13. Functions

본 셸 스크립트 프로그래밍에서 종종 간과되는 기능 중 하나는 스크립트 내에서 사용할 함수를 쉽게 작성할 수 있다는 점입니다. 이 작업은 일반적으로 두 가지 방법 중 하나로 수행되는데, 간단한 스크립트의 경우 함수가 호출되는 것과 동일한 파일에 함수를 선언하기만 하면 됩니다. 그러나 일련의 스크립트를 작성할 때는 유용한 함수의 '라이브러리'를 작성하고 해당 함수를 사용하는 다른 스크립트의 시작 부분에 해당 파일을 소싱하는 것이 더 쉬운 경우가 많습니다. 이 방법은 나중에 설명하겠습니다. 어떤 방법을 사용하든 방법은 동일하므로 여기서는 주로 첫 번째 방법을 사용하겠습니다. 두 번째 (라이브러리) 방법은 기본적으로 동일하지만 명령어

```
. ./library.sh
```

가 스크립트 시작 부분에 표시됩니다.

셸 함수를 프로시저로 호출할지, 함수로 호출할지 혼동할 수 있는데, 함수의 정의는 일반적으로 단일 값을 반환하고 아무 것도 출력하지 않는 것입니다. 반면에 프로시저는 값을 반환하지 않지만 출력을 생성할 수 있습니다. 셸 함수는 둘 중 하나만 수행하거나 둘 다 수행할 수 있습니다. 일반적으로 셸 스크립트에서는 이러한 것을 함수라고 부릅니다.

함수는 네 가지 방법 중 하나로 값을 반환할 수 있습니다:

- 변수 상태 변경
- 종료 명령을 사용하여 셸 스크립트를 종료
- **return** 명령을 사용하여 함수를 종료하고 제공된 값을 셸 스크립트의 호출 섹션에 반환
- **echo** 출력을 **stdout**으로 반환하며, 이는 c=`expr $a + $b`와 같이 호출자에 의해 포착시킴

종료는 프로그램을 중지하고 반환은 호출자에게 제어권을 돌려준다는 점에서 C와 비슷합니다. 차이점은 셸 함수는 전역 매개변수를 변경할 수는 있지만 매개변수를 변경할 수 없다는 점입니다.

함수를 사용하는 간단한 스크립트는 다음과 같습니다:

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{
USER=$1
PASSWORD=$2
shift; shift;
# Having shifted twice, the rest is now comments ...
COMMENTS=$@
```

```
echo "Adding user $USER ..."
echo useradd -c "$COMMENTS" $USER
echo passwd $USER $PASSWORD
echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein Bob Holness the presenter
add_a_user fred badpassword Fred Durst the singer
add_a_user bilko worsepassword Sgt. Bilko the role model
echo "End of script..."
```

4행은 ()로 끝나는 함수 선언으로 식별됩니다. 그 뒤에는 {가 오고, 일치하는 } 다음에 오는 모든 것이 해당 함수의 코드로 간주됩니다. 이 코드는 함수가 호출될 때까지 실행되지 않습니다. 함수는 읽혀지지만 실제로 호출될 때까지는 기본적으로 무시됩니다.

이 예제에서는 사용자 추가 및 패스워드 명령 앞에 echo가 붙었는데, 이는 올바른 명령이 실행되는지 확인하는 데 유용한 디버깅 기법입니다. 또한 루트가 되거나 시스템에 수상한 사용자 계정을 추가하지 않고도 스크립트를 실행할 수 있다는 의미이기도 합니다!

우리는 셸 스크립트가 순차적으로 실행된다는 생각에 익숙해져 있습니다. 하지만 함수는 그렇지 않습니다. 이 경우 add_a_user 함수는 읽혀서 구문을 확인하지만 명시적으로 호출될 때까지 실행되지 않습니다. 실행은 "Start of script..."라는 echo 문으로 시작됩니다. 다음 줄인 add_a_user bob letmein Bob Holness는 함수 호출로 인식되어 add_a_user 함수가 입력되고 환경에 특정 추가 사항과 함께 실행이 시작됩니다:

```
$1=bob
$2=letmein
$3=Bob
$4=Holness
$5=the
$6=presenter
```

따라서 해당 함수 내에서 $1은 함수 외부에서 $1이 무엇으로 설정되어 있든 상관없이 bob으로 설정됩니다. 따라서 함수 내에서 '원본' $1을 참조하려면 함수를 호출하기 전에 다음과 같이 이름을 지정해야 합니다: A=$1과 같이 함수를 호출하기 전에 이름을 지정해야 합니다. 그런 다음 함수 내에서 $A를 참조할 수 있습니다. 시프트 명령을 다시 사용하여 $3 이후의 매개변수를 $@로 가져옵니다. 그런 다음 이 함수는 사용자를 추가하고 비밀번호를 설정합니다. 이 함수는 해당 효과에 대한 주석을 표시하고 메인 코드의 다음 줄로 제어권을 반환합니다.

변수의 범위

다른 언어에 익숙한 프로그래머라면 셸 함수의 범위 규칙에 놀랄 수도 있습니다. 기본적으로 매개변수 ($1, $2, $@ 등)를 제외하고는 범위 지정이 없습니다. 다음의 간단한 코드 세그먼트를 예로 들어보겠습니다:

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
  x=2 }

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

이 스크립트를 scope.sh a b c로 호출하면 다음과 같은 출력을 제공합니다:

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

함수 내에서 $@ 매개변수는 함수가 호출된 방식을 반영하기 위해 변경됩니다. 그러나 변수 x는 사실상 전역 변수 (global)로, myfunc가 이를 변경했으며 제어가 메인 스크립트로 돌아갈 때에도 이 변경 사항은 여전히 유효합니다.

함수의 출력이 다른 곳으로 파이프되는 경우 함수는 하위 셸에서 호출됩니다. 즉, "myfunc 1 2 3 | tee out.log"는 두 번째에도 여전히 "x는 1"이라고 표시됩니다. 이는 새로운 셸 프로세스가 호출되어 myfunc()를 파이프하기 때문입니다. 이것은 디버깅을 매우 불편하게 만들 수 있습니다. Astrid는 "| tee"가 추가되었을 때 갑자기 실패하는 스크립트가 있었는데, 왜 그래야 하는지 즉시 알 수 없었습니다. 간단한 예로 "ls | grep foo"의 경우, grep이 먼저 시작되어야 하고, ls가 시작되면 그 stdin이 ls의 stdout에 연결되어야 합니다. 셸 스크립트에서는 tee를 통해 파이프할 것이라는 사실을 알기도 전에 셸이 이미 시작되었으므로 운영 체제에서 tee를 시작한 다음 새 셸을 시작하여 myfunc()를 호출해야 합니다. 이것은 실망스럽지만 알아둘 가치가 있습니다.

함수는 호출된 값도 변경할 수 없으며, 스크립트에 전달된 매개 변수가 아닌 변수 자체를 변경해야 합니다. 예시를 통해 이를 보다 명확하게 확인할 수 있습니다:

```
#!/bin/sh

myfunc()
{
  echo "\$1 is $1"
  echo "\$2 is $2"
  # cannot change $1 - we'd have to say:
  # 1="Goodbye Cruel"
  # which is not a valid syntax. However, we can # change $a:
  a="Goodbye Cruel"
}

### Main script starts here

a=Hello
b=World
myfunc $a $b
echo "a is $a"
echo "b is $b"
```

이 다소 냉소적인 함수는 $a를 변경하여 "Hello World"라는 메시지를 "Goodbye Cruel World"로 바꿉니다 .

재귀 (Recursion)

함수는 재귀적일 수 있습니다 . 다음은 팩토리얼 함수의 간단한 예입니다 :

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}
```

```
while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

약속한 대로 이제 셸 스크립트 간에 라이브러리를 사용하는 방법에 대해 간략하게 설명하겠습니다 . 나중에 살펴보겠지만 공통 변수를 정의하는 데에도 사용할 수 있습니다 .

common.lib

```
# common.lib
# Note no #!/bin/sh as this should not spawn
# an extra shell. It's not the end of the world # to have one, but clearer not to.
#
STD_MSG="About to rename some files..."

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=`basename $i $FROM`
    mv $i ${j}$TO
  done
}
```

function2.sh

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

function3.sh

```
#!/bin/sh
# function3.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

여기에서는 두 개의 사용자 셸 스크립트인 function2.sh와 function3.sh가 각각 공통 라이브러리 파일 common.lib를 소싱하고 해당 파일에 선언된 변수와 함수를 사용하는 것을 볼 수 있습니다. 이는 셸 프로그래밍에서 코드 재사용이 어떻게 이루어지는지 보여주는 예시일 뿐입니다.

Exit Codes

종료 코드에 대한 자세한 내용은 튜토리얼의 힌트 및 팁 섹션(14장)의 종료 코드 부분을 참조하세요. 지금은 리턴 콜에 대해 간략히 살펴보겠습니다.

```
#!/bin/sh

adduser()
{
  USER=$1
  PASSWORD=$2
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWORD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

## Main script starts here
```

```
adduser bob letmein Bob Holness from Blockbusters
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
elif [ "$?" -eq "2" ]; then
   echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

이 스크립트는 두 개의 외부 호출 (useradd 및 passwd)을 검사하고 실패할 경우 사용자에게 알려줍니다. 그런 다음 이 함수는 사용자 추가에 문제가 있는 경우 반환 코드 1을, 패스워드에 문제가 있는 경우 반환 코드 2를 정의합니다. 이렇게 하면 호출 스크립트는 문제가 어디에 있는지 알 수 있습니다.

Revision #1
Created 17 January 2024 04:16:20 by 고기만두 (MeatDumpling)
Updated 17 January 2024 04:16:33 by 고기만두 (MeatDumpling)