO

현재 실행중인 함수의 caller 의 라인넘버 `call stack` 을 나타내는 `array` 변수입니다. 가령 `a()` 함수의 라인번호 10 에서 `b()` 함수를 호출하고 `b()` 함수의 라인번호 20 에서 `c()` 함수를 호출하여 현재 실행중이라면 `${BASH_LINENO[0]}` 는 caller 라인번호인 20 이되고 `${BASH_LINENO[1]}` 는 10 이 되는 식입니다. `${BASH_SOURCE[$i+1]}` 와 같이 사용하면 caller 의 정확한 위치를 알수있습니다.

BASH_REMATCH

`[[keyword 의 =~` 연산자를 이용한 패턴매칭시에 소괄호 `()` 를 이용해 캡처를 할수있습니다. 이때 전체매칭은 `$BASH_REMATCH[0]` 에 첫번째 `()` 매칭은 `$BASH_REMATCH[1]` ... 에 각각 저장됩니다.

```
#!/bin/bash

regex=$1
string=$2

if [[ $string =~ $regex ]]; then
    echo "$string matches"
    i=0
    n=${#BASH_REMATCH[@]}
    while [[ $i -lt $n ]]
    do
        echo "  capture[$i]: ${BASH_REMATCH[$i]}"
        (( i++ ))
    done
else
    echo "$string does not match"
fi

##### 실행 #####

$ ./regex1.sh 'a(b{2,3})([xyz])c' aabbxcc
aabbxcc matches
capture[0]: abbxc
capture[1]: bb
capture[2]: x
```

BASH_SOURCE

현재 실행되는 함수가 위치한 파일이름의 call stack 을 나타내는 array 변수입니다. 가령 fileA.sh 의 a() 함수가 fileB.sh 의 b() 함수를 호출하고 b() 함수가 fileC.sh 의 c() 함수를 호출하여 현재 실행중이라면 `${BASH_SOURCE[0]}` 은 fileC.sh 이되고 `${BASH_SOURCE[1]}` 은 fileB.sh 이 ... 되는 식입니다.

BASH_SUBSHELL

subshell 의 중첩된수를 나타냅니다.

```
$ echo $BASH_SUBSHELL
0

$ (echo $BASH_SUBSHELL;(echo $BASH_SUBSHELL))
1
2
```

BASH_VERSION

bash 버전정보를 나타내는 array 변수 입니다.

BASH_VERSION

bash 버전정보를 나타냅니다.

BASH_XTRACEFD

xtrace 메시지만 특정 FD (file descriptor) 로 보낼때 사용합니다. `set -o xtrace` 옵션을 사용하여 xtrace 를 하면 기본적으로 `stderr` 로 메시지가 출력됩니다. 그러므로 실행되는 명령들 중에서도 에러 메시지가 발생하면 둘이 섞이게 되는데 이 변수를 이용하면 xtrace 메시지만 특정 FD 로 보낼 수 있습니다.

```
----- test.sh -----
#!/bin/bash

exec 3> xtrace.txt      # FD 3번을 생성하고 xtrace.txt 로 출력
BASH_XTRACEFD=3        # trace 메시지를 FD 3번 으로 보냄

set -x

date -%Y                # error 메시지 발생
AA=100
BB=200
CC=`expr $AA + $BB`
date -%Y                # error 메시지 발생

set -
-----

# 오류 메시지는 기존대로 stderr 로 출력
$ ./test.sh
date: invalid option -- '%'
Try 'date --help' for more information.
date: invalid option -- '%'
Try 'date --help' for more information.

$ cat xtrace.txt
+ date -%Y
+ AA=100
+ BB=200
++ expr 100 + 200
+ CC=300
+ date -%Y
+ set -
```

CHILD_MAX

종료된 child process 의 상태값을 얼마나 기억할지를 나타냅니다.

COLUMNS

현재 터미널의 컬럼수를 나타냅니다. `select` 명령에 의해 사용됩니다.

COMP_CWORD

자동완성 함수에서 사용되는 변수로 프롬프트에서 타입중인 명령에서 현재 커서가 위치한 단어의 index 를 나타내며 `${COMP_WORDS[COMP_CWORD]}` 로 스트링값을 구할수 있습니다.

COMP_LINE

자동완성 함수에서 사용되는 변수로 프롬프트에서 작성중인 명령의 현재 라인 내용을 나타냅니다.

COMP_POINT

자동완성 함수에서 사용되는 변수로 프롬프트에서 명령을 작성할때 현재 커서가 위치한 문자 의 index 를 나타냅니다. 커서가 명령 라인의 맨 끝에 있다면 그 값은 `${#COMP_LINE}` 와 같습니다.

COMP_TYPE

자동완성 함수에서 사용되는 변수로 현재 시도된 completion type 을 나타냅니다.

COMP_KEY

자동완성 함수에서 사용되는 변수로 completion 시도시 사용된 키값을 나타냅니다.

COMP_WORDBREAKS

자동완성 함수에서 사용되는 변수로 프롬프트에서 현재까지 작성한 명령을 단어들로 분리할때 이 변수에 있는 값들을 기준으로 합니다. unset 할경우 기능을 상실하므로 추후에 다시 값을 대입해도 적용되지 않습니다. 기본값은 다음과 같습니다.

```
0000000 sp " ' > < = ; | & ( :
          20 22 27 3e 3c 3d 3b 7c 26 28 3a
```

COMP_WORDS

자동완성 함수에서 사용되는 변수로 프롬프트에서 현재까지 타입한 명령의 단어들을 담고있는 array 입니다.

COMPREPLY

자동완성 함수에서 사용되는 `array` 변수로 여기에 입력된 단어들이 `tab` 키를 이용해 자동완성을 시도할때 보여지게 됩니다.

COPROC

`coproc` builtin 명령을 사용할때 생성되는 file descriptors 를 담고있는 `array` 입니다.

DIRSTACK

현재까지 방문한 디렉토리 정보를 담고있는 `array` 입니다. `dirs` builtin 명령에서 보여주는 내용과 같습니다.

EMACS

shell 을 시작할때 이변수에 `t` 값이 설정되 있으면 emacs shell buffer 에서 실행되는것으로 간주하여 line editing 이 disable 됩니다.

ENV

`$BASH_ENV` 와 같습니다. shell 이 POSIX 모드로 실행될때 사용됩니다.

EUID

numeric effective user id 로 readonly 입니다. set uid 비트가 설정된 프로그램을 실행했을때 바뀌게 됩니다

FCEDIT

`fc -e` 명령을 사용할때 적용되는 기본 에디터 입니다.

FIGNORE

파일 확장자를 `:` 로 분리해서 등록해 놓으면 filename completion 시에 제외됩니다. 예)
`.o:.class`

FUNCNAME

현재 실행되는 함수이름의 call stack 을 나타내는 array 변수입니다. 가령 a() 함수가 b() 함수를 호출하고 b() 함수가 c() 함수를 호출하여 현재 실행중이라면 이때 `${FUNCNAME[0]}` 은 c() 가되고 `${FUNCNAME[1]}` 은 b() 가 ... 되는 식입니다.

FUNCNEST

maximum function nesting level 을 설정할수 있습니다. 설정된 nesting level 을 넘어서면 명령실행이 중단 됩니다.

GLOBIGNORE

globbing 에서 사용하는 패턴을 `:` 로 분리해서 등록해 놓으면 매칭에서 제외시킵니다.

GROUPS

현재 사용자가 속해있는 그룹을 나타내는 array 변수입니다.

HISTCMD

현재 명령의 history number 를 나타냅니다.

HISTCONTROL

명령 history 의 작동방식을 `:` 로 분리하여 설정할수 있습니다.

- **ignorespace** : space 로 시작하는 명령라인을 history 에 저장하지 않습니다.
- **ignoredups** : 이전 history 명령과 중복될경우 저장하지 않습니다.
- **ignoreboth** : ignorespace:ignoredups 와 같습니다.
- **erasedups** : 이전 모든 history 라인을 비교하여 중복된 history 를 제거한후 저장합니다.

HISTFILE

history 를 저장할 파일을 나타냅니다.

HISTFILESIZE

history 파일에 저장될 최대 라인수를 나타냅니다.

HISTIGNORE

history 리스트에 저장할때 제외시킬 명령패턴을 `:` 로 분리하여 등록합니다.

HISTSIZE

history 리스트에 기억될 최대 명령수를 나타냅니다. 디폴트 값은 500 입니다.

HISTTIMEFORMAT

history 번호에 이어 timestamp 를 붙일수 있습니다. history file 에도 저장됩니다. 예) `export HISTTIMEFORMAT="%Y%m%d %T "`

HOSTFILE

`/etc/hosts` 와 같은 포맷의 파일을 등록해 놓으면 hostname completion 시에 사용됩니다.

HOSTNAME

호스트 이름을 나타냅니다.

HOSTTYPE

호스트 타입을 나타냅니다. 예) `x86_64`

IGNOREEOF

`set -o ignoreeof` 일때 사용되는 변수로 터미널에서 연이어 몇번을 `ctrl-d` (EOF) 입력해야 exit 되는지 숫자를 지정합니다.

INPUTRC

Readline 라이브러리의 설정파일을 지정할수 있습니다. 디폴트는 `~/.inputrc` 입니다.

LANG

`LC_` 로 시작하는 변수로 설정하지 않은 로케일 카테고리에 대해서는 이변수의 값이 적용됩니다.

LC_ALL

`LC_` 와 `LANG` 으로 설정한 로케일 값을 overriding 합니다.

LC_COLLATE

sorting the results of filename expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within filename expansion and pattern matching 과 관련된 로케일 입니다.

LC_CTYPE

filename expansion and pattern matching 과 관련된 로케일 입니다.

LC_MESSAGES

`$" "` 를 사용한 스트링을 번역하는데 사용되는 로케일 입니다.

LC_NUMERIC

number formatting 에 관련된 로케일 입니다.

LINENO

Script 이나 shell function 실행시 현재 라인번호를 나타냅니다.

LINES

현재 터미널 라인수를 나타냅니다. `select` 명령에 의해 사용됩니다.

MACHTYPE

GNU cpu-company-system format 으로 시스템 정보를 나타냅니다.

MAILCHECK

메일체크 주기를 초단위로 설정합니다.

MAPFILE

`mapfile` 명령으로 라인들을 읽어들이때 변수를 지정하지 않으면 여기에 저장됩니다.

OLDPWD

`cd` 명령을 사용할때 바로 전 디렉토리를 나타냅니다.

OPTERR

`getopts` 명령에서 사용하는 변수로 0 으로 설정하면 에러메시지를 표시하지 않습니다. 기본값은 1 입니다. 스크립트가 실행될때마다 새로 설정되기 때문에 `getopts` 명령을 사용하는 스크립트 내에서 설정해야합니다.

OSTYPE

shell 이 실행되는 OS 환경을 나타내며 자동설정값 입니다. 예) linux-gnu

PIPESTATUS

파이프로 연결된 명령들이 종료했을때 명령들의 리턴값을 담고있는 array.

POSIXLY_CORRECT

`export POSIXLY_CORRECT=t` 해놓으면 bash process 가 posix mode 로 실행됩니다. 현재 shell 도 `set -o posix` 한것과 같이 posix mode 로 바뀝니다.

PPID

shell 의 parent process id 로 readonly 변수입니다.

PROMPT_COMMAND

`PS1` 프롬프트를 표시하기전에 등록된 명령을 실행합니다.

```
ramesh@dev-db ~-> export PROMPT_COMMAND="date +%k:%m:%S; date;"
19:06:44
Thu Jun 11 19:14:44 KST 2015
ramesh@dev-db ~->

19:06:47
Thu Jun 11 19:14:47 KST 2015
ramesh@dev-db ~->
```

PROMPT_DIRTRIM

프롬프트에 디렉토리 경로를 표시할때 이 변수에 설정된 값만큼만 표시하고 나머지 뒷부분은 ... 로 표시합니다.

PS3

`select` 명령에서 사용하는 프롬프트 입니다.

PS4

디버깅을 하기위해 `-x` | `set -o xtrace` 옵션을 설정하여 스크립트를 실행하면 명령 실행전에 매개변수 확장, 치환이 완료된 명령문을 보여줍니다. 이때 PS4 값이 설정돼 있으면 명령문 앞에 표시합니다. 기본값은 `+` 입니다.

```
# PS4 설정예
export PS4='+(${BASH_SOURCE[0]}:${LINENO}): ${FUNCNAME[0]}:${FUNCNAME[0]}(): '
```

PWD

`cd` 명령에 의해 현재 디렉토리 경로가 저장됩니다.

RANDOM

0 ~ 32767 사이의 random number 를 제공합니다.

READLINE_LINE

readline script 을 작성할때 사용되는 변수로 프롬프트에서 명령을 입력할때 현재라인의 내용을 나타냅니다.

READLINE_POINT

readline script 을 작성할때 사용되는 변수로 프롬프트에서 명령을 입력할때 현재 커서가 위치한 문자 의 index 를 나타냅니다.

REPLY

`read` 명령을 사용할때 값이 저장될 변수를 지정하지 않으면 이변수에 저장됩니다.

SECONDS

shell 이 시작한이래 경과한 시간 또는 script 가 실행을 시작한 이후 경과한 시간을 초단위로 나타냅니다. 임의로 값을 입력하면 그값으로부터 더해집니다.

SHELL

현재 쉘 프로그램의 전체경로 입니다.

SHELLOPTS

`set` 명령으로 설정한 옵션들이 `:` 로 분리되어 저장됩니다.

SHLVL

`bash` process 의 중첩된 수를 나타냅니다. 가령 현재 쉘에서 다시 `bash` 를 실행하면 `SHLVL` 은 2 가됩니다. `shell script` 를 실행했을때, `bash -c 'echo $SHLVL'` 형식으로 명령을 실행했을때 값이 올라갑니다. 다시말해 `child process` 일 경우에 해당하므로 `subshell` 에는 적용되지 않습니다. `subshell` 의 중첩된 수는 `BASH_SUBSHELL` 을 통해 알 수 있습니다.

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
```

TIMEFORMAT

`time` 키워드에서 결과를 출력할때 사용되는 포맷 스트링 입니다.

TMOUT

`read` , `select` 명령사용시 입력대기 시간을 초단위로 설정할수 있습니다. 대기시간을 초과하면 입력을 중단하고 다음명령으로 진행합니다. 터미널에서 설정할경우 해당 시간동안 명령 입력이 없으면 `exit` 합니다.

TMPDIR

`shell` 이 사용하는 임시 디렉토리를 지정할수 있습니다.

UID

현재 사용자의 `numeric real user id` 로 `readonly` 변수 입니다. 시스템에 `login` 했을때 `id` 를 말합니다.

Positional Parameters

- 스크립트 파일을 실행할 때
- function 을 호출할 때
- 스크립트 파일을 source 할 때

인수를 주게되면 해당 스크립트 및 함수 내에서 positional paramters 가 자동으로 설정됩니다. 첫 번째 인수는 `$1` , 두번째는 `$2` ... 식으로 할당되며 scope 은 local 변수와 같습니다. `sh` 과 같이 array 를 사용할 수 없는 환경에서 활용할 수 있습니다.

\$0

스크립트 파일 이름을 나타냅니다.

`bash -c` 형식으로 실행했을 경우는 첫번째 인수를 가리킵니다.

```
$ bash -c 'echo $0, $1, $2' 11 22 33
11, 22, 33
```

\$#

`$0` 을 제외한 전체 인수의 개수를 나타냅니다.

\$1, \$2, \$3 ...

각 인수들을 나타냅니다.


```
----- test.sh -----
#!/bin/sh

echo number of arguments = $#

echo \"$0\" = \"$0\"

echo \"$1\" = \"$1\"
echo \"$2\" = \"$2\"
echo \"$3\" = \"$3\"
```

```
##### output #####
```

```
$ ./test.sh 11 22 33
number of arguments = 3
$0 = ./test.sh
$1 = 11
$2 = 22
$3 = 33
```

\$@ , \$*

`$@` , `$*` 는 positional parameters 전부를 포함합니다. array 에서 사용되는 `@` , `*` 기호와 의미가 같다고 볼 수 있습니다. 변수를 quote 하지 않으면 단어분리에 의해 두 변수의 차이가 없지만 quote 을 하게 되면 `"$@"` 의 의미는 `"$1"` `"$2"` `"$3"` ... 와 같게되고 `"$"` 의 의미는 `"$1c$2c$3 ... "` 와 같게됩니다. (여기서 c 는 IFS 변수값의 첫번째 문자 입니다.)

```

----- test.sh -----
#!/bin/sh

echo \@ : \@
echo \$* : \$*

echo '===== \@' ====='
for v in "\@"; do          # 또는 for v do ...
    echo "$v"
done

echo '===== \$*' ====='
for v in "\$*"; do
    echo "$v"
done

##### output #####

$ ./test 11 22 33
\@ : 11 22 33
\$* : 11 22 33
===== \@ =====
11
22
33
===== \$* =====
11 22 33

```

set

set 명령은 보통 shell 옵션을 설정할때 사용하지만 스크립트 내에서 positional paramters 를 설정하거나 삭제할때도 사용됩니다.

- `set -- 11 22 33` 또는 `set - 11 22 33`
`$1` `$2` `$3` 값을 각각 11 22 33 으로 설정합니다.
- `set --`
현재 설정되어 있는 positional parameters 를 모두 삭제합니다.

```

----- test.sh -----
#!/bin/sh

# set 명령을 이용하여 script 내에서 positional parameters 를 설정
set -- 11 22 33

echo number of arguments = $#

echo \$0 = "$0"

echo \$1 = "$1"
echo \$2 = "$2"
echo \$3 = "$3"

##### output #####

$ ./test.sh
number of arguments = 3
$0 = ./test.sh
$1 = 11
$2 = 22
$3 = 33

```

shift

shift [n]

shift 명령은 현재 설정되어 있는 positional parameters 를 좌측으로 `n` 만큼 이동시킵니다. 결과로 `n` 개의 positional parameters 가 삭제 됩니다.

```

$ set -- 11 22 33 44 55

$ echo $@
11 22 33 44 55

$ shift 2

$ echo $@
33 44 55

```

\$IFS

`$IFS` 변수와 `set` 명령을 이용하여 스트링에서 필드를 분리해낼 수 있습니다.

```
#!/bin/sh

line="11:22:33:44:55"

set -f; IFS=:          # globbing 을 disable
set -- $line          # IFS 값에 따라 필드를 분리하여 positional parameters 에 할당
set +f; IFS=$' \t\n'

echo number of fields = $#
echo field 1 = "$1"
echo field 2 = "$2"

shift 3
echo \${@} = "${@}"

##### output #####

number of fields = 5
field 1 = 11
field 2 = 22
${@} = 44 55
```

Substring expansion

Substring expansion 은 bash 에서만 사용할수 있습니다.

```
$ set -- 11 22 33 44 55

$ echo ${@:3}
33 44 55

$ echo ${@:2:2}
22 33
```

Special Parameters

\$?

바로 이전에 실행된 명령의 종료상태 값을 나타냅니다.

\$\$

현재 shell process id 를 나타냅니다. subshell 에서도 같은 값을 가집니다.

\$!

`&` 메타문자를 이용하여 가장최근에 background 로 실행된 process id 를 나타냅니다.

\$_

이전 명령에서 사용된 마지막 인수를 값으로 가집니다. 사용된 인수가 없으면 명령을 리턴합니다.

\$-

Set 명령에 의해 현재 shell 에 enable 되있는 option flags 을 보여줍니다.

Command Aliases

프롬프트 상에서 자주 사용하는 명령문이나 복잡하고 긴 명령문에 별칭을 붙여서 사용할 수 있습니다. 별칭이 실행될 때는 설정한 명령문이 그대로 치환되어 실행됩니다. 별칭 이름으로 `/`, `\`, `$`, ```, `=`, `"`, `'`, `,` shell 메타문자는 사용할 수 없습니다.

```
alias cp='cp -i'
alias mv='mv -i'
alias grep='grep --color=auto'
alias p4='ps axfo user,ppid,pid,pgid,sid,TTY,stat,args'

alias geturl='python /some/cool/script.py'
$ geturl http://example.com/excitingstuff.jpg

# '|' ';' '{ }' '(' ')' 를 이용하여 여러 명령을 설정할 수도 있습니다.
alias name1='command1 | command2 | command3'
alias name2='command1; command2; command3'
alias name3='{ command1; command2; command3 ;}'
```

- 설정할 경우 어떤 명령이나 함수 보다도 우선순위가 높습니다.
심지어 shell 키워드, 메타문자도 alias 하여 사용할 수 있습니다.
- alias 는 명령문을 alias 합니다. 중간에 사용되는 인수를 alias 해서 사용할수는 없습니다.
- shell 프롬프트 상에서만 사용할 수 있습니다.
non-interactive shell 인 script 실행시에는 기본적으로 disable 됩니다.
(`shopt -s expand_aliases` 옵션으로 컨트롤할 수 있습니다.)
- 설정한 alias 는 subshell 에서도 동일하게 사용할 수 있습니다.
- child process 에서는 사용할 수 없습니다.
(가령 `vi` 에디터에서 외부명령을 실행할때 사용할 수 없습니다.)
- 함수를 정의할때 alias 를 사용하면 자동으로 expand 됩니다.

```
$ alias mv='mv -i'

$ mymv() { mv "$1" "$2" ;}

$ declare -f mymv
mymv ()
{
    mv -i "$1" "$2"          # mv alias 가 확장되어 함수가 정의된다.
}
```

- BASH_ALIASES associative array 를 통해서 alias 된 명령을 구할 수 있습니다.

```
$ alias p3='ps afo user,ppid,pid,pgid,sid,tty,stat,args'

$ echo "${BASH_ALIASES[p3]}"
ps afo user,ppid,pid,pgid,sid,tty,stat,args
```

- 함수처럼 alias 에 인수를 전달할 수는 없습니다.
- 설정한 alias 를 escape 하려면 이름앞에 `\` 를 붙여 사용하거나 quote 하면됩니다. 또한 `command` , `builtin` 명령을 사용할 수도 있습니다.
- 설정한 alias 를 삭제하려면 `unalias` 명령을 사용합니다.

Command History

터미널을 열었을때 실행되는 interactive shell 에서는 명령 history 기능을 사용할 수 있습니다. 명령 history 는 이전에 한번 사용했던 명령을 다시 타입 할 필요 없이 재사용할 수 있게 해줍니다. 터미널 별로 history list 가 생성되며 사용한 명령들이 목록에 추가됩니다. 터미널 종료 시에는 `shopt -s histappend` 옵션 설정에 따라 `$HISTFILE` 에 현재 history list 가 저장됩니다.

Non-interactive shell 인 스크립트 실행시에는 기본적으로 disable 됩니다.

명령 라인을 찾는 방법

프롬프트에서 `history` 명령을 실행하면 현재 history list 에 있는 목록들이 번호와 함께 표시됩니다. 이 번호는 해당 명령 라인을 지정할때 사용됩니다. history 확장에는 기본적으로 `!` 문자를 사용하며 `!` 문자 뒤에 공백이 오면 logical NOT 으로 사용되고 그렇지 않으면 history 확장으로 해석됩니다.

!n

n 번 명령을 리턴합니다.

```
$ !2145
lsb_release -d
Description:    Ubuntu 15.04

$ echo command is : !2145
command is : lsb_release -d
```

!!

바로 이전 명령을 나타냅니다.

```
$ history
...
 3  ps af
 4  cat README.md
 5  find . -type f -name *.log -size +10M -exec rm -f {} \;
$ !!
find . -type f -name *.log -size +10M -exec rm -f {} \;
```

!-n

이전 n 번째 명령을 나타냅니다.

```
$ history
...
 3 date
 4 ps af
 5 find . -name "*.log"

$ !-1
find . -name "*.log"

$ !-2
ps af

$ !-3
date
```

!string

명령 이름이 string 으로 시작하는 가장 최근 명령을 찾습니다.

```
$ history
...
 3 find . -type f -name *.log -size +10M -exec rm -f {} \;
 4 find . -name "*.log"
 5 ps af

$ !fi
find . -name "*.log"
```

!?string[?]

이건 명령이름을 검색하는게 아니고 전체 명령라인 중에 string 이 포함돼 있는지를 찾습니다. 가장 최근에 매칭이 되는 라인을 리턴합니다.

```
$ history
...
 3 find . -name "*.tmp" -o -name "*.old"
 4 find . -name "*.tmp"
 5 find . -name "*.log"

$ !?tmp
find . -name "*.tmp"

$ !?tmp? -exec rm -f {} \;
find . -name "*.tmp" -exec rm -f {} \;
```

찾은 명령 라인에서 원하는 인수를 지정하는 방법

명령 라인을 지정한 후에 `:` 문자를 붙인 후 원하는 인수들을 지정할 수 있습니다. 0 번은 명령을 나타내고 이후 인수들은 1, 2, 3 ... 번으로 지정할 수 있습니다.

```
command arg1 arg2 arg3 arg4 arg5
```

```
$ !com:0      # 0 번은 명령
command
```

```
$ !com:1      # 1 번은 첫번째 인수
arg1
```

```
$ !com:2      # 2 번은 두번째 인수
arg2
```

```
$ !com:2-4    # '-' 를 이용해 범위를 지정
arg2 arg3 arg4
```

```
$ !com:*      # '*' 는 모든 인수를 나타냅니다.
arg1 arg2 arg3 arg4 arg5
```

```
$ !com:3*     # '3*' 은 3 번째 인수부터 끝까지
arg3 arg4 arg5
```

```
$ !com:^      # '^' 는 첫번째 인수를 나타냅니다.
arg1          # !com:1 과 같고, ':' 없이 !com^ 와 같이 사용할 수도 있습니다.
```

```
$ !com:$      # '$' 는 마지막 인수를 나타냅니다.
arg5          # ':' 없이 !com$ 와 같이 사용할 수도 있습니다.
```

명령라인 지정을 생략하면 바로 이전 명령이 사용됩니다.

```
$ history
...
 3  date
 4  ps af
 5  echo 11 22 33 44

$ !:0
echo

$ !:1
11

$ !*
11 22 33 44

$ !^
11

$ !$
44
```

지정한 라인, 인수에 modifiers 적용하기

라인을 지정한 뒤, 또는 인수들을 지정한 뒤에 `:` 문자를 붙인 후 modifiers 를 적용시킬 수 있습니다.

s/old/new/

지정한 명령라인에서 old 에 해당하는 스트링을 new 로 변경합니다. new 부분에 `&` 문자가 오면 old 로 대체됩니다. 기본적으로 처음에 매칭되는 하나만 적용되며 모두에 적용하려면 `g` 옵션을 추가합니다. 직전 명령에서 사용했던 치환을 다시 사용하려면 `&` 문자를 이용할 수 있습니다.

```
find . -name "2010-09-*.log" -o -name "2011-01-*.log"
...

# 처음 하나만 변경된다
$ !find:s/log/txt/
find . -name "2010-09-*.txt" -o -name "2011-01-*.log"

# 'g' 옵션을 추가하면 모두 변경된다.
$ !find:gs/log/txt/
find . -name "2010-09-*.txt" -o -name "2011-01-*.txt"

# '&' 는 직전 명령에서 사용했던 패턴을 의미합니다.
$ !find:g&
find . -name "2010-09-*.txt" -o -name "2011-01-*.txt"
```

파일 경로명에서 파일명 분류하기

```
cat /home/foo/bar/tmp/readme.txt
...

$ !cat:h                # 'h' 는 head 를 의미
cat /home/foo/bar/tmp    # cat 명령이 함께 포함됨

$ !cat:h/README
cat /home/foo/bar/tmp/README

$ !cat:1:h              # ':1' 인수에서 head 를 구함
/home/foo/bar/tmp

$ head !cat:1:h/README
head /home/foo/bar/tmp/README

$ !cat:t                # 't' 는 tail 을 의미
readme.txt

$ tail !cat:t
tail readme.txt
```

파일 경로명에서 확장자 분류하기

```
cat /home/foo/bar/tmp/readme.txt
...

$ !cat:r                # 확장자를 remove
cat /home/foo/bar/tmp/readme

$ !cat:r.old
cat /home/foo/bar/tmp/readme.old

$ !cat:e                # 확장자 (extension) 만 구함.
.txt
```

결과물 quoting 하기

```
cat a1.txt a2.txt a3.txt
...

$ !cat:q                # 전체 라인이 quote 된다.
'cat a1.txt a2.txt a3.txt'

$ !cat:x                # space 로 분리되어 각각 quote 된다.
'cat' 'a1.txt' 'a2.txt' 'a3.txt'
```

실행 금지하기

history 확장이 되면 바로 결과물이 실행되는데 `p` 옵션을 붙이면 결과만 표시하고 실행을 금지할 수 있습니다.

```
find . -name "*.log"
...

$ !find:s/log/tmp/:p      # 결과만 프린트되고 실행은 되지 않는다
find . -name "*.tmp"
```

History 관련 환경 변수

- HISTIGNORE

history 리스트에 저장할때 제외시킬 명령패턴을 `:` 로 분리하여 등록합니다.

```
HISTIGNORE='ls:ls -al:cd:bg:fg:history'
```

- HISTFILESIZE

history 파일에 저장될 최대 라인수를 나타냅니다.

- HISTSIZE

history 리스트에 기억될 최대 명령수를 나타냅니다. 디폴트 값은 500 입니다.

- HISTFILE

history 를 저장할 파일을 지정합니다.

- HISTCONTROL

명령 history 의 작동방식을 `:` 로 분리하여 설정할수 있습니다.

- **ignorespace** : space 로 시작하는 명령라인을 history 에 저장하지 않습니다.
- **ignoredups** : 이전 history 명령과 중복될경우 저장하지 않습니다.
- **ignoreboth** : ignorespace:ignoredups 와 같습니다.
- **erasedups** : 이전 모든 history 라인을 비교하여 중복된 history 를 제거한후 저장합니다.

- HISTTIMEFORMAT

history 번호에 이어 timestamp 를 붙일 수 있습니다. history file 에도 저장됩니다.

예) export HISTTIMEFORMAT="%F %T "

History 관련 옵션

Set

- **history**

명령 history 기능을 enable, disable 할 수 있습니다.

- **-H | histexpand**

`!` 문자를 이용한 history 확장 기능을 제공합니다. `set -o history` 이 설정돼있어야 사용할 수 있습니다.

Shopt

- **cmdhist**

multiple-line 명령을 작성할 경우 명령 줄들이 각각 다른 history 번호로 할당돼서 다음에 재사용하기가 어려운데 이 옵션을 사용하면 `newline` 을 `;` 로 치환해서 저장해 줍니다. `lithist` 옵션과 같이 사용하면 `newline` 도 그대로 유지됩니다.

- **lithist**

multiple-line 명령을 작성할 경우 `newline` 을 유지해 줍니다.

- **histreedit**

history 확장이 실패할 경우 입력했던 내용이 없어지지 않고 다시 수정할 수 있는 기회를 줍니다.

- **histverify**

history 확장된 명령을 바로 실행하지 않고 필요시 수정할 수 있게 `enter` 를 칠 기회를 줍니다.

- **histappend**

shell 을 exit 할때 `HISTFILE` 변수에 설정돼 있는 파일에 현재 history list 를 append 합니다. off 이면 overwrite 합니다.

터미널이 비정상적으로 종료할 경우 history list 가 저장되지 않습니다. 그럴 경우를 위해 `PROMPT_COMMAND='history -a'` 를 설정해 사용할 수 있습니다.

History builtin 명령

`history [-c] [-d offset] [n] or history -anrw [filename] or history -ps arg [arg...]`

현재 세션의 history list 를 관리하며 history file 을 read 하거나 write 해서 여러 터미널 세션 간에 history 를 동기화할 수 있습니다.

| 옵션 | 설명 |
|------------------|---|
| -c | 현재 세션의 history list 를 모두 삭제합니다. |
| -d offset | offset 위치의 항목을 삭제합니다. |
| -r | history file 을 읽어들이고 내용을 현재 세션의 history list 에 append 합니다 |
| -n | history file 에서 아직 읽어들이지 않은 항목이 있으면 모두 읽어드립니다. |
| -a | 현재 세션의 history list 를 history file 에 append 합니다. |
| -w | 현재 세션의 history list 를 history file 에 write 합니다. |

Command Completion

명령에서 사용되는 인수나 옵션이 종류가 많거나 또는 이름이 길거나 하면 일일이 기억하기도 힘들고 오류 없이 입력하기도 어렵습니다. 이럴 때 **tab** 키를 이용한 자동완성 기능이 유용하게 사용되는데요 명령문 자동완성 기능은 각 명령에서 자체적으로 제공하는 것은 아니고 **complete**, **compgen**, **compgopt** 명령을 이용해 만들어 사용하는 것입니다. 자주 사용하는 명령에 자동완성 기능이 없다면 만들어서 추가할 수도 있고 이미 설정돼 있는 기능이라도 마음에 들지 않으면 삭제하고 재설정할 수도 있습니다.

```
# find 명령을 사용하기 위해 'find -' 까지 입력한 뒤에 tab 키를 누르면
# 이용할 수 있는 옵션 목록을 한눈에 보여줍니다.
$ find -[tab]
-amin          -fprint0      -mmin         -quit
-anewer        -fprintf     -mount        -readable
-atime         -fstype      -mtime        -regex
-cmin          -gid         -name         -regextype
-cnewer        -group       -newer        -samefile
-context       -help        -nogroup      -size
-ctime        -ignore_readdir_race -noignore_readdir_race -true
-daystart     -ilname      -noleaf       -type
-delete       -iname       -nouser       -uid
-depth        -inum        -nowarn       -used
-empty        -ipath       -ok           -user
-exec         -iregex      -okdir        -version
-execdir      -iwholename  -path         -warn
-executable   -links       -perm         -wholename
-false        -lname       -print        -writable
-fls          -ls          -print0       -xdev
-follow       -maxdepth    -printf       -xtype
```

자동완성을 위해 현재 등록돼 있는 명령 이름들은 **complete builtin** 명령을 통해 볼수있습니다.

```
$ complete -p find
complete -F _find find      # find 명령의 자동완성을 위해 _find 함수가 사용됨

$ complete -p help
complete -A helptopic help  # help 명령의 자동완성을 위해 helptopic action 이 사용됨
```

Completion 은 명령문 스트링을 만드는 작업.

사실 자동완성과 명령은 관계가 없습니다. 무슨말이냐 하면 자동완성은 단지 명령문 스트링을 만드는 작업입니다. 다음은 실제 시스템에 'hello' 라는 명령은 없지만 자동완성을 설정하고 있는 예입니다.


```
# hello 를 위한 자동완성 단어들을 등록
$ completion -W "aaa bbb ccc ddd" hello

# 'hello ' 까지 입력하고 tab 키를 누르면 등록했던 단어들이 표시된다.
$ hello [tab]
aaa bbb ccc ddd

# 'hello c' 까지 입력하고 tab 키를 누르면 'hello ccc' 명령문이 자동완성 된다.
$ hello c[tab]

# 자동완성 등록확인
$ complete -p hello
complete -W 'aaa bbb ccc ddd' hello
```

기본적으로 사용할 수 있는 자동완성 이름들

위의 예에서는 `-w wordlist` 옵션을 이용해 직접 자동완성 단어를 등록해 사용했지만 `completion`에서는 시스템 내에서 기본적으로 사용할 수 있는 여러 이름들을 카테고리 별로 분류하여 제공합니다. 이와 같은 이름들은 `-A action` 옵션을 이용해 사용할 수 있습니다.

```
# 자동완성으로 export action 을 사용
$ complete -A export hello

# 'hello ' 까지 입력하고 tab 키를 누르면 현재 export 된 변수 이름들을 보여준다.
$ hello [tab]
ANT_HOME                LC_ADDRESS              SCALA_HOME
CDPATH                  LC_IDENTIFICATION      SESSION
CLUTTER_IM_MODULE      LC_MEASUREMENT         SESSION_MANAGER
COLORTERM               LC_MONETARY             SESSIONTYPE
DART_SDK                LC_NAME                 SHELL
DBUS_SESSION_BUS_ADDRESS LC_NUMERIC              SHLVL
...
```

다음은 `-A` 옵션으로 사용할 수 있는 action 입니다.

| action | 설명 |
|-----------------------|---|
| -a alias | alias 이름 |
| arrayvar | array 변수 이름 |
| binding | Readline 에서 사용하는 key binding 이름. |
| -b builtin | Shell builtin 명령 이름. |
| -c command | 현재 시스템 내에 있는 모든 명령 이름 (builtin 명령 포함). |
| -d directory | 현재 디렉토리에 있는 디렉토리 이름. IGNORE 변수를 이용해 필터링 할 수 있습니다. |
| disabled | Disable 된 shell builtin 명령 이름. |
| enabled | Enable 된 shell builtin 명령 이름. |
| -e export | Export 된 shell 변수 이름. |
| -f file | 현재 디렉토리에 있는 파일 이름 (디렉토리도 포함). IGNORE 변수를 이용해 필터링 할 수 있습니다. |
| function | Shell 함수 이름. |
| -g group | Group 이름 |
| helptopic | Help builtin 명령으로 도움말을 볼수있는 이름들. |
| hostname | /etc/hosts 에 설정돼있는 호스트이름. (HOSTFILE 변수 설정도 포함). |
| -j job | Job 이름. |
| -k keyword | Shell 키워드. |
| running | Running jobs 이름. |
| -s service | Service 이름. |
| setopt | Set 명령에서 사용할 수 있는 옵션 이름. |
| shopt | Shopt 명령에서 사용할 수 있는 옵션 이름. |
| signal | Signal 이름. |
| stopped | Stopped jobs 이름 |
| -u user | User 이름. |
| -v variable | 모든 shell 변수 이름. |

Completion Options

자동완성 이름들을 생성하는 것 외에도 `-o option` 옵션 설정을 통해 completion 의 동작 방식을 변경할 수 있습니다. 이 옵션은 추후에 `compopt` 명령을 사용해 변경할 수 있습니다. 현재 설정된 옵션 상태는 `compopt` 명령이름 으로 확인할 수 있으며 `-` 로 표시된 것은 현재 설정되어있는 상태고 `+` 로 표시된 것은 설정이 안된 상태입니다

- **bashdefault**

자동완성 이름을 생성하지 못했을때 default Bash completions 을 사용하게 됩니다.

```
$ complete -o bashdefault -w ' ' hello

# bash 변수이름 자동완성
$ hello $BASH_[tab]
$BASH_ALIASES          $BASH_COMMAND          $BASH_SOURCE
$BASH_ARGC             $BASH_COMPLETION_COMPAT_DIR $BASH_SUBSHELL
$BASH_ARGV             $BASH_LINENO          $BASH_VERSINFO
$BASH_CMDS             $BASH_REMATCH         $BASH_VERSION

# 패턴에 매칭되는 파일을 표시
$ hello Read*[tab]
ReadObject.class  ReadObject.java

$ hello Read*.j*[tab]
$ hello ReadObject.java    # 파일이름 완성
```

- **default**

자동완성 이름을 생성하지 못했을때 readline 기본 파일이름 완성을 사용하게 됩니다.

```
# 자동완성 이름이 하나도 생성되지 못하게 -w ' ' 설정
$ complete -w ' ' hello

$ hello [tab] # tab 키를 눌러도 아무것도 나타나지 않는다.

# 이번에는 '-o default' 옵션 추가
$ complete -o default -w ' ' hello

# 자동완성 이름을 생성하지 못하자 readline 기본 파일이름 완성을 사용.
$ hello [tab]
2013-03-19 154412.csv  music/          video/
Address.java          ReadObject.class WriteObject.class
address.ser           ReadObject.java  WriteObject.java
```

- **dirnames**

자동완성 이름을 생성하지 못했을때 디렉토리 이름 완성을 사용하게 됩니다.

```
$ complete -o dirnames -w ' ' hello

# 자동완성 이름을 생성하지 못하자 디렉토리 이름 완성을 사용.
$ hello [tab]
music/          video/
```

- **filenames**

자동완성 이름을 파일이름 으로 취급합니다. 그래서 이름에 공백이나 특수문자가 있을 경우 자동으로 **escape** 해주고 동일한 이름의 디렉토리가 있을 경우는 뒤에 **/** 를 붙여줍니다.

```
$ complete -w 'hoo\ bar foo&bar music' hello

# 자동완성 이름이 수정없이 그대로 완성된다.
$ hello hoo bar
$ hello foo&bar
$ hello music

# 이번에는 -o filenames 옵션 사용
$ complete -o filenames -w 'hoo\ bar foo&bar music' hello

# 자동완성 이름을 파일로 취급하여 이름에 공백이나 특수문자가 있을 경우 escape 해주고
# 같은 이름의 디렉토리가 있을 경우 '/' 를 붙여준다.
$ hello hoo\ bar
$ hello foo\&bar
$ hello music/
```

- **noquote**

기본적으로 파일이름을 완성할 때는 공백이나 특수문자를 **escape** 주는데 이 옵션을 설정하면 **disable** 됩니다.

```
# 현재 디렉토리 목록
$ ls
2013-03-19 154412.csv  address.ser  music/          ReadObject.java  WriteObject.class
Address.java          foo&bar.cvs  ReadObject.class  video/           WriteObject.java

# file action 을 사용하여 현재 디렉토리에 있는 파일이름을 자동완성으로 사용
$ complete -A file hello

# 기본적으로 파일이름은 공백이나 특수문자를 escape 해준다.
$ hello 2013-03-19\ 154412.csv
$ hello foo\&bar.cvs

# 이번에는 -o noquote 옵션 사용
$ complete -o noquote -A file hello

# 파일이름의 자동 escape 기능이 disable 된다.
$ hello 2013-03-19 154412.csv
$ hello foo&bar.cvs
```

- **nospace**

이름을 완성하고 나면 다음 이름을 위해 공백을 띄우게 되는데요. 이 옵션을 그걸 방지합니다.

```
$ complete -w 'aaa bbb ccc' hello

# 이름을 완성하고 나면 자동으로 공백을 띄운다.
$ hello aaa [stop]

# 이번에는 -o nospace 옵션 사용
$ complete -o nospace -w 'aaa bbb ccc' hello

# 이름을 완성하고 나서 공백을 띄우지 않는다.
$ hello aaa[stop]
```

• plusdirs

생성된 자동완성 이름에 디렉토리 이름 완성을 추가 합니다. 또한 자동완성 이름과 매칭되는 파일이 있을경우 공백, 특수문자를 escape 해줍니다.

```
# 현재 디렉토리 목록
$ ls
2013-03-19 154412.csv  address.ser  music/          ReadObject.java  WriteObject.class
Address.java          foo&bar.cvs  ReadObject.class video/           WriteObject.java

# -o plusdirs 옵션 설정
$ complete -o plusdirs -w 'aaa bbb ccc' hello

# 자동완성 이름 aaa bbb ccc 외에 디렉토리 이름 완성이 추가 되었다
$ hello [tab]
aaa      bbb      ccc      music/ video/
```

-G globpat

globbing 을 자동완성 이름을 생성하는데 사용할 수 있습니다.

```
$ complete -G "*.java" hello

$ hello [tab]
Address.java      ReadObject.java  WriteObject.java
```

-X filterpat

이것은 생성된 자동완성 이름을 패턴을 이용해 필터링 하는 기능입니다.

```
$ complete -w 'aaa.x bbb.y ccc.z' -X "*.z" hello

$ hello          # '*.z' 패턴에 의해 'ccc.z' 가 필터링 되었다.
aaa.x  bbb.y
```

-P prefix, -S suffix

생성된 자동완성 이름 앞, 뒤에 prefix, suffix 를 붙일 수 있습니다.

```
$ complete -w 'aaa bbb ccc' -S "/" hello

$ hello
aaa/  bbb/  ccc/
```

-D

자동완성이 설정되지 않은 명령들에게 default 로 적용하기 위해 사용

-E

명령행이 empty 상태일때 적용하기 위해 사용

자동완성 함수

위에서 여러가지 자동완성 이름을 생성하는 방법을 살펴보았지만 사실 주로 사용하는 방법은 `-F` 함수이름 을 이용하는 것입니다. 자동완성 함수는 script 처럼 child process 가 생성돼서 실행되는 게 아니고 현재 shell 에서 실행되므로 interactive shell 환경을 그대로 가지고 있습니다. (alias, history, job control enabled). 그러므로 alias 가 적용된다든지 하면 오류가 발생할 수 있으니 주의해야 합니다.

함수에서 사용되는 변수들

`$1` , `$2` , `$3` 의 의미는 다음과 같습니다.

```
mycomm argument1 argument2 argu[tab]
  \              |              \
  \_ $1          |_ $3          \_ $2
(command)      (previous)    (current)
```

current 가 " " 문자로 시작할 경우 \$2 는 " " 문자를 값으로 전달받지 못합니다. 그럴 경우 `${COMP_WORDS[COMP_CWORD]}` 를 이용하면 됩니다. 함수에서 주로 사용하는 변수는 다음과 같습니다.

- **COMP_WORDS**

현재까지 입력한 명령의 단어들을 값으로 가지고 있는 array 변수입니다. 위의 예에서는

```
COMP_WORDS[0]=mycomm , COMP_WORDS[1]=argument1 , COMP_WORDS[2]=argument2 ,
COMP_WORDS[3]=argu 가 됩니다.
```

- **COMP_CWORD**

현재 커서가 위치한 단어의 index 를 나타냅니다. 위의 예에서는 3 이 됩니다. \$2 는

```
${COMP_WORDS[COMP_CWORD]} 와 같고 $3 은 ${COMP_WORDS[COMP_CWORD-1]} 와 같게됩니다.
```

- **COMP_REPLY**

Array 변수로, 함수를 통해 만들어진 자동완성 단어를 최종적으로 이변수에 넣은후 리턴하게 됩니다. 그럼 tab 키를 이용한 자동완성시에 이 변수의 값들이 자동완성 단어로 사용되게 됩니다.

compgen 명령

이명령은 자동완성 이름을 생성하는데 사용됩니다. 앞서 소개한 complete 명령의 옵션들을 거의 동일하게 사용할 수 있습니다. 한가지 유용한 기능은 마지막에 word 를 인수로 주면 word 와 매칭 되는 단어들만 선택되게 됩니다. 그러므로 주로 COMP_REPLY 변수에 값을 저장하는 용도로 사용됩니다.

```
# 마지막에 word 인수를 주면 매칭되는 단어들만 나온다
$ compgen -W 'a111 b222 b333 c444' -- b
b222
b333
```

예제)

다음은 간단하게 2 단계의 옵션을 처리하는 내용입니다. 우선 자동완성 함수는 script 가 아니기 때문에 파일 첫줄에 shebang line (`#!/bin/bash`) 은 필요하지 않습니다. 본문중에 사용되는 자동완성 단어는 직접 입력하였으나 꼭 그럴 필요는 없고 여러 유틸리티 프로그램을 활용해 생성할 수 있습니다. 최종적으로 COMP_REPLY array 변수에 단어들을 넣어주고 return 하면 되는 것입니다.

다음 내용을 mycomp.sh 에 작성하였다면 `source mycomp.sh` 한 후 프롬프트 상에서 테스트해봅니다. 정상적으로 동작 되는 것을 확인하였으면 `~/.bash_completion` 에 저장하여 사용하거나 `/etc/bash_completion.d/` 에 놓고 사용하시면 됩니다.

파일 마지막에 complete 명령의 `-F` 옵션으로 hello 의 자동완성 함수를 등록하는 것을 볼 수 있습니다.

```
mycomp_() {

    local comm=$1
    local cur=$2
    local prev=$3

    local options="--fruit --planet --animal"
    local fruits="apple orange banana"
    local planets="mars jupiter saturn"
    local animals="lion tiger elephant"

    if [ ${COMP_CWORD} -eq 1 ]; then
        COMPREPLY=( $(compgen -W "$options" -- "$cur") )
        return
    fi

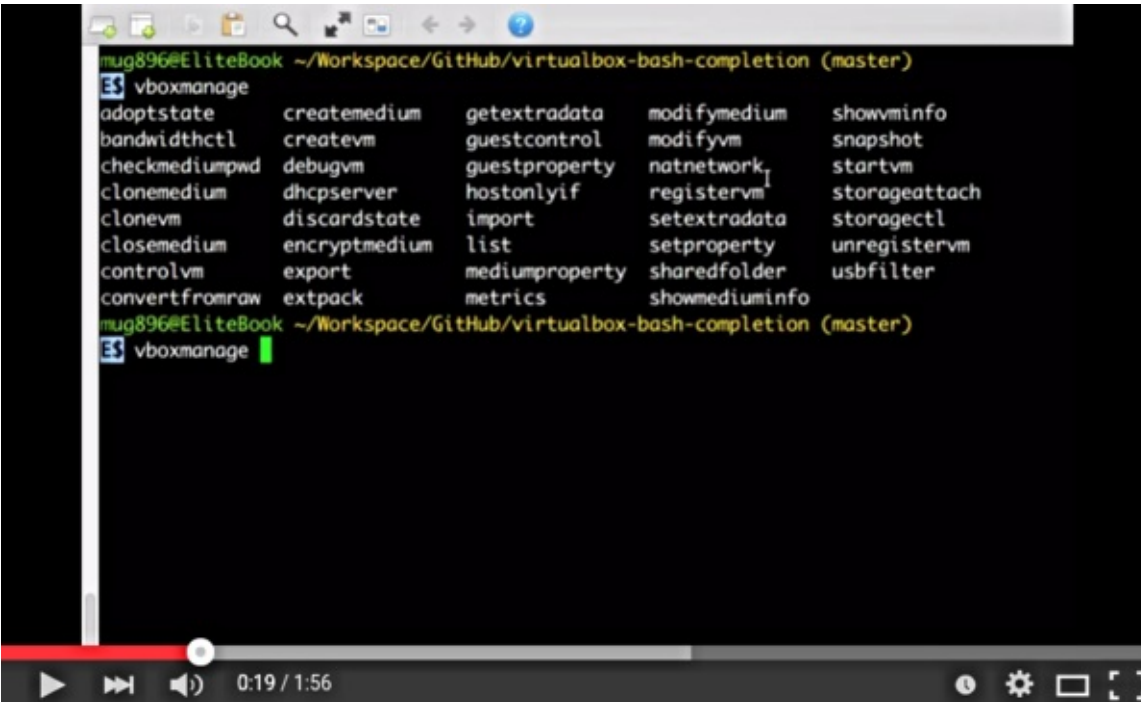
    if [ ${COMP_CWORD} -eq 2 ]; then
        case "$prev" in
            --fruit)
                COMPREPLY=( $(compgen -W "$fruits" -- "$cur") )
                ;;
            --planet)
                COMPREPLY=( $(compgen -W "$planets" -- "$cur") )
                ;;
            --animal)
                COMPREPLY=( $(compgen -W "$animals" -- "$cur") )
                ;;
        esac
        return
    fi
}

complete -F mycomp_ hello
```

Virtualbox Bash Completion

vboxmanage 같은 명령은 서브 명령과 옵션들이 많아 터미널에서 사용하기가 어려운데 자동완성 함수를 작성해 놓으면 편리하게 사용할 수 있습니다.

<https://github.com/mug896/virtualbox-bash-completion>



The screenshot shows a terminal window with a dark background. At the top, the prompt is `mug896@EliteBook ~/Workspace/GitHub/virtualbox-bash-completion (master)`. Below the prompt, the command `vboxmanage` has been entered, and a list of 25 VirtualBox commands is displayed in a grid-like format. The commands are: `adoptstate`, `bandwidthctl`, `checkmediumpwd`, `clonemedium`, `clonevm`, `closemedium`, `controlvm`, `convertfromraw`, `createmedium`, `createvm`, `debugvm`, `dhcpcserver`, `discardstate`, `encryptmedium`, `export`, `extpack`, `getextradata`, `guestcontrol`, `guestproperty`, `hostonlyif`, `import`, `list`, `mediumproperty`, `metrics`, `modifymedium`, `modifyvm`, `natnetwork`, `registervm`, `setextradata`, `setproperty`, `sharedfolder`, `showmediuminfo`, `showvminfo`, `snapshot`, `startvm`, `storageattach`, `storagectl`, `unregistervm`, and `usbfilter`. Below the list, the prompt is repeated, and the command `vboxmanage` is entered again, followed by a green cursor.

```
mug896@EliteBook ~/Workspace/GitHub/virtualbox-bash-completion (master)
vboxmanage
adoptstate      createmedium    getextradata    modifymedium     showvminfo
bandwidthctl    createvm        guestcontrol     modifyvm         snapshot
checkmediumpwd  debugvm         guestproperty    natnetwork       startvm
clonemedium     dhcpcserver     hostonlyif       registervm       storageattach
clonevm         discardstate    import           setextradata     storagectl
closemedium     encryptmedium   list             setproperty      unregistervm
controlvm       export          mediumproperty   sharedfolder      usbfilter
convertfromraw  extpack         metrics          showmediuminfo

mug896@EliteBook ~/Workspace/GitHub/virtualbox-bash-completion (master)
vboxmanage
```

Readline

CLI (command line interface) 를 사용하는 프로그램에서 line editing, 명령 history, 자동완성 기능을 제공하는 라이브러리로 shell 프롬프트 상에서 명령문을 작성할때 사용하는 키조합이나, 명령 history, 자동완성 기능은 이 라이브러리를 이용하는 것입니다. line editing 에는 emacs 와 vi 두 종류의 키조합을 제공하는데 기본설정은 emacs 이며 `set -o emacs` or `set -o vi` 옵션을 통해 변경할 수 있습니다.

Readline init file

readline 은 `~/.inputrc` 설정 파일을 통해 옵션을 설정하고 키조합을 커스터마이징 할 수 있습니다. 다음은 `completion-ignore-case` 옵션을 설정하여 `tab` 및 `alt-/` 키를 이용한 자동완성 시에 대, 소문자 구분없이 하고 `alt-space` 에는 "git " 을 `ctrl-space` 에는 "docker " 스트링이 프롬프트 상에 입력되게 하며 평션키 `F9` , `F10` 에는 자주 사용하는 명령을 바인딩하는 예입니다.

키조합에 사용되는 키값은 `read` 명령으로 구할 수 있습니다.

구한 키값에서 `^` 문자를 `\e` 로 수정하여 사용합니다.

```
$ read          # enter 후에 원하는 키를 입력
^[[20~         # F9 키를 누른 상태
```

`inputrc` 파일을 수정한 후에 `c-x c-r` 를 입력하거나 새로 터미널 창을 열면 설정된 기능을 사용할 수 있습니다. 한가지 주의할 점은 window manager 나 터미널창 자체에서 이미 키조합이 설정되었으면 적용이 되지 않으므로 필요없는 키조합은 먼저 삭제하시기 바랍니다.

~/.inputrc 활용 예제

```
# 대,소문자 구분없이 word completion
set completion-ignore-case on

# emacs 모드로 설정
set editing-mode emacs

# emacs 모드일경우 적용
$if mode=emacs

# alt-space 치면 프롬프트에 git 입력
"\e ": "git "

# ctrl-space 치면 프롬프트에 docker 입력
"\C-@": "docker "

# \C-k\C-u 는 kill-line(C-k), unix-line-discard(C-u) line editing 명령으로
# 현재 프롬프트에 입력되어있는 스트링을 모두 삭제합니다. \C-m 은 enter 를 의미합니다.

# F9 키에 git pull 명령 바인딩
"\e[20~": "\C-k\C-ugit pull\C-m"

# F10 키에 git push 명령 바인딩
"\e[21~": "\C-k\C-ugit push\C-m"

$endif
```

유용한 키보드 shortcut

- 명령에서 파일이름 자동완성 기능을 제공하지 않을 경우 `Alt-/` 로 자동완성을 할 수 있습니다.
- 이전 명령에서 사용된 마지막 인수를 입력하고 싶을 때는 `Alt-.` 을 사용하면 됩니다.
- `ctrl-r` 은 history 를 검색하여 입력한 패턴에 해당하는 명령을 불러와 실행할 수 있게 해줍니다. 연속해서 누르면 매칭되는 다음 항목으로 넘어갑니다.
- 프롬프트에서 명령문을 입력중에 editor 를 불러오고 싶을때 `c-x c-e` 를 하면 현재 입력중인 내용과 함께 vi editor 가 열립니다. `:wq` 명령으로 vi 를 종료하면 수정된 명령이 실행되고 `:cq` 명령으로 종료하면 실행되지 않습니다.
- 전체 목록은 <http://readline.kablamo.org/emacs.html> 에서 볼 수 있습니다.

Readline wrapper

CLI 를 사용하는 프로그램에서 readline 기능을 사용할수 있게 해주는 프로그램 입니다. `sqlplus`, `ed` 같은 프로그램을 사용할때 line editing, history 기능이 없어서 불편했다면 설치해 사용할 수 있습니다.

```
$ sudo apt-get install rlwrap
```

```
$ rlwrap ed -p :
```

Debugging

Syntax Check

`-n` | `set -o noexec` 옵션은 명령을 실행하지 않으므로 **syntax errors** 를 체크하는 용도로 사용할 수 있습니다. 단 **syntax** 만 체크합니다. 명령 이름에 **typo** 가 있는지 또는 명령 사용에 오류가 있는지는 각 명령에 해당하는 것이므로 체크되지 않습니다.

```
./test.sh: line 8: syntax error near unexpected token `fi'
```

Typo Check

`-u` | `set -o nounset` 옵션은 존재하지 않는 변수를 사용할 경우 에러로 간주하여 **exit** 합니다. 그러므로 **typo check** 용도로 사용할 수 있습니다. 다음의 경우 `-u` 옵션을 사용하지 않았다면 **Workspace** 디렉토리 전체가 삭제될 수 있습니다.

```
#!/bin/bash -u
...
myTestDir=test2
...
rm -rf ~/Workspace/$myTestdir
...
##### output #####

./test.sh: line 15: myTestdir: unbound variable
```

스크립트 실행 당시 어떤 변수가 존재할 수도 있고 아닐 수도 있을 경우 다음과 같은 매개변수 확장 기능을 이용하면 **exit** 되는 것을 방지할 수 있습니다.

```
# SOME_ENV_VAR 가 존재하지 않으면 null 을 리턴하게 되고 exit 되지 않습니다.

if [ "${SOME_ENV_VAR:-}" ]; then      # 또는 ${SOME_ENV_VAR:-} 을 사용해도 됨
    echo exist
else
    echo null or unset
fi
```

다음은 함수에 인수를 전달하지 않았을때 체크하는 방법과, 변수가 존재하지 않을때 대체값을 사용하는 방법입니다.

```
myfunc()
{
    # 함수에 인수를 전달하지 않으면 $1 변수는 존재하지 않는 상태.
    if [ -n "${1-}" ]; then ...

    # flag 변수가 존재하지 않거나 null 값일 경우 1 을 사용
    res=$(( ${flag:-1} + 100 ))
}
```

Xtrace

Xtrace 의 가장 큰 장점은 매개변수확장, 명령치환, 산술확장이 완료된 명령문을 보여주고 실행한다는 점입니다. 그러므로 명령 실행시 변수값에 대한 정보를 알 수 있고, colon 명령 `:` 을 이용하면 인수 부분에서 필요한 연산 결과를 얻어 낼수도 있습니다. `-x | set -o xtrace` 옵션으로 설정할 수 있으며 `set -` 은 `set +o xtrace` 와 같은 의미로 스크립트 일부분을 trace 할때 간단히 사용할 수 있습니다..

다음은 xtrace 를 이용해 `[` 명령의 'too many arguments' 오류를 trace 하는 예입니다.

```

### trace 에 사용된 스크립트 : xtrace.sh

#!/bin/bash

AA="foo bar"

if [ $AA = "foo bar" ]; then
    echo same !!!
else
    echo not same !!!
fi

### 실행결과

$ ./xtrace.sh
$ ./xtrace.sh: line 14: [: too many arguments
not same !!!

### xtrace 결과

$ bash -x ./xtrace.sh
+ BB='foo bar'
+ '[' foo bar = 'foo bar' ']'          # AA 변수값이 foo bar 두개로 나온다.
./xtrace.sh: line 14: [: too many arguments
+ echo not same '!!!'
not same !!!

### 변수 $AA 를 "$AA" 로 quote 한 후 실행결과

$ bash -x ./xtrace.sh
+ BB='foo bar'
+ '[' 'foo bar' = 'foo bar' ']'        # AA 변수값이 'foo bar' 하나로 나온다.
+ echo same '!!!'
same !!!                               # 결과도 정상적으로 나옴.

```

xtrace 를 할때 변수이름 앞에 \$ 문자가 없으면 값이 표시되지 않습니다. 이때 값을 보기 위해 echo 명령을 사용하게 되면 echo 명령도 함께 trace 가 돼서 보기가 안좋은데요. 이때 colon 명령 : 을 활용할 수 있습니다. : 도 명령이기 때문에 인수가 확장돼서 나온다는 점을 이용하는 것입니다.

```
#!/ bin/bash

set -x
for i in {25..28}; do
    (( i = i * 123 ))
    : i = $i, i / 2 = $(( i / 2 ))      # ':' 명령 실행 전에 인수확장이 됨
done
set -

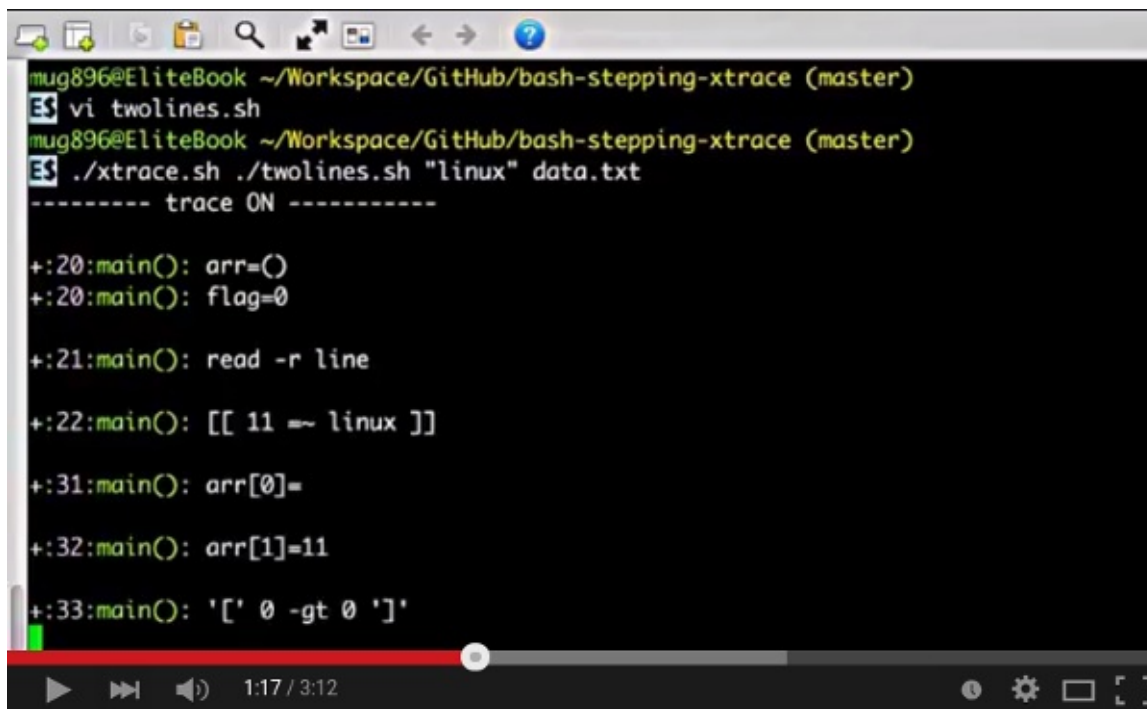
##### output #####

+ for i in '{25..28}'
+ (( i = i * 123 ))
+ : i = 3075, i / 2 = 1537
+ for i in '{25..28}'
+ (( i = i * 123 ))
+ : i = 3198, i / 2 = 1599
+ for i in '{25..28}'
+ (( i = i * 123 ))
+ : i = 3321, i / 2 = 1660
+ for i in '{25..28}'
+ (( i = i * 123 ))
+ : i = 3444, i / 2 = 1722
+ set -
```

Xtrace 에서 stepping 을 하자

Xtrace 는 프로그래밍 언어에서 debugging 할때처럼 break point 를 설정하거나 명령을 하나씩 stepping 할수가 없는데요. DEBUG trap 과 job control 을 이용하면 비슷하게 흉내낼 수 있습니다.

<https://github.com/mug896/bash-stepping-xtrace>



Trapping ERR

스크립트 실행시에 어떤 명령이 에러로 종료했다면 스크립트 실행을 중단해야 하지만 기본적으로 shell 은 중단없이 끝까지 진행을 합니다. 이러한 경우 스크립트 실행을 중지시키기 위해서 `-e` | `set -o errexit` 옵션을 사용할 수 있습니다. 이 옵션은 한가지 단점이 있는데 에러가 난 명령에서 오류 메시지를 출력하지 않으면 별다른 메시지가 표시되지 않는다는 점입니다. 이럴때 ERR trap 과 BASH_SOURCE, LINENO, BASH_COMMAND, `$?` 변수를 이용하면 에러가 발생한 명령의 위치와 종료값을 알 수 있습니다.

오류를 발생하는 명령으로 사용되는 스크립트 : `error_command.sh`

```
#!/bin/bash

echo ----- error_command.sh ----- start
exit 3
echo ----- error_command.sh ----- end
```

메인 스크립트 : trap error.sh

`-E` | `set -o errtrace` 옵션은 function 과 subshell 에서도 ERR trap 을 가능하게 합니다.

```
#!/bin/bash -Ee
trap 'exitcode=$?; echo ERR!!! "${BASH_SOURCE[0]}": $LINENO: exit $exitcode: "$BASH_COMMA'

echo ----- trap_err.sh ----- start
./error_command.sh
echo ----- trap_err.sh ----- end
```

실행결과

```
$ ./trap_err.sh
----- trap_err.sh ----- start
----- error_command.sh ----- start
ERR!!! ./trap_err.sh: 5: exit 3: ./error_command.sh
```

if, while, &&, || 에서는 ERR trap 이 되지 않는다.

오류가 나는 명령이 if, while, &&, || 에서 사용될 경우는 trap 이 되지 않습니다.

```
# &&, || 에서 사용되는 경우
./error_command.sh || true

# if 문에서 사용되는 경우
if ./error_command.sh; then
:
else
:
fi
```

이럴 경우 trap 을 사용하려면 다음과 같이 할 수 있습니다.

```
./error_command.sh
[ $? -eq 0 ] && echo true

./error_command.sh
if [ $? -eq 0 ]; then
...

```

ERR trap 을 할때 문제점

산술연산 에서는 연산결과가 0 이면 false 이므로 종료상태 값으로 1 을 리턴한다.

```
# let, (( )) 는 산술연산 표현식

$ i=0                # i++ 는 postfix operator 로 연산결과는 0 이고
$ (( i++ ))          # 이후에 값이 증가하여 1 이 됩니다.
$ echo $?            # 그러므로 종료상태 값으로 1 을 리턴
1
-----

$ i=0
$ let i++            # 위와 마찬가지로
$ echo $?
1
-----

# expr 도 산술연산 명령으로 1 - 1 은 0 이므로 false 에 해당합니다.
# 그러므로 expr 명령의 종료상태 값으로 1 을 리턴합니다.
$ foo=$(expr 1 - 1)
$ echo $?
1
```

&& , || 에서는 ERR trap 이 되지 않지만 함수에서는 에러를 리턴한다.

```
# 다음의 경우 test 가 실패했을때 && 메타문자로 인해 trap 이 되지 않는다.
test -d nosuchdir && ...

# 하지만 함수에서는 마지막 명령의 종료상태 값을 리턴하므로 에러 trap 이 된다.
f1() { test -d nosuchdir && ... ;}
f1
```

read 명령에서 -d 옵션 값으로 null 을 사용하거나 end-of-file 상태를 만났을때 에러를 리턴한다.

Dash

Dash (Debian Almquist SHell) 는 POSIX 호환의 /bin/sh 구현체로 각기 다른 shell 환경을 가지는 여러 종류의 OS 에서 호환성의 문제없이 실행될 수 있습니다. interactive shell 보다는 주로 script 를 작성하는데 사용되며 bash 에 비해 실행파일의 크기가 작고 의존하는 라이브러리도 적어서 빠른 start-up time 을 요구하는 boot script 용으로 많이 사용됩니다. (OS 가 부팅될때 많은수의 shell script 가 실행되므로 shell 의 start-up time 은 부팅속도 개선에 중요한 역할을 합니다.)

<http://unix.stackexchange.com/a/148061/117612> 에서 bash 와 비교한 것을 볼 수 있습니다.

dash 는 script 를 작성하는데 필요한 기본적인 기능만 제공합니다. 그러므로 `[[]]` , `(())` , `let` 과 같은 bash 에서만 제공하는 특수 표현식은 (보통 bashism 이라고 합니다) 사용할 수 없습니다.

Quotes

echo 명령의 기본적인 동작방식은 `echo -e` 와 같습니다. 그러므로 single quotes, double quotes 모두에서 escape 문자를 사용할 수 있습니다. `$' '` 는 bashism 으로 사용할 수 없습니다.

```
$ AA=" \t\n"           # double quote
$ echo -n "$AA" | od -a
00000000  sp  ht  nl

$ AA=' \t\n'          # single quote
$ echo -n "$AA" | od -a
00000000  sp  ht  nl
```

\$((...))

다음 연산자들은 사용할 수 없습니다.

`++` , `--` 연산자

```
$ : $(( i++ ))
sh: 2: arithmetic expression: expecting primary: " i++ " # 오류발생

# 다음과 같이 수정해 사용합니다.

i=$((i+1))   또는   i=$((i-1))
:=$((i+=1))  또는   :=$((i-=1))
```

, 연산자

```
$ : $(( i+=1, i+=1 ))
sh: 21: arithmetic expression: expecting EOF: " i+=1 , i+=1 " # 오류발생
```

**** 연산자**

```
$ echo $(( 2**3 ))
sh: 363: arithmetic expression: expecting primary: " 2**3 " # 오류발생
```

Parameter expansion

다음은 제외하고는 사용할 수 있습니다.

Substring expansion : `${parameter:offset:length}` ...

Search and replace : `${parameter/pattern/string}` ...

Indirection : `${!parameter}`

Case modification : `${parameter^^}, ${parameter,,}` ...

[!]

패턴매칭에서 `[]` 를 사용할때 `not` 을 의미하는 문자로 `^` 대신에 `!` 를 사용해야 합니다.

```
case "goo" in
    [!f]*)
        echo 'not f*'
        ;;
esac
```

Redirections

`1>&3-` 는 두단계로 나누어서 합니다. `1>&3 3>&-`

`command &> file` 는 `command > file 2>&1` 로 변경합니다.

`command1 |& command2` 는 `command1 2>&1 | command2` 로 변경합니다.

read

`-r` 옵션만 사용가능 합니다.

<<

Here document << 는 사용할 수 있습니다.

trap

pseudo-signals 중에서 EXIT 만 사용할 수 있습니다.

명령에 선행하는 대입연산

read 명령 에서는 bash 와 동일하게 동작하나 그외 eval 이나 함수를 사용할 경우는 변수에 값이 대입돼버리고 이전 값으로 복귀가 안됩니다.

```
# read 명령에서는 bash 와 동일하게 동작
$ echo -n "$IFS" | od -a
00000000 sp ht n1
$ IFS=: read v1 v2 v3
1:2:3
$ echo $v1 $v2 $v3
1 2 3
$ echo -n "$IFS" | od -a # 원래 값으로 복귀
00000000 sp ht n1

# eval 을 사용할 경우
$ a=1 b=2 c=3
$ a=4 b=5 c=6 eval 'echo $a $b $c'
4 5 6
$ echo $a $b $c # 원래 값으로 복귀가 안되고 변수에 값이 대입됨
4 5 6

# script 파일을 실행하는 것처럼 child process 가 생성되는 경우는 정상작동합니다.
$ echo $a $b $c
1 2 3
$ a=4 b=5 c=6 bash -c 'echo $a $b $c'
4 5 6
$ echo $a $b $c
1 2 3
```

Bashism

다음은 bash 에서만 사용할 수 있는 기능으로 dash 에서는 사용할 수 없습니다.

+=

+= 대입 메타문자는 사용할 수 없습니다. 대신에 다음과 같은 식으로 사용하면 됩니다.

```

### bash ###
$ AA=hello
$ AA+=" World"
$ echo "$AA"
hello World

### dash ###
$ AA=hello
$ AA="$AA World"    # 또는 "$AA" " World"
$ echo "$AA"
hello World

```

\$' ', \$" "

bashism 으로 사용할 수 없습니다.

Array

Array 는 POSIX 에 정의되어있지 않으므로 indexed array, associative array 모두 사용할 수 없고 AA=() 표현식도 사용할 수 없습니다. 대신에 다음과 같이 positional parameters 를 활용할 수 있습니다.

```

#!/bin/sh

line="11:22:33:44:55"

set -f; IFS=:          # globbing 을 disable
set -- $line           # IFS 값에 따라 필드를 분리하여 positional parameters 에 할당
set +f; IFS=$' \t\n'

echo number of fields = $#
echo field 1 = "$1"
echo field 2 = "$2"

shift 3
echo \${@} = "${@}"

##### output #####

number of fields = 5
field 1 = 11
field 2 = 22
${@} = 44 55

```

'[' 명령에서 '==' 사용

```
[ "$var1" == "$var2" ]    # bashism 으로 사용할 수 없음

[ "$var1" = "$var2" ]     # '=' 로 수정
```

[[. . .]], ((. . .))

for ((i=0; i<3; i++))...

bashism 으로 모두 사용할 수 없습니다.

Brace expansion

Process substitution

<<<

here string 은 사용할 수 없습니다.

function

```
function myfunc() { ... ;}  # function 키워드는 사용할 수 없음

myfunc() { ... ;}           # 수정
```

source

```
source ./subscript.sh      # source 명령은 사용할 수 없음

. ./subscript.sh           # '.' 명령으로 수정
```

declare

local 사용

select, disown

shopt

shopt 으로 설정할 수 있는 옵션은 모두 사용할 수 없습니다.

\$LINENO, \$PIPESTATUS

\$RANDOM

\$RANDOM 변수는 사용할 수 없으므로 다음과 같은 방법을 이용합니다.

```
random=$(awk 'BEGIN{srand(); printf "%d\n", (rand()*256)}')
```

Colors

ASCII 차트에서 0x20 이하 문자들과 0x7F 는 A B C 1 2 3 와같이 화면에 표시되는 모양은 없지만 라인개행을 한다든지 backspace 로 문자를 삭제하는 등의 기능을 하는 control 문자입니다. 그 외 function 키나 방향키에 대한 값 들은 특수한 방법을 이용해 전송하는데요. 방법은 ascii 차트에서 escape 문자에 해당하는 0x1b 문자를 먼저 보내고 뒤에 관련된 문자들(sequence)를 전송합니다 (그래서 escape sequence 라고 부릅니다)

```
$ od -tax1 [enter]
^[[OP^[OQ^[OR^[A^[[B      # f1 ~ f3 키, 위 아래 방향키를 누르고 [enter]
00000000 esc  0  P esc  0  Q esc  0  R esc  [  A esc  [  B nl
          1b 4f 50 1b 4f 51 1b 4f 52 1b 5b 41 1b 5b 42 0a
^C      # ctrl-c 로 종료
```

위의 od 명령을 이용해 function 키와 방향키를 눌렀을때의 키값을 조회해 보면 제일앞에 0x1b (esc) 문자가 나오는것을 볼 수 있습니다. 이렇게 escape sequence 를 전송받은 프로그램은 나름 대로 해석하여 처리하게 됩니다.

Color 표시를 위한 escape sequence

터미널에 color 를 표시할 때도 escape sequence 를 터미널 프로그램에 전송해 처리하게 하는데 작성하는데 간단한 규칙이 있습니다.

1. escape

escape 문자는 다음 중에 하나를 사용할 수 있습니다.

```
\e , \x1b (16진수) , \033 (8진수)
```

2. [

시작을 나타냅니다.

3. style; foreground; background 값

style 값은 여러개를 적을 수 있습니다.

4. m

종료를 나타냅니다.

사용예)

```
# \e 문자 처리를 위해 echo 명령에서 -e 옵션을 사용합니다.
echo -e "\e[0;31m      red foreground"
echo -e "\e[0;44m      blue background"
echo -e "\e[0;1m       bright style"
echo -e "\e[0;31;44m    red foreground, blue background"
echo -e "\e[0;1;31;44m  bright style, red foreground, blue background"

# 다음에 이어지는 스트링에는 속성이 적용되지 않게 reset 합니다.
echo -e "\e[0m"

# 먼저 reset 한 후 속성을 적용합니다.
echo -e "\e[0;1;31m  ..."
```

Color, Style 속성값

다음은 16 color terminal 에 적용할 수 있는 값입니다. style 값은 10 보다 작고 foreground 값은 30 번대, background 값은 40 번대를 사용하는 것을 알 수 있습니다. 값만으로 구분이 가능하므로 순서에 상관없이 적을 수 있습니다.

```
# Style
0 - Reset all attributes # 이전에 적용했던 속성을 reset 합니다.
1 - Bright
2 - Dim
4 - Underscore
5 - Blink
7 - Reverse # foreground 와 background 색상이 서로 바뀌어 표시됩니다.
8 - Hidden # 스트링이 보이지 않게됩니다.

# Foreground
30 - Black
31 - Red
32 - Green
33 - Yellow
34 - Blue
35 - Magenta
36 - Cyan
37 - White

# Background
40 - Black
41 - Red
42 - Green
43 - Yellow
44 - Blue
45 - Magenta
46 - Cyan
47 - White
```

스크립트로 전체 색상 프린트 해보기

```
#!/bin/bash

for fgcolor in {30..37} ; do
  for bgcolor in {40..47}; do
    for attr in 0 1 2 4 5 7; do
      echo -en "\e[${attr};${fgcolor};${bgcolor}m"
      echo -n " ${attr};${fgcolor};${bgcolor} "
      echo -en "\e[0m"
    done
    echo
  done
done
echo

-----

#!/bin/bash

for fgcolor in {30..37} {90..97} 39 ; do
  for bgcolor in {40..47} {100..107} 49 ; do
    for attr in 0 1 2 4 5 7 ; do
      echo -en "\e[${attr};${bgcolor};${fgcolor}m"
      echo -n " ${attr};${bgcolor};${fgcolor} "
      echo -en "\e[0m"
    done
    echo
  done
done
echo
```

tput 명령을 이용한 설정

실제 터미널 장치가 메인프레임에 연결되어 사용되던 과거에는 터미널의 종류가 한두 가지가 아니었을 것이라는 것은 쉽게 짐작할 수 있습니다. 각각의 터미널 장치들은 제공하는 기능도 달랐고 control characters, escape sequences 를 이용한 commands 들도 틀렸다고 합니다. 그래서 이와같은 터미널들의 정보를 라이브러리로 만들어놓은 것이 terminfo (termcap 의 새로운 버전) 입니다 (우분투의 경우 /lib/terminfo 에서 볼수있습니다). terminfo 라이브러리를 이용하면 device independent 한 방법으로 프로그램 할 수 있습니다.

위에서 처럼 color escape sequence 를 직접 하드코드하면 다른 종류의 터미널을 emulation 하는 프로그램에서는 동일하게 동작하지 않을 수 있습니다. 실제로 Ctrl-Alt-F1 ~ F6 에서 제공되는 리눅스 가상콘솔은 \$TERM 값이 linux 이고 gnome-terminal 같은 경우는 xterm 으로 gnome-terminal 에서 설정한 색상은 가상콘솔 에서 다르게 보입니다.

tput 명령으로 설정을 하면 현재 터미널의 terminfo 정보를 이용해 값을 설정하므로 터미널 종류가 다른 환경에서도 동일하게 적용될 수 있습니다.

```
# foreground color 적용
tput setaf {color숫자}

# background color 적용
tput setab {color숫자}

# Style 속성 적용
tput bold - Set bold mode
tput dim  - turn on half-bright mode
tput smul - begin underline mode
tput rmul - exit underline mode
tput rev  - Turn on reverse mode
tput smso - Enter standout mode (bold on rxvt)
tput rmso - Exit standout mode

tput sgr0 - Turn off all attributes ( '\e[0m' 와 같은 기능 )

# Color 값
0 - Black
1 - Red
2 - Green
3 - Yellow
4 - Blue
5 - Magenta
6 - Cyan
7 - White
```

tput 명령은 매 실행시마다 terminfo 라이브러리를 조회하므로 속도가 느리다는 단점이 있습니다. 그러므로 먼저 사용되는 속성값들을 변수에 저장해 놓고 사용하는 것이 좋습니다.

사용예)

```

color_reset=$(tput sgr0)
color_f_red=$color_reset$(tput setaf 1)
color_b_blue=$color_reset$(tput setab 4)
color_s_bold=$color_reset$(tput bold)
color_my1=$color_reset$(tput setaf 1)$(tput setab 4)
color_my2=$color_reset$(tput bold)$(tput setaf 1)$(tput setab 4)

echo "$color_f_red    red foreground"
echo "$color_b_blue    blue background"
echo "$color_s_bold    bold style"
echo "$color_my1        red foreground, blue background"
echo "$color_my2        bold style, red foreground, blue background"

echo $color_reset

# 다음에 이어지는 스트링에는 속성이 적용되지 않게 reset 합니다.
$(tput sgr0)

# 먼저 reset 한 후 속성을 적용합니다.
$(tput sgr0)$(tput setaf 1) ...

```

스크립트로 전체 색상 프린트 해보기

```

#!/bin/bash

for fgcolor in {0..7} ; do
  for bgcolor in {0..7}; do
    for attr in sgr0 bold dim smul rev; do
      echo -n "$(tput $attr)$(tput setaf $fgcolor)$(tput setab $bgcolor)"
      echo -n " $attr;$fgcolor;$bgcolor "
      echo -n "$(tput sgr0)"
    done
    echo
  done
done
echo
done

```

Prompt 설정

Shell 은 \$PS1 변수 값을 이용해 prompt 를 표시할때 다음의 과정을 거칩니다.

1. prompt escape sequences 처리

2. eval echo \"\"\$PS1\""

예를들어 이전 명령의 종료상태값을 prompt 에 표시하기 위해 `PS1="hello \w $? > "` 와 같이 double quotes 을 이용한다면 이때 이미 `$?` 변수값이 확장되어 대입된 상태가 되므로 항상 같은 값을 표시하게 됩니다. 그러므로 매번 prompt 가 표시될때 마다 새로운 `$?` 변수값이 표시되게 하려면 single quotes 을 사용하여 설정해야합니다.

```
AA=hello
PS1="$AA \w '$?' > "
```

또한 prompt escape sequences 중에 하나인 `\$` 는 double quotes 에서 `$` 로 escape 되므로 올바르게 적용하기 위해서는 single quotes 을 사용합니다.

```
AA=hello
PS1="$AA \w '$? \$ ' "
```

color escape sequences 같은 non-printing 문자를 prompt 에 사용할 경우는 항상 `\[\]` 으로 감싸줘야 커서의 위치션이 올바르게 설정됩니다.

```
AA=hello

color_reset="\[\e[0m\]"
color_f_red="\[\e[0;31m\]"

PS1="$AA \w $color_f_red'$?'"$color_reset" '\$ '

color_reset="\[${tput sgr0}\]"
color_f_red="$color_reset\[${tput setaf 1}\]"

PS1="$AA \w $color_f_red'$?'"$color_reset" '\$ ' "
```

Prompt Escape Sequences

| Code | Description |
|------------|--|
| \a | an ASCII bell (07) 문자 |
| \d | "Weekday Month Date" 포맷의 날짜 (e.g., "Tue May 26") |
| \D{format} | strftime(3) 함수에서 format 스트링이 사용되고 결과가 표시됩니다. { } 는 꼭 사용해야 되며 \D{} 와같이 empty 일 경우는 현재 로케일 설정에 따라 시간이 표시됩니다. |
| \e | an ASCII escape (033) 문자 |
| \h | the hostname 에서 첫번째 ' . ' 까지의 부분 |
| \H | the hostname |
| \j | 현재 background jobs 의 개수 |
| \l | 현재 shell 의 terminal device 이름 (tty5) |
| \n | newline |
| \r | carriage return |
| \s | 현재 shell 이름 (the basename of \$0) |
| \t | 현재 시간 (24-hour HH:MM:SS 포맷) |
| \T | 현재 시간 (12-hour HH:MM:SS 포맷) |
| \@ | 현재 시간 (12-hour am/pm 포맷) |
| \A | 현재 시간 (24-hour HH:MM 포맷) |
| \u | 현재 사용자의 username |
| \v | the version of bash (e.g., 2.00) |
| \V | the release of bash, version + patch level (e.g., 2.00.0) |
| \w | 현재 working directory (\$HOME 은 '~' 로 표시) |
| \W | 현재 working directory 에서 basename 만 표시 (\$HOME 은 '~' 로 표시) |
| !\ | 이번 명령의 history number |
| \# | 이번 명령의 command number (터미널을 열고 enter 로 명령을 사용할때 마다 하나씩 올라감) |
| \\$ | effective UID 가 0 이면 '#' 그렇지 않으면 '\$' |
| \nnn | 8 진수 문자 |
| \\ | a backslash |
| \[| non-printing 문자 sequence 의 시작 (가령 color escape sequence 를 적용할때 사용해야됨) |
| \] | non-printing 문자 sequence 의 종료 |

Tips

대입 연산을 할 땐 변수를 quote 하지 않아도 된다.

변수값에 개행문자가 있거나 공백이 중복되어 사용되었을 경우 quote 하지 않고 사용하게 되면 단어분리에 의해 포맷이 유지가 되지 않습니다. 그런데 예외가 있습니다. 바로 `=`, `+=` 대입 메타문자를 사용할 때인데요. 대입 메타문자는 변수를 quote 하지 않아도 포맷을 그대로 유지해 줍니다.

```
$ AA="I
> like
> winter      and      snow"

$ echo $AA      # 변수를 quote 하지 않으면 공백과 개행이 유지되지 않는다.
I like winter and snow

$ BB=$AA        # 대입연산 에서는 변수를 quote 하지 않아도 된다.

$ echo "$BB"     # 공백과 개행이 유지 된다.
I
like
winter      and      snow

-----

$ ARR=(11 22 33 44 55)

$ AA=${ARR[*]}   # array 원소들이 공백으로 분리돼 있지만 정상적으로 대입된다.

$ echo "$AA"
11 22 33 44 55
```

명령에 선행하는 대입 연산

명령에 선행해서 대입 연산을 할 경우 해당 변수는 해당 명령에만 적용이 되고 명령이 종료된 후에는 사라집니다. `read` 명령은 IFS 값을 이용해 읽어들이는 라인의 필드를 분리하는데 다음과 같이 명령 앞에서 변경하여 사용하면 종료후에 IFS 값을 복구하는 과정을 거치지 않아도 됩니다.

```
$ echo -n "$IFS" | od -a      # read 명령 사용전 IFS 값
00000000 sp ht nl

# IFS 값을 read 명령에 한해 일시적으로 ':' 로 사용
$ IFS=: read -ra arr <<< "Arch Linux:Ubuntu Linux:Suse Linux"

$ echo "${arr[1]}"           # 정상적으로 필드가 분리 되었다.
Ubuntu Linux

$ echo -n "$IFS" | od -a      # read 명령 사용후 IFS 값
00000000 sp ht nl           # 명령 사용 후 값이 변하지 않았다.
```

대입한 값은 명령이 실행돼야 적용된다.

다음은 대입 연산이 적용되지 않는 경우인데요. 이유는 echo 명령이 시작하기 전에 \$A \$B \$C 변수확장이 일어나기 때문입니다.

```
$ A=1 B=2 C=3
$ A=4 B=5 C=6 echo $A $B $C    # 대입한 값이 적용되지 않는다.
1 2 3
```

다음과 같이 eval 명령을 사용하면 해결할 수 있습니다.

```
$ A=1 B=2 C=3
$ A=4 B=5 C=6 eval 'echo $A $B $C'
4 5 6
$ echo $A $B $C                # 명령 종료 후에는 원래 값으로 복귀
1 2 3
```

활용하여 array 값을 다룰때도 사용할 수 있습니다.

```
#!/bin/bash

r_num=1

set -f
while read -r record; do
    IFS=':' eval 'fields=($record)'
    IFS='@' eval 'echo ">>> ${fields[*]}"'
    echo record $r_num has ${#fields[@]} fields
    let r_num++
done < <( cat datafile )
set +f
```

Recommand Sites

- <http://wiki.bash-hackers.org>

This wiki is intended to hold documentation of any kind about the GNU Bash. The main motivation was to provide human-readable documentation and information to not force users to read every bit of the Bash manpage - which is hard sometimes. However, these docs here are not meant as newbie tutorial.

- <http://www.gnu.org/software/bash/manual>

Bourne-Again SHell manual

- <http://mywiki.woledge.org/BashGuide>

Bash Guide

- <http://www.tldp.org/LDP/abs/html>

Advanced Bash-Scripting Guide

- <http://www.tldp.org/LDP/Bash-Beginners-Guide/html>

Bash Guide for Beginners

- <https://wiki.kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/index.html>

고급 Bash 스크립팅 가이드

- <https://wiki.kldp.org/HOWTO/html/Bash-Prog-Intro-HOWTO/>

BASH 프로그래밍 입문 하우투

- <http://www.catonmat.net/sitemap>

- <http://www.shellcheck.net>

automatically detects problems with sh/bash scripts and commands.

- <http://explainshell.com>

write down a command-line to see the help text that matches each argument

- <https://regex101.com/>

Test regular expression