기본 기능부터 명령어 사용법까지

유닉스 이론과 실습 3판

Chapter 13. 배시 셸 프로그래밍

목차

1. 셸 스크립트
2. 셸 변수 사용하기
3. 사용자로부터 입력 받기
4. 연산자
5. 제어문
6. 함수
7. 디버깅

학습목표

- 다양한 셸 변수를 이해하고 활용하는 방법을 익힌다.
- 사용자로부터 입력을 받아 스크립트 파일에서 처리하는 방법을 익힌다.
- 다양한 연산자와 문자열 테스트, 파일 테스트를 활용하는 방법을 익힌다.
- 조건문과 반복문의 사용 방법을 익힌다.
- 함수를 이용해 스크립트 작성하는 방법을 익힌다.
- 스크립트 실행 오류를 찾아 수정하는 방법을 익힌다.

교재에 없지만 꼭 알아야 될 내용

1. 파일 시스템의 개념

- 파일 시스템의 개념
  - 파일 시스템(file system)은 운영체제에서 파일을 저장하는 데 쓰이는 수단이다. 운영체제가 파티션이나 디스크에 파일들을 저장하기 위한 방법들이고 자료 구조이다
  - 파일을 저장하는 데 사용되는 파티션이나 디스크를 가리킬 때 사용되기도 한다
  - 파일시스템은 운영체제마다 다르게 설계되어 있으며, 그 성능 또한 차이가 있다
    - 리눅스 : ext4라는 파일시스템을 사용
    - 솔라리스 : ufs라는 파일시스템을 사용
  - 파일시스템의 개별 파일이 디스크에 저장되는 방법을 이해하기 위해서는 i-node와 슈퍼블록(super-block)의 개념을 알아야 한다

1. 파일 시스템의 개념

i – node

- 리눅스는 파일을 생성할 때, 두 가지의 절차를 거친다. 먼저 파일의 데이터를 저장하기 위한 디스크 내의 공간을 확보한다. 그리고 난 후 파일에 대한 기본 정보를 저장하기 위해 i-node(혹은 index node)라는 구조를 만든다
- i-node에는 리눅스가 파일을 사용하는 데 필요한 모든 정보가 들어 있다
- 리눅스는 모든 i-node를 큰 표(table)에 보관한다
- 표 안에서 각각의 i-node는 i-number(혹은 indexnumber)라고 부른다
- 디렉토리 안에는 실제 파일이 들어 있는 것이 아니고, 모든 디렉터리는 파일의 이름과 i-number만을 가지고 있을 뿐이다. 그 안에는 이름의 목록과 각 이름에 대한 i-number가 있다

1. 파일 시스템의 개념

- i – node
  - i-node 의 내용

정 보	의 미
모드	파일의 타입과 허가 마스크(mask)
링크 카운트	i-node 번호와 함께 entry를 가지고 있는 디렉터리 수
사용자 ID	파일의 소유자 ID
그룹 ID	파일의 소유 그룹
크기	파일의 크기
엑세스 시간	파일의 최근 액세스 시간
mod 시간	파일의 최근 갱신일
i-node 시간	i-node구조를 마지막으로 수정한 시간
블록 목록	파일의 첫 번째 세그먼트를 가지고 있는 디스크 블록의 번호목록
간접 목록	다른 블록들의 목록

- 슈퍼블록
  - 슈퍼블록은 디스크에 저장되는 가장 중요한 정보이다. 이것은 디스크의 헤드와 실린더 등의 수, i-node 목록의 헤드, 그리고 자유 블록에 대한 정보를 가지고 있다.
  - 슈퍼블록은 모든 블록 그룹에 똑 같은 복사본을 갖도록 시스템이 자동으로 만든다. 따라서 파일 시스템이 손상되었을 때 이러한 슈퍼블록들을 이용하여 복구가 가능하게 된다.

파일의 종류

2. 파일의 종류

- 일반 파일
  - 좁은 의미의 파일은 일반 사용자가 작성한 문서, 프로그램, 자료 등을 말한다. 이 파일들은 대부분의 사용자들이 작업을 할 때 이용하는 것이다
    - 일반 파일의 예
      - ◇hello.c : C 언어 소스를 담고 있는 일반파일
      - ◇hello.hwp : 한글 워드프로세서로 작성된 문서 파일

- 디렉터리 파일
  - 디렉터리 파일은 디렉터리에 포함된 다양한 파일들에 대한 위치, 크기, 생성한 시간 등의 정보를 가지고 있다.
  - 디렉터리는 항상 다음의 두 파일을 기본적으로 가지고 있다

문 자	의 미
.	현재의 디렉터리
..	한 단계 위의 디렉터리

  - 모든 사용자는 홈 디렉터리라는 자신만의 디렉터리를 가지고 있다. 사용자가 시스템에 로그인을 하면, 자동적으로 홈 디렉터리로 위치하게 된다

2. 파일의 종류

- 특수 파일
  - 리눅스 시스템에서는 주변장치로부터 데이터를 입력받거나 출력하려면 /dev 디렉터리에 존재하는 입출력장치 파일들을 사용한다. 이러한 /dev 디렉터리내의 입출력장치 파일들을 특수 파일이라 하며, 특수 파일은 문자 특수 파일과 블록 특수 파일로 나눌 수 있다.

    - 문자 특수 파일
      - 문자 특수 파일은 버퍼가 없는 장치로부터 한번에 한 문자씩 입출력을 하기 위해 쓰이는 파일이다.
      - 키보드와 모니터 같은 단말기는 문자 특수 파일을 이용한다.

    - 블록 특수 파일
      - 블록 특수 파일은 일정한 크기의 묶음으로 자료를 입출력하는 장치에서 쓰이며, 버퍼 기술을 사용해 자료 전송 효율을 높인다
      - 하드 디스크장치는 문자 또는 블록 특수 파일을 이용한다.

2. 파일의 종류

- 소켓
  - 소켓(socket)은 두 호스트 컴퓨터 사이의 전달을 담당하는 응용 프로그램 인터페이스(API:Application Program Interface)이다. 다시 말해서, 소켓은 네트워크의 입출력을 담당한다.
  - 소켓을 사용하려면, 소켓을 만들고 로컬호스트와 원격호스트의 주소를 설정하면 된다. 그러나 연결되지 않은 소켓으로 호스트 간에 통신하는 방법도 있다.
  - 연결 소켓의 경우 연결이 이루어진 두 지점간의 자료를 전송해 준다. 그러나 비연결 소켓의 경우에는 매번 목표 지점을 정해주어서 전송한다.

2. 파일의 종류

- 네임드 파이프
  - 네임드 파이프(named pipes)는 프로세스 간 통신을 수행하기 위해 만들어진 파일로 두 프로세스간의 데이터를 중개해주는 역할을 한다.
  - 데이터를 네임드 파이프에 쓰면 보내어지고, 네임드 파이프에서 데이터를 읽어서 받는 것이다. 네임드 파이프에서 데이터는 선입선출 방식인 FIFO(First-In First-Out) 방식으로 다루어진다.

- 심볼릭 링크
  - 일반적으로 링크(link)라고 하면 심볼릭(symbolic) 링크 혹은 소프트(soft) 링크를 말하며 윈도우의 바로 가기 기능이나 단축 아이콘을 생각하면 이해하기 쉽다. 즉, 실제 파일의 내용을 그대로 둔 채 그 파일을 가리키도록 만드는 것을 말한다.
  - 링크를 사용하는 주된 이유는 불필요한 파일복사를 줄이고 파일시스템을 보다 기능적으로 사용하기 위함이다
  - 심볼릭 링크의 장점은 디렉터리도 파일처럼 링크할 수 있다는 점이다
  - 링크를 삭제할 때는 링크만 삭제되며, 링크가 가리키는 원본 파일은 그대로 존재한다. 그반대인 경우도 마찬가지이다.

2. 파일의 종류

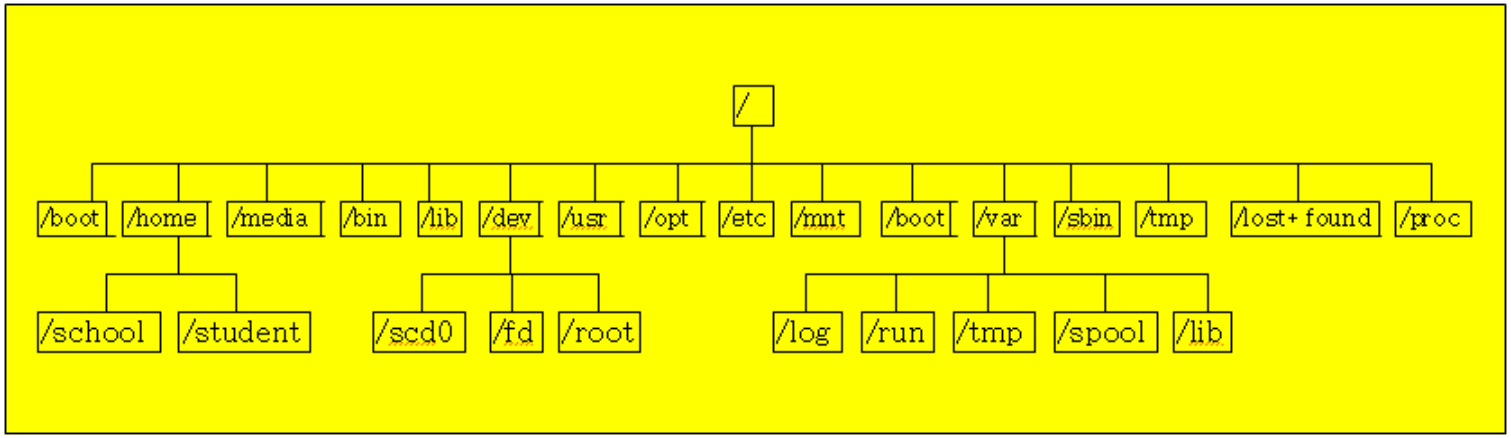
- 하드 링크
  - 심볼릭 링크가 단순히 원본에 대한 정보만을 가지고 있는 데 비해 하드 링크는 원본을 복사한 사본을 생성한다.
  - 심볼릭 링크와 마찬가지로 링크로 접근을 하거나 원본으로 접근을 할 경우 파일의 내용이 수정되었다면 원본과 하드 링크 된 파일이 모두 수정되어 항상 같은 내용을 유지한다.
  - 자원을 공유하되 데이터를 안전하게 관리할 목적일 때 사용된다.

파일 시스템의 구조

3. 파일 시스템의 구조

- 파일 시스템의 구조
  - 리눅스를 설치하면, 상당히 많은 디렉터리가 자동으로 생성된다. 이러한 디렉터리는 대부분의 유닉스들이 유사하다.

  - 리눅스에서의 기본적인 디렉터리 구조



3. 파일 시스템의 구조

파일 시스템의 구조

- 리눅스의 기본적인 디렉토리 구조

디랙토리	내 용
/	가장 위쪽의 디렉터리는 루트 디렉터리(root directory)라고 하고 /(슬래쉬)로 표현한다.
/bin	binaries의 약자이며, 실행 파일들이 모여 있다. 이 디렉터리에는 필수적인 많은 프로그램들이 포함되어 있다.
/dev	/dev 안의 파일들은 디바이스 드라이버들이다. 이것들은 디스크 드라이버, 모뎀, 메모리 등과 같은 시스템 디바이스나 자원들을 액세스하는 데 사용된다.
/etc	시스템 설정 파일, 프로그램, 유틸리티 등 다양한 프로그램들이 있으며 대부분의 파일들은 관리자에 의해 사용되는 것이다. 시스템의 전반에 걸친 설정 파일들 및 초기 스크립트 파일들이 있다.
/home	/home은 일반 사용자들의 홈 디렉터리로 할당하여 사용하는 공간이다.
/lib	공유 라이브러리 이미지를 포함하고 있다. 이 파일들은 일반적으로 사용되고 많은 프로그램에서 호출되는 코드들을 포함하고 있다. 공유 될 수 있는 루틴을 단독으로 포함하지 않고, 그 루틴을 일반 장소인 /lib 안에 저장함으로서 많은 사용자들이 공유할 수 있도록 되어 있다.
/proc	가상 파일 시스템이다. 이 디렉터리의 내용들은 시스템에서 운영되고 있는 다양한 프로세서들에 관한 내용과 프로그램에 대한 정보를 포함하고 있다.
/tmp	어떤 프로그램들은 특정 데이터가 임시 파일 안에 저장되기를 원한다. 이런 데이터들이 위치하는 장소이다.

3. 파일 시스템의 구조

파일 시스템의 구조

- 리눅스의 기본적인 디렉토리 구조

디랙토리	내 용
/usr	응용 패키지들이 설치되어 있는 디렉터리이다. 마이크로소프트 Windows의 Program Files 디렉터리와 같은 역할을 한다고 생각하면 이해하기 쉬울 것이다.
/opt	Add-On 소프트웨어 패키지가 설치된다.
/mnt	마운트 디렉터리 라고 하며, 각종 입출력장치와 연결할 때 사용한다. 마운트하지 않은 상태에서는 빈 디렉터리로 존재하지만 입출력장치를 마운트하게 되면 해당 파일 시스템의 내용이 그대로 이곳에 놓이게 된다.
/var	이 디렉터리 아래에는 시스템 작동 중 변경되는 파일들이 담겨 있다. 로그 파일이나 스풀 파일들이 그것들인데, 즉 다른 시스템과 공유가 되지 않음을 의미한다.
/lost+found	파일 시스템의 이상 유무를 진단하고 복구하는 프로그램인 fsck(FileSystem Check)에 의해서 사용되는 디렉터리이다. 손상된 파일이나 디렉터리를 /lost+found 디렉터리로 연결한 뒤에 오류를 수정하게 된다.
/media	시디롬이나 플로피 디스크 혹은 USB 메모리 등이 임시로 마운트 되는 디렉터리이다.
/boot	리눅스 시스템 부팅에 관련한 모든 파일이 포함되어 있다.
/sbin	Standalone bimary 디렉터리이며 시스템 운영에 필요한 명령어들을 포함하고 있다. 슈퍼유저만이 사용할 수 있는 halt, shutdown 등의 명령어들로 구성되어 있다.

계층 구조

디렉토리 계층 구조

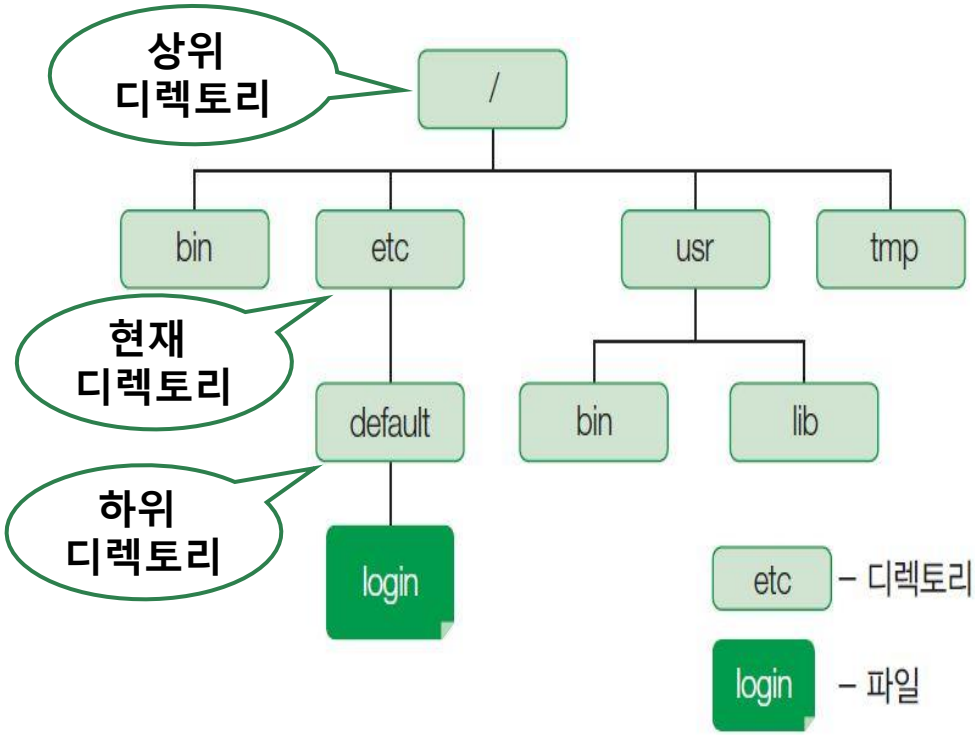
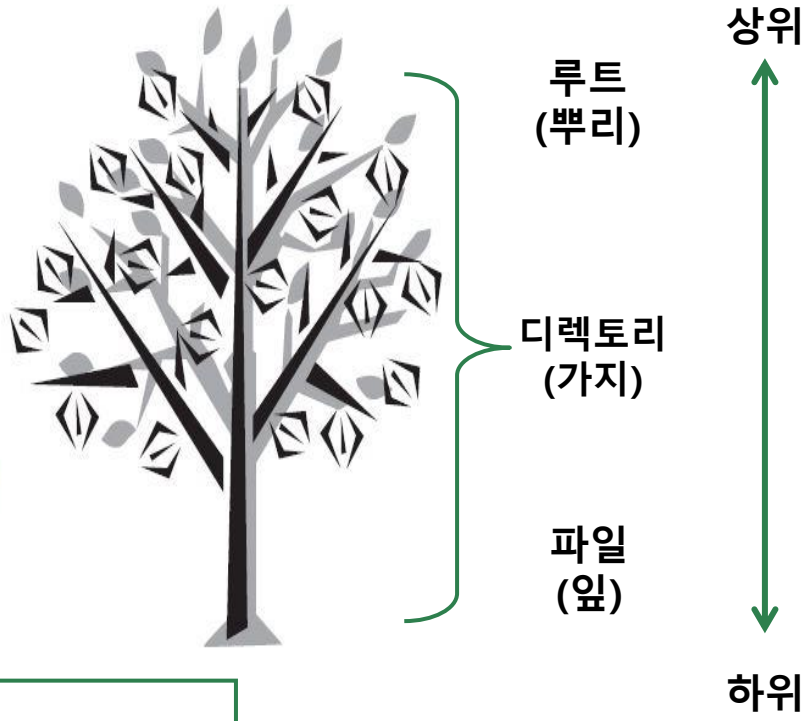


그림 2-1 디렉토리와 나무 대비



홈 디렉토리 : ~
현재 디렉토리 : .
상위 디렉토리 : ..
하위 디렉토리 : 이름

절대 경로와 상대 경로

❖ 경로

- 파일 시스템에서 특정 파일의 위치

❖ 절대경로

- 루트 디렉토리를 기준으로 함
- 루트 디렉토리부터 특정 파일까지 가는데 거치는 모든 디렉토리의 이름 표시
- 항상 / 로 시작

❖ 상대 경로

- 시현재 위치를 기준으로 함
- 하위로 내려갈 때는 디렉토리의 이름을, 상위로 올라갈 때는 .. 추가
- 슬래시 이외의 문자로 시작
- 같은 파일의 상대 경로라도 현재 위치에 따라 달라짐'

절대 경로와 상대 경로

❖ 디렉토리 계층 구조가 다음과 같고, 현재 디렉토리가 user1일 때

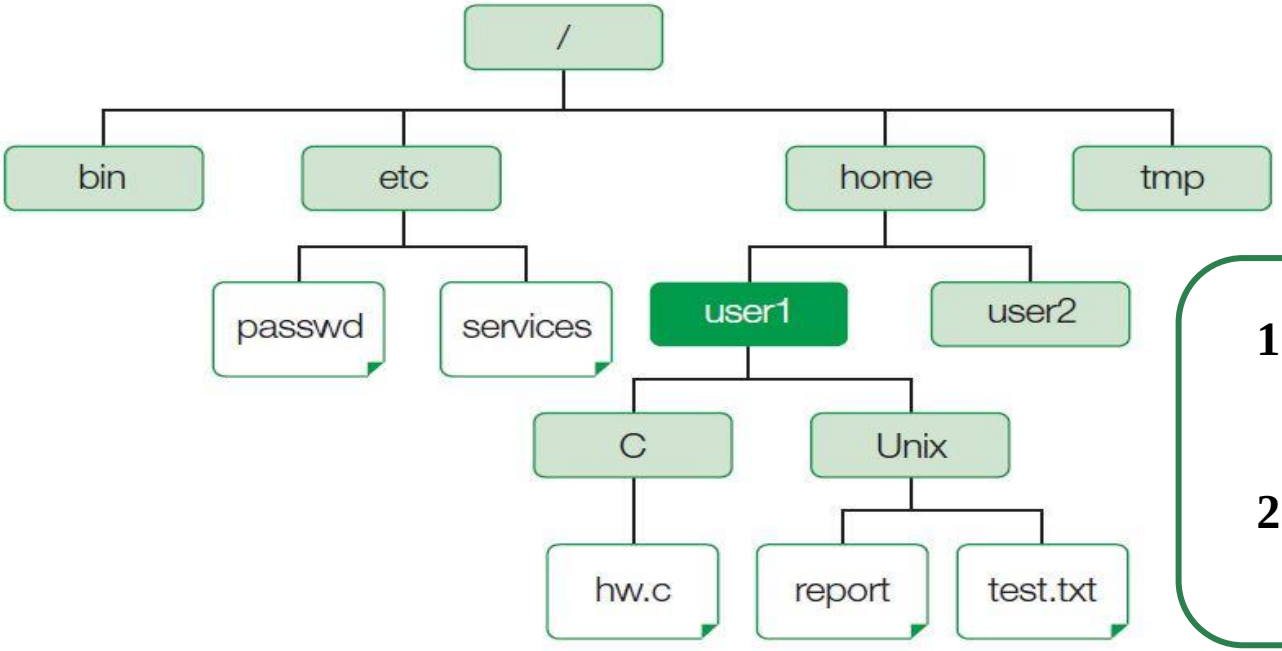


그림 2-2 디렉토리 계층 구조의 예

1. Unix의 절대경로 :
/home/user1/Unix

2. Unix의 상대경로 :
Unix

user2의 절대경로와 상대 경로는?

파일과 디렉토리 명명 규칙

파일과 디렉토리 명명 규칙

❖ 명명규칙

- ❖ 255자 / 를 사용할 수 없다.

❖ 사용 방법

- ❖ .을 사용하면 숨김파일
- ❖ 사용 하기 좋은 파일명
  - 알파벳(대소문자 구분), 숫자, 하이픈(-), 밑줄(_), 점(.)
- 사용자제
  - 공백( ), *, &, |, ", ', :, ~, #, $, (,), \ , ;, <, >
  - 쓰는 경우 이름을 따옴표로 감싸거나 모든 특수문자
- 좋은 이름
  - C, helloWorld.c, unix, .secrete, smaple12
- 나쁜 이름
  - *hl, I'am, #77dir, my dir, book₩
- 쓸 수 없는 이름
  - Mydir/, /test, wrong/name

Section 01

셸 스크립트

01. 셸 스크립트

❖ 스크립트?

- "영화나 방송이 대본과 각본 따위의 방송원고" 즉 연기자들이 스크립트에 따라 역할을 수행하는 것.
- 소프트 웨어 에서의 스크립트 : 컴파일 하지 않고 바로 실행하는 언어
- 인터프리터라 불리는 다른 프로그램에 의해 실행되는 프로그램
- 자바 스크립트, Perl, Lua, 파이썬,...

❖ 셸 스크립트

- 셸에서 제공하는 스크립트 언어 이며 셰링 해석해서 실행하는 것.
- 유닉스 명령 + 셸이 제공하는 프로그램 구성 요소

❖ 셸 스크립트 만들기

- 셸 스크립트 파일 이름은 키워드나 앨리어스 ,내장 명령과 같은 이름을 쓰지 않는 것이 바람직 함
  - 셸 명령 실행 순서 : 앨리어스 → 키워드(if, while, until 등) → 함수 → 내장 명령 (cd, echo 등) → 스크립트, 유틸리트 등 PATH 경로에 있는 실행가능한 파일

01. 셸 스크립트

❖ 셸 스크립트 만들기

❖ vi 에디터로 편집

- **예 : test_script**

```
user1@solaris11:~$ mkdir Unix/ch13
user1@solaris11:~$ cd Unix/ch13
user1@solaris11:~/Unix/ch13$ cat -n test_script
     1  #!/usr/bin/bash
     2  # My First Script Program
     3  echo "Hello Unix World."
     4  pwd
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트

❖ 셸 스크립트 실행하기

▪ 셸을 실행하면서 인자로 스크립트 이름 지정

```
user1@solaris11:~/Unix/ch13$ bash test_script     ➡ 셸의 인수로 넘겨주는 방법
Hello Unix World.
/export/home/user1/Unix/ch13
user1@solaris11:~/Unix/ch13$ test_script          ➡ 자체적으로 실행하는 방법
-bash: test_script: command not found
user1@solaris11:~/Unix/ch13$
```

▪ 파일을 직접 실행

```
user1@solaris11:~/Unix/ch13$ chmod +x test_script   ➡ 실행 권한 부여
user1@solaris11:~/Unix/ch13$ test_script
-bash: test_script: command not found
user1@solaris11:~/Unix/ch13$ ./test_script
Hello Unix World.
/export/home/user1/Unix/ch13
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트

❖ 셸 스크립트 종료하기 : exit

exit [n]

❖ 스크립트의 종료

❖ 종료 상태를 $? 변수에 저장

❖ 사용 예 : exit 20

❖ test_exit

스크립트가 종료되는 경우?
- 파일의 마지막 명령을 실행
- exit 문 실행
- n의 범위 : 0~255
- 관행 : 0은 성공 , 1~255 는 오류코드로 사용

```
user1@solaris11:~/Unix/ch13$ cat -n test_exit
     1  #!/usr/bin/bash
     2  # test_exit: exit와 $?를 테스트하는 스크립트
     3  exit 20
```

```
user1@solaris11:~/Unix/ch13$ chmod +x test_exit    ➡ 실행 권한 부여
user1@solaris11:~/Unix/ch13$ ./test_exit           ➡ 실행 권한 부여 후 실행
user1@solaris11:~/Unix/ch13$ echo $?               ➡ 프로그램의 종료 상태 확인
20
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트 구성요소

❖#!

- ❖ 매직 넘버
- ❖ 파일의 가장 처음에 위치
- ❖ 스크립트를 실행할 프로그램 지정
  - 각 셸마다 제공하는 스크립트 언어의 문법이 조금씩 다르므로, 이 스크립트를 실행할 셸을 지정해 주어야 올바로 실행됨

❖예 : test_script

```
user1@solaris11:~$ mkdir Unix/ch13
user1@solaris11:~$ cd Unix/ch13
user1@solaris11:~/Unix/ch13$ cat -n test_script
     1  #!/usr/bin/bash
     2  # My First Script Program
     3  echo "Hello Unix World."
     4  pwd
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트

❖ #!

- 셸이 아닌, 다른 실행 가능한 명령을 지정해 주어도 됨
- more 나 rm 등 명령어를 사용하면 파일의 인자로 전달 되 그 명령이 수행 된다.

❖ 예 : test_sharp

```
user1@solaris11:~/Unix/ch13$ cat -n test_sharp
     1  #!/usr/bin/more
     2  # test_sharp: 스스로를 출력하는 스크립트
     3  # 이 스크립트를 실행하면 자기 자신을 화면에 출력한다.
     4  # 주석문도 모두 출력한다.
     5  echo "This line is printed."
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트

- #
  - 주석 (comment)
  - 프로그램에 대한 설명
  - 행 전체, 또는 행의 일부를 주석으로 처리할 수 있음

- 예 : test_sharp2

```
user1@solaris11:~/Unix/ch13$ cat -n test_sharp2
     1  #!/usr/bin/rm
     2  # test_sharp2: 자기 자신을 삭제하는 스크립트
     3  # 이 스크립트를 실행하면 이 파일은 삭제됩니다.
     4  WHATEVER=2020
     5  echo "This line is not printed."
     6  exit $WHATEVER  # 이미 파일이 삭제되어 의미가 없음
user1@solaris11:~/Unix/ch13$
```

01. 셸 스크립트

❖ 셸 명령

- ❖ 셸의 모든 명령어 스크립느 내용으로 사용 가능
- ❖ 여러 명령을 순차적으로 수행할 때 스크립트 파일로 저장하여 실행
- ❖ 예 : find_script

```
user1@solaris11:~/Unix/ch13$ cat -n find_script
     1  #!/usr/bin/bash
     2  # find_script: /usr/bin                    ➡ 디렉토리에서 스크립트 파일을 검색
     3  file /usr/bin/* | grep script | more
user1@solaris11:~/Unix/ch13$
```

```
user1@solaris11:~/Unix/ch13$ chmod +x find_script
user1@solaris11:~/Unix/ch13$ ./find_script
/usr/bin/2to3:          executable /usr/bin/python2.7 script
/usr/bin/2to3-2.7:      executable /usr/bin/python2.7 script
/usr/bin/2to3-3.4:      executable /usr/bin/python3.4 script
/usr/bin/2to3-3.5:      executable /usr/bin/python3.5 script
```

중략

01. 셸 스크립트

❖ 셸 프로그램

- ❖ 각 셸이 제공하는 프로그램을 위한 구문
- ❖ 셸 변수, 인자처리, 각종 연산자, 제어문 등 포함
- ❖ 스크립트의 예 :

```
user1@solaris11:~$ cat -n /etc/profile | more
     1  #
     2  # Copyright (c) 1989, 2018, Oracle and/or its affiliates. All rights res
erved.
     3  #
     4
     5  # The profile that all logins, except csh, get before using their own .p
rofile.
     6
     7  if [[ -z $TERM && $(tty) == "/dev/console" ]]; then
     8          if [[ $(uname -p) == "i386" ]]; then
     9                  console=$(eeprom console 2>/dev/null)
    10                  case ${console##console=} in
```

중략

Section 02

셸 변수 사용하기

02. 셸 변수 사용하기

❖ 변수 : 프로그램에서 처리하는 다양한 정보를 저장하는 곳

❖ 종류

- 셸 변수(현재 셸에서만 사용 가능), 환경변수(모든 셸에서 사용가능)

❖ 지정 방법

- 변수명 = 값

표 13-1 셸 변수 표현식

형식	의미
${name}	name 값으로 대치한다.
${name:-word}	name이 정의되어 있으면 그 값을, 그렇지 않으면 word에 명시된 값을 사용한다.
${name:=word}	name이 정의되어 있지 않거나 널이라면 word에 명시된 값을 사용한다. name의 값은 영구적으로 대체된다.
${name:+word}	name이 정의되어 있고 널 문자가 아니라면 word 값을 사용한다. 그렇지 않으면 대체가 이루어지지 않는다.
${name:?word}	name이 정의되어 있고 널 문자가 아니라면 그 값을 사용한다. 그렇지 않으면 word를 출력하고 셸로부터 종료한다.

02. 셸 변수 사용하기

❖ 사용 예

```
user1@solaris11:~$ test=cookbook          ➡ ① test 변수에 cookbook 문자열 저장
user1@solaris11:~$ echo $test             ➡ ② test 변수 값 출력
cookbook
user1@solaris11:~$ echo ${test}           ➡ ③ test 변수 값 출력
cookbook
user1@solaris11:~$ echo ${test:-unix}     ➡ ④ test 변수가 정의되어 있으므로 그 값을 출력
cookbook
user1@solaris11:~$ echo ${test1:-unix}    ➡ ⑤ test1 변수가 없으므로 문자열 unix 출력
unix
user1@solaris11:~$ echo ${test:=unix}     ➡ ⑥ test 변수가 정의되어 있으므로 그 값을 출력
cookbook
```

02. 셸 변수 사용하기

❖ 사용 예

```
user1@solaris11:~$ echo ${test1:=unix} ➡ ⑦ test1 변수가 없으므로 unix를 그 값으로 저장
unix
user1@solaris11:~$ echo ${test1:=unix1}➡ ⑧ ⑦에서 test1 변수의 값이 unix로 설정되어 출력
unix
user1@solaris11:~$ echo ${test:+unix}   ➡ ⑨ test가 정의되어 있으므로 unix로 대체되어 출력
unix
user1@solaris11:~$ echo ${test:?unix}   ➡ ⑩ test 변수가 정의되어 있으므로 그 값 출력
cookbook
user1@solaris11:~$ echo ${test2:?unix}  ➡ ⑪ test2 변수가 없으므로 unix 출력 후 스크립트 종료
-bash: test2: unix
$
```

02. 셸 변수 사용하기

- ❖ 셸 변수 문자열 처리하기
- ❖ 변수의 값이 문자열일 때 문자열 내 패턴을 찾아 일부분을 제거하는 표현식
- ❖ 2.x 이전의 배시셸에서는 동작하지 않음

표 13-2 하위 문자열 처리(2.x 이전의 배시 셸에서는 동작하지 않음)

표현식	기능
${variable%pattern}	variable 값 뒤부터 패턴과 일치하는 가장 작은 부분을 찾아 제거
${variable%%pattern}	variable 값 뒤부터 패턴과 일치하는 가장 큰 부분을 찾아 제거
${variable#pattern}	variable 값 앞부터 패턴과 일치하는 가장 작은 부분을 찾아 제거
${variable##pattern}	variable 값 앞부터 패턴과 일치하는 가장 큰 부분을 찾아 제거
${#variable}	variable의 문자 수를 출력

02. 셸 변수 사용하기

- ❖ 셸 변수 문자열 처리 : %, #
- ❖ 사용 예
  - % : 뒤에서부터 패턴과 일치하는 최소 부분을 제거 (%%는 최대부분)

```
user1@solaris11:~$ path1="/usr/bin/local/bin"
user1@solaris11:~$ echo ${path1%/bin}
/usr/bin/local
user1@solaris11:~$ echo ${path1%%/bin*}
/usr
user1@solaris11:~$
```

%% 사용시 지정한 패턴이 변수값 중간에 있다면, 패턴 이후에 임의의 값이 나올 수 있다는 표시로 * 지정해야 함!

02. 셸 변수 사용하기

❖ 셸 변수 문자열 처리 : %, #

❖ 사용 예

▪ # : 앞에서부터 패턴과 일치하는 최소 부분을 제거 (##은 최대부분)

```
user1@solaris11:~$ path2="/export/home/user1/.profile"
user1@solaris11:~$ echo ${path2#/export}
/home/user1/.profile
user1@solaris11:~$ echo ${path2##*/}
.profile
user1@solaris11:~$
```

.## 을 사용할 때는 모든/ 가 포함된 문자열을 제거!
• 이 예는 경로에서 파일명만 추출할 때 사용

02. 셸 변수 사용하기

❖ 셸 변수 문자열 처리 : %, #

❖ 사용 예

- #변수 : 변수에 저장된 문자의 개수 출력
- path1="/usr/bin/local/bin"
- path2="/export/home/user1/.profile

```
user1@solaris11:~$ echo ${#path1}
18
user1@solaris11:~$ echo ${#path2}
27
user1@solaris11:~$
```

02. 셸 변수 사용하기

- ❖ 명령행 인자 처리
- ❖ 명령행 인자 : 스크립트를 실행할 때 인자로 주어진 값
- ❖ 위치 매개 변수
  - 명령행 인자를 저장하는 스크립트 변수
  - 인자의 위치에 따라 이름이 정해짐

표 13-3 포지션 인자

명령행 인자	의미
$0	현재 셸 스크립트의 이름
$1 – $9	명령행에 주어진 1번 ~ 9번 포지션 인자
${10}	10번째 포지션 인자
$#	전체 포지션 인자 개수
$*	모든 포지션 인자
$@	$*와 의미가 같지만 큰 따옴표로 묶으면 의미가 달라짐
"$*"	"$1 $2 $3"를 의미하며 하나의 문자열로 취급됨
"$@"	"$1" "$2" "$3"를 의미하며 각각 독립적인 문자열로 취급됨
$?	최근 실행된 명령의 종료 값
set --	모든 포지션 인자들을 삭제하거나 설정을 해지함

02. 셸 변수 사용하기

❖ 사용 예 : test_position

```
1  #!/usr/bin/bash
2  # test positional parameter
3  #
4  echo '$*' : $*
5  echo '$#' : $#
6  echo '$@' : $@
7  echo $1 $2 $3
8  set --
```

```
user1@solaris11:~/Unix/ch13$ ./test_position one two three
$* : one two three
$# : 3
$@ : one two three
one two three
user1@solaris11:~/Unix/ch13$
```

02. 셸 변수 사용하기

❖ 인용 부호 : 셸 특수 문자의 의미를 없애기 위해 사용

인용 부호	기능	사용법
작은 따옴표 (‘ ’)	모든 특수문자들이 해석되는 것을 막음	$ echo ‘$test’ $test
큰 따옴표 (“ ”)	변수나 명령의 대체만 허용	$echo “$test” 100
역슬래시 (₩)	단일 문자가 해석되는 것을 막음	$echo ₩$test $test

❖ 명령 대체 : 명령 실행 결과를 문자열로 반환

기호	사용법
백쿼터 (` `)	$ echo `date` Sunday, April 15, 2012 11:05:06 AM KST
$(명령)	$ echo $(date) Sunday, April 15, 2012 11:15:11 AM KST

03. 사용자로부터 입력 받기

- ❖ 사용자로부터 직접 입력 : read
- ❖ 셸 내장 명령으로 터미널이나 파일로부터 입력 처리
- ❖ 사용 형식

표 13-4 read 사용 형식

형식	의미
read x	표준 입력에서 한 행을 입력 받아 x에 저장한다.
read first last	표준 입력에서 한 행을 입력받아 첫 번째 단어를 first에 저장하고 나머지를 last에 저장한다.
read –p prompt	prompt를 출력시키고 입력을 기다린다. 입력된 값은 REPLY 변수에 저장한다.

03. 사용자로부터 입력 받기

- ❖ 사용자로부터 직접 입력 : read
- ❖ 사용 예 : test_read

```
user1@solaris11:~/Unix/ch13$ cat -n test_read
     1  #!/bin/bash
     2  #
     3  # read user input
     4  #
     5  read x              ➡ 메시지 없이 입력을 기다림
     6  echo "x: $x"
     7
     8  read x y
     9  echo "x: $x"
    10  echo "y: $y"
    11
    12  read -p "Input: "
    13  echo "input: $REPLY"
user1@solaris11:~/Unix/ch13$
```

03. 사용자로부터 입력 받기

- here 문서를 통한 입력 : <<
- 표준 입력을 사용자로부터 직접 받아들이지 않고 자동 처리
- TERMINATOR가 입력될 때까지 기술된 부분을 키보드 입력으로 처리
  - 키보드 입력의 종료 문자로 사용하는 EOF(^D) 문자를 파일 안에서 사용할 수 없기 때문에 입력 종료를 나타내는 문자열을 지정하여 사용

사용법 :

```
here  문서
명령 << ENDOFDOC
입력
ENDOFDOC
```

03. 사용자로부터 입력 받기

❖ here 문서를 통한 입력 : <<

예 : test_here

```
user1@solaris11:~/Unix/ch13$ cat -n test_here
     1  #!/usr/bin/bash
     2  # test here document
     3  #
     4  cat -n << END                    ➡ ① 종료 문자로 END 지정
     5  This                             ➡ ② 5~8행은 cat 명령에 표준 입력으로 입력
     6  is a test doc
     7  for
     8  here document.
     9  END                              ➡ ③ 입력 종료
user1@solaris11:~/Unix/ch13$ chmod +x test_here
user1@solaris11:~/Unix/ch13$ ./test_here  ➡ ④ 실행 결과 5~8행의 내용 출력
```

04. 연산자

- ❖ 프로그램에서 자료를 처리하는 방법
- ❖ 산술 연산자, 비교 연산자, 논리 연산자, 비트 연산자 제공
- ❖ 수치 연산자 사용시 let 또는 (( )) 사용해야 함

표 13-5 수치 연산자

연산자	의미	사용 예
–	음수(단항 연산)	x=–5
!	논리 부정(not)	((! x < y))
~	비트 반전(not)	~y
*, /, %	곱셈, 나눗셈, 나머지 연산	let y=3 * 5
+, –	덧셈, 뺄셈	let x=x+1
<<, >>	비트 왼쪽 시프트, 비트 오른쪽 시프트	((y = x << 2))
<=, >=, <, >, ==, !=	비교 연산	((x < y))
&, ^, ₩	비트 AND, XOR, OR 연산	let "z = x ^ y"
&&, ||	논리 AND, OR	((x<y || x==3))
=	변수값 지정	let z=1
*=, /=, %=, +=, ==, <<=, >>=, &=, ^=, |=	단축 연산	let z+=1

04. 연산자

❖ 사용 예

```
user1@solaris11:~/Unix/ch13$ a=5
user1@solaris11:~/Unix/ch13$ echo $a
5
user1@solaris11:~/Unix/ch13$ let a = 20     ➡ let에서 공백을 사용 못함
-bash: let: =: syntax error: operand expected (error token is "=")
user1@solaris11:~/Unix/ch13$ let "a = 20"   ➡ 공백을 포함하려면 " "를 사용해야 함
user1@solaris11:~/Unix/ch13$ echo $a
20
user1@solaris11:~/Unix/ch13$ (( a = 30 ))   ➡ (( ))에서는 공백 사용 가능
user1@solaris11:~/Unix/ch13$ echo $a
30
user1@solaris11:~/Unix/ch13$ a=$a*5         ➡ let이나 (( ))을 사용하지 않으면 문자열로 처리
user1@solaris11:~/Unix/ch13$ echo $a
30*5
```

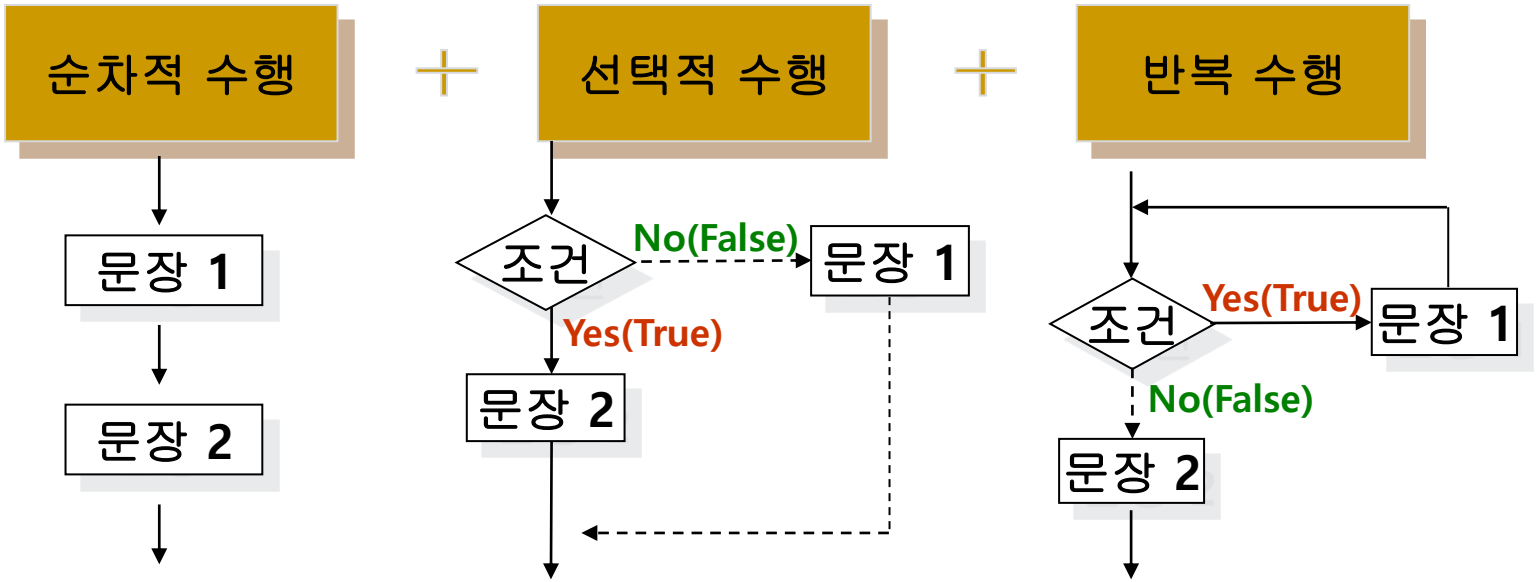
04. 연산자

❖ 사용 예

```
user1@solaris11:~/Unix/ch13$ echo $((5*6))    ➡ 계산 결과를 바로 출력
30
user1@solaris11:~/Unix/ch13$ echo $(( ! 2+3*4 )) ➡ 우선순위에 따라 ! 먼저 수행. 2는 0이 됨
12
user1@solaris11:~/Unix/ch13$ echo $(( 2 << 1 ))  ➡ 왼쪽 shift는 *2와 같음. 2번 shift는 *4
4
user1@solaris11:~/Unix/ch13$ echo $((3 ^ 5 ))    ➡ XOR 연산 결과
6
user1@solaris11:~/Unix/ch13$
```

05. 제어문

- ❖ 프로그램내의 문장 실행 순서를 제어하는 것
- ❖ 선택적 실행문
  - ▪ 프로그램 실행문을 조건에 따라 선택적으로 실행
  - ▪ if , select , case 문
- ❖ 반복 실행문
  - ▪ 프로그램 실행문을 정해진 횟수나 조건에 따라 반복 실행
  - ▪ while , do , for, unitl 문



05. 제어문

엄밀히 말하면 조건 명령을 실행하여
• 그 실행값이 0이 아닌 값이면 then 다음의 명령을 실행하고,
• 0이면 else 다음의 명령을 실행한다.

❖ 선택적 실행문 : if~then~else

❖ 주어진 조건의 참, 거짓 여부에 따라 명령 실행

```
if   조건명령
then
     명령
[ else
     명령 ]
fi
```

❖ 예 : test_if

```
user1@solaris11:~/Unix/ch13$ cat -n test_if
     1  #~/usr/bin/bash
     2  #
     3  # if then else test
     4  #
     5  echo "Input x: "
     6  read x
     7  echo "Input y: "
     8  read y
     9
    10  if ((x < y))
    11  then
    12       echo "$x is less than $y."
    13  else
    14       echo "$y is less than $x."
    15  fi
user1@solaris11:~/Unix/ch13$ chmod +x test_if
user1@solaris11:~/Unix/ch13$ ./test_if
Input x:
100
Input y:
200
100 is less than 200.
user1@solaris11:~/Unix/ch13$
```

05. 제어문

❖ 선택적 실행문 : if~then~elif~else

❖ 주어진 조건의 참, 거짓 여부에 따라 명령 실행

```
if    조건명령1
then
      명령
elif  조건명령2
then
      명령
else
      명령
fi
```

❖ 예 : test_if

```
user1@solaris11:~/Unix/ch13$ cat -n test_elif
     1  #!/usr/bin/bash
     2  #
     3  # elif test
     4  #
     5  echo "Input Score: "
     6  read score
     7
     8  if (( $score >= 90 ))
     9  then
    10      echo "Your score is great."
    11  elif (( $score >= 80 ))
    12  then
    13      echo "Your score is good."
    14  else
    15      echo "Your score is not good."
    16  fi
user1@solaris11:~/Unix/ch13$ chmod +x test_elif
user1@solaris11:~/Unix/ch13$ ./test_elif
Input Score:
70
Your score is not good.
user1@solaris11:~/Unix/ch13$
```

05. 제어문

- ❖ 조건 테스트 : 문자열 연산자
- ❖ 조건 명령에 사용하는 문자열 연산자
- ❖ 내장 명령 [[ ]] 사용

표 13-6 문자열 연산자

연산자	동작
string = pattern string==pattern	string이 pattern과 일치. = 연산자 양쪽에 공백이 있어야 한다. = 대신에 ==를 사용할 수 있다.
string != pattern	string이 pattern과 일치하지 않는다(!= 앞뒤에 공백이 있어야 함).
string	string이 널이 아니다.
-z string	string의 길이가 0이다.
-n string	string의 길이가 0이 아니다.
-l string	string의 길이

05. 제어문

❖ 조건 테스트 : 문자열 연산자

❖ 사용 예

: test_string

```
user1@solaris11:~/Unix/ch13$ cat -n test_string
     1  #!/usr/bin/bash
     2  #
     3  # test string
     4  #
     5  echo "Are you OK (y/n): "
     6  read ans
     7
     8  if [[ $ans = [Yy]* ]]          # y로 시작하는 문자열인지 비교
     9  then
    10      echo "Happy to hear it."
    11  else
    12      echo "That's too bad."
    13  fi
```

05. 제어문

❖ 조건 테스트 : 테스트 플래그

test 플래그	기능
-a file	파일이 존재
-e file	파일이 존재
-L file	심볼릭 링크 파일
-O file	사용자가 file의 소유자
-G file	파일의 그룹 ID가 스크립트의 그룹 ID와 같음
-S file	소켓 파일

05. 제어문

❖ 조건 테스트 : 테스트 플래그

test 플래그	기능
-r file	읽기 가능
-w file	쓰기 가능
-x file	실행 가능
-b file	블록 장치 특수 파일
-c file	문자 장치 특수 파일
-d file	디렉토리 파일
-p file	파이프 파일
-u file	setuid 권한 부여 파일
-g file	setgid 권한 부여 파일
-k file	sticky bit 접근 권한 부여 파일
-s file	파일의 크기가 0이 아님

05. 제어문

❖ 예 : test_flie

```
/user1@solaris11:~/Unix/ch13$ cat -n test_file
     1  #!/usr/bin/bash
     2  #
     3  # test file type
     4  #
     5  echo "Input file name : "
     6  read file
     7
     8  if [[ ! -e $file ]]     # if file exists
     9  then
    10      echo "$file File not exists."
    11  elif [[ -f $file ]]     # if regular file
    12  then
    13      echo "$file is a regular file."
    14  elif [[ -d $file ]]
    15  then
    16      echo "$file is a directory."
    17  else
    18      echo "$file is a special file."
    19  fi
user1@solaris11:~/Unix/ch13$ chmod +x test_file
user1@solaris11:~/Unix/ch13$ ./test_file
Input file name :
aaa
aaa File not exists.
user1@solaris11:~/Unix/ch13$ ./test_file
Input file name :
test_if
test_if is a regular file.
user1@solaris11:~/Unix/ch13$ ./test_file
Input file name :
/dev/null
/dev/null is a special file.
user1@solaris11:~/Unix/ch13$
```

05. 제어문

- 선택적 실행문 : case 문
- 주어진 변수의 값에 따라 실행할 명령 따로 지정
- 변수의 값이 value1 이면 value1부터 ;;을 만날 때까지 명령 실행
- 값의 지정에 특수기호, | (or 연산자) 사용 가능
- 일치하는 값이 없으면 기본값인 * 부터 실행

```
case   변수  in
value1)
     명령 ;;
value2)
     명령 ;;
*)
     명령 ;;
esac
```

05. 제어문

❖ 선택적 실행문 : case 문

❖ 예 : test_case

```
user1@solaris11:~/Unix/ch13$ cat -n test_case
     1  #!/usr/bin/bash
     2  #
     3  # test case
     4  #
     5  echo "Select command : "
     6  read cmd
     7
     8  case $cmd in
     9       [0-9]) # 0~9 digit
    10            date
    11            ;;
    12       cd¦CD) # cd or CD
    13            echo $HOME
    14            ;;
    15       [aA-C]*) # string with a, A, B, C
    16            pwd
    17            ;;
    18       *)
    19            echo "Usage : select command"
    20            ;;
    21  esac
```

```
user1@solaris11:~/Unix/ch13$ chmod +x test_case
user1@solaris11:~/Unix/ch13$ ./test_case
Select command :
9
2019년 05월 18일 토요일 오후  9시 16분 08초 KST
user1@solaris11:~/Unix/ch13$ ./test_case
Select command :
cd
/export/home/user1
user1@solaris11:~/Unix/ch13$ ./test_case
Select command :
Bob
/export/home/user1/Unix/ch13
user1@solaris11:~/Unix/ch13$ ./test_case
Select command :
next
Usage : select command
user1@solaris11:~/Unix/ch13$
```

05. 제어문

❖ 반복 실행문 : for

❖ 리스트 안의 각 값들에 대해 지정한 명령을 순차 실행

```
for 변수 in list
do
    명령
done
```

Tip] list 대신 $(<file)을 사용하면 외부 파일의 내용을 입력으로 받아서 처리!

❖ 예 : test_for

```
6  for num in 0 1 2
7  do
8      echo "Number is $num"
9  done
```

```
user1@solaris11:~/Unix/ch13$ chmod +x test_for
user1@solaris11:~/Unix/ch13$ ./test_for
Number is 0
Number is 1
Number is 2
user1@solaris11:~/Unix/ch13
```

05. 제어문

❖ 명령 행 인자 처리 가능

❖ 예 : test_for3

```
user1@solaris11:~/Unix/ch13$ cat -n test_for3
     1  #!/usr/bin/bash
     2  #
     3  # test for loop
     4  #
     5
     6  for person in $*
     7  do
     8      echo "Hi, $person"
     9  done
user1@solaris11:~/Unix/ch13$ chmod +x test_for3
user1@solaris11:~/Unix/ch13$ ./test_for3 user2 user3 user4
Hi, user2
Hi, user3
Hi, user4
user1@solaris11:~/Unix/ch13$
```

05. 제어문

- ❖ 반복 실행문 : while
- ❖ 조건 명령이 정상 실행되는 동안 명령 반복

```
while  조건명령
do
    명령
done
```

- ❖ 예 : test_while

```
user1@solaris11:~/Unix/ch13$ cat -n test_while
     1  #!/usr/bin/bash
     2  #
     3  # test while loop
     4  #
     5
     6  count=1
     7  sum=0
     8  while (( count<=10 ))
     9  do
    10     (( sum+=count ))
    11     let count+=1
    12  done
    13
    14  echo "Sum(1~10) : $sum"
user1@solaris11:~/Unix/ch13$ chmod +x test_while
user1@solaris11:~/Unix/ch13$ ./test_while
Sum(1~10) : 55
user1@solaris11:~/Unix/ch13$
```

05. 제어문

❖ 반복 실행문 : until

❖ 조건 명령이 정상 실행될 때까지 명령 반복

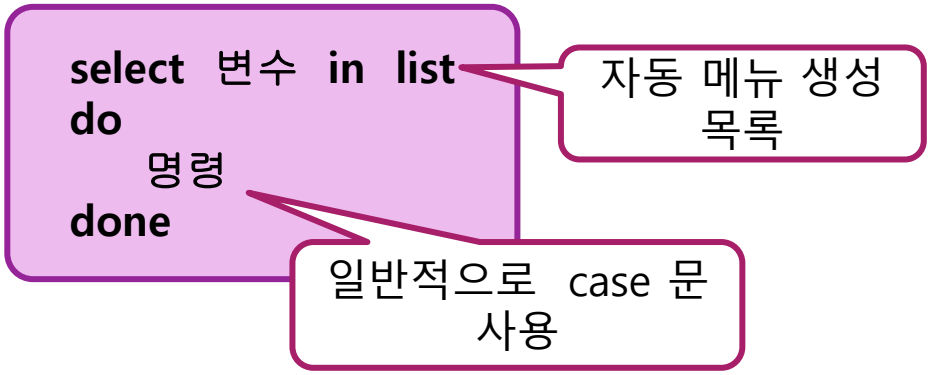
```
until 조건명령
do
    명령
done
```

❖ 예 : test_until

```
user1@solaris11:~/Unix/ch13$ cat -n test_until
     1  #!/bin/bash
     2  #
     3  # test until loop
     4  #
     5
     6  echo "Input name :"
     7  read person
     8
     9  until who | grep $person # > /dev/null
    10  do
    11      sleep 3
    12  done
    13
    14  echo "\007"    # beep
user1@solaris11:~/Unix/ch13$ chmod +x test_until
user1@solaris11:~/Unix/ch13$ ./test_until
Input name :
user2
user2      pts/1        5월 18 21:37     (192.168.122.1)
```

05. 제어문

- 반복 실행문 : select
- 메뉴를 생성할 수 있는 반복 실행문
- list에 지정한 항목을 자동으로 선택 가능한 메뉴로 만들어 화면에 출력해줌
- 사용자는 각 항목에 자동 부여된 번호를 선택
- 사용자 입력은 select와 in 사이에 지정된 변수에 저장
- 보통 case 문과 결합하여 입력 값 처리



05. 제어문

❖ 반복 실행문 : select

❖ 예 : test_select

```
user1@solaris11:~/Unix/ch13$ cat -n test_select
     1  #!/bin/bash
     2  #
     3  # test select
     4  #
     5
     6  PS3="Input command(1~3) :"
     7
     8  select cmd in pwd date quit # pwd=1, date=2, quit=3
     9  do
    10      case $cmd in
    11          pwd) pwd ;;
    12          date) date ;;
    13          quit) break ;;
    14          *) echo "Invalid input, select number" ;;
    15      esac
    16
    17      REPLY=-1   # null
    18  done
```

05. 제어문

- 루프 제어문 : continue
- 루프 안에서 사용
- 이후 실행 순서를 무시하고 루프의 처음으로 돌아가는 명령
- 숫자를 지정하면 중첩된 루프 안에서 특정 루프의 처음으로 돌아갈 수 있음

05. 제어문

- ❖ 루프 제어문 : continue
- ❖ 예 : test_continue

```
user1@solaris11:~/Unix/ch13$ cat -n test_continue
     1  #!/bin/bash
     2  #
     3  # test continue
     4  #
     5
     6  for person in $(< list)
     7  do
     8      if [[ $person == user2 ]]
     9      then
    10          continue
    11      fi
    12
    13      echo "Hello, $person"
    14  done
user1@solaris11:~/Unix/ch13$ chmod +x test_continue
user1@solaris11:~/Unix/ch13$ ./test_continue
Hello, user3
Hello, user4
user1@solaris11:~/Unix/ch13$
```

06. 함수

❖ 함수 : 하나의 목적으로 사용되는 명령들의 집합

❖ 앨리어스와의 차이점

- 조건에 따라 처리 가능
- 인자 처리 가능

```
function  함수이름
{
    명령들
}
```

❖ 예 : trash

```
user1@solaris11:~/Unix/ch13$ mkdir ~/.TRASH
user1@solaris11:~/Unix/ch13$ function trash {
> mv $* ~/.TRASH
> }
user1@solaris11:~/Unix/ch13$
```

06. 함수

❖ 정의된 함수 확인

```
user1@solaris11:~/Unix/ch13$ typeset -f
trahsh ()
{
    mv $* ~/.TRASH
}
user1@solaris11:~/Unix/ch13$
```

❖ 함수 호출

```
user1@solaris11:~/Unix/ch13$ touch a b c
user1@solaris11:~/Unix/ch13$ trash a b c
user1@solaris11:~/Unix/ch13$ ls
user1@solaris11:~/Unix/ch13$ ls ~/.TRASH
a b c
```

06. 함수

❖ 함수의 종료 : return

❖ 함수 종료 조건

- 함수 본문 안의 마지막 문장 실행
- return 문 실행

return [n]

❖ 지정한 값이 함수의 종료 값으로 $?에 저장

06. 함수

- ❖ 함수의 종료 : return
- ❖ 예 : test_add

```
user1@solaris11:~/Unix/ch13$ cat -n test_add
     1  #!/bin/bash
     2  #
     3  # test add function
     4  #
     5
     6  function add {
     7      typeset sum
     8
     9      (( sum = $1 + $2 ))
    10      return $sum
    11  }
    12
    13  add $1 $2
    14  echo "$1 + $2 = $?"
user1@solaris11:~/Unix/ch13$ chmod +x test_add
user1@solaris11:~/Unix/ch13$ ./test_add 1 2
1 + 2 = 3
user1@solaris11:~/Unix/ch13$
```

06. 함수

❖ 함수 삭제 : unset

unset -f 함수명

❖ 정의된 함수를 삭제

❖ 사용 예

```
user1@solaris11:~/Unix/ch13$ unset -f trash
user1@solaris11:~/Unix/ch13$ typeset -f
```

07. 디버깅

❖ 스크립트 실행도중 발생한 오류 수정 방법

❖ 구문 오류

- 셸이 실행도중 구문오류가 발생한 라인번호 출력

- 실행 오류
- 오류 메시지 없이 실행이 안되거나 비정상 종료
- 오류 수정 방법
- bash -x, trap

07. 디버깅

- ❖ 디버깅 : bash -x
- ❖ 가장 간단한 스크립트 실행 오류 수정방법
- ❖ 스크립트의 각 행이 실행될 때마다 화면에 출력됨

```
user1@solaris11:~/Unix/ch13$ bash -x test_while
+ count=1
+ sum=0
+ (( count<=10 ))
+ (( sum+=count ))
+ let count+=1
+ (( count<=10 ))
+ (( sum+=count ))
+ let count+=1
+ (( count<=10 ))
+ (( sum+=count ))
+ let count+=1
+ (( count<=10 ))
(중략)
+ let count+=1
+ (( count<=10 ))
```

```
+ echo 'Sum(1~10) : 55'
Sum(1~10) : 55
user1@solaris11:~/Unix/ch13$
```

07. 디버깅

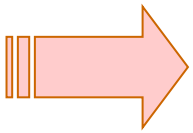
❖ 디버깅 : trap

trap 명령 시그널

❖ 지정한 시그널이 스크립트로 전달될 때마다 지정한 명령 실행

❖ 스크립트의 명령이 한 줄씩 실행될 때마다 DEBUG 시그널이 스크립트로 전달됨

❖ DEBUG 시그널을 받을 때마다 원하는 변수값 출력 가능



스크립트가 실행되는 도중 변수값 확인

07. 디버깅

❖ 예 : test_trap

```
user1@solaris11:~/Unix/ch13$ cat -n test_trap
     1  #!/bin/bash
     2  # test_trap : trap을 테스트하는 스크립트
     3
     4  trap 'echo "$LINENO : count=$count"' DEBUG
     5  count=1
     6  sum=0
     7
     8  while (( count <= 10 ))
     9  do
    10      (( sum+=count ))
    11      let count+=1
    12  done
    13
    14  echo "Sum(1~10): $sum"
```

```
user1@solaris11:~/Unix/ch13$ chmod +x test_trap
user1@solaris11:~/Unix/ch13$ ./test_trap
5 : count=
6 : count=1
8 : count=1
10 : count=1
11 : count=1
8 : count=2
```

```
10 : count=2
11 : count=2
(중략)
8 : count=10
10 : count=10
11 : count=10
8 : count=11
14 : count=11
Sum(1~10): 55
user1@solaris11:~/Unix/ch13$
```