Review

- Conditions
- The test, or 'if' Command
- Control Structures
  - if statement
  - if-else-if statement
  - for loop statement
  - while loop statement
  - until loop statement
  - case statement

Functions

- You can define functions in a shell script.
- SYNTAX:
  ```bash
  function_name ()
  {
  statement
  statement2
  ...
  }
  ```
- Function prototypes cannot be placed for calling a function before function definition.
- A function must be defined before the function call.
- Functions with Local Variables

```bash
#!/bin/sh
# function.sh: an example for a function definition
function_hello()
{
  local yourname
  echo -n "What is your name?"
  read yourname
  echo "Hello $yourname"
  echo "Parameter variables for the function_hello are $#"
  echo "Parameters variables for this shell script are $0"
  function_hello you your yours
  echo "your name is $yourname"
  echo "Parameters variables for this shell script are $0"
  exit 0
}
```

Functions with Local Variables

- We can declare local variables within a function by using `local` keyword which in only in the function scope.
- If the local variable has same name as a global variable, it overlays that variable, but only within the function.
Functions with Local Variables

```bash
#!/bin/sh
# local.sh for testing local variable
# local variable is in the function scope
yourlocation() {
    local mylocation
    echo "where are you now?"
    read mylocation
    mylocation = "Salisbury"
    echo "I am still in $mylocation"
    exit 0
}
mylocation
yourlocation
```

Function with Return Value

- In bash, we can define a function with a return value.
- A function can return **0 (true)** or **1 (false)** as a result.

```bash
#!/bin/bash
# myname.sh: demonstrate a function with return value
yes_or_no() {
    echo "Is your name $* ?"
    while true
        echo -n "Enter yes or no"
        read x
        case "$x" in
            y | yes ) return 0;;
            n | no ) return 1;;
            *) echo "Answer yes or no"
            esac
    done
}

if yes_or_no "$*"
then
    echo "Hi $*, nice name"
else
    echo "Never mind"
fi
exit 0
```

```bash
#!/bin/bash
# returnf.sh: demonstrate a function with return value
add() {
    local rval=`expr $1 + $2`
    return $rval
}
subtract() {
    local rval=`expr $1 - $2`
    return $rval
}
multiply() {
    local rval=`expr $1 \* $2`
    return $rval
}
divide() {
    local rval=`expr $1 / $2`
    return $rval
}
add 2 3
a=$?
# $? Hold the status currently returns
echo $a
subtract 4 1
a=$?
echo $a
multiply 2 3
echo $?
divide 4 2
echo $?
exit 0
```

Recursive Function

- Does bash permit recursion?
- Yes, but it's so slow since it uses big memory space.
- Running a script with recursion could possibly lock up your system!
### Recursive Function

```bash
#!/bin/bash
# fact2.sh
# find factorial of given command line arg
# since system does not provide enough space for $?, echo $? Can display wrong answer!
# you can assign return value to variable and display

factorial() {
    local result=1
    for i in $(seq 1 $1); do
        let result=result*$i
    done
    echo $result
}

# display usage
if [[ $# -eq 0 ]]; then
    echo "Usage: need a number for input"
    exit 1
fi

# call factorial
factorial $1
exit 0
```

### Recursive Function

```bash
#!/bin/bash
# fact1.sh
# find factorial of given command line arg
# since system does not provide enough space for $?, echo $? Can display wrong answer!
# you can assign return value to variable and display

factorial() {
    local result=1
    for i in $(seq 1 $1); do
        let result=result*$i
    done
    echo $result
}

# display usage
if [[ $# -eq 0 ]]; then
    echo "Usage: need a number for input"
    exit 1
fi

# call factorial
factorial $1
exit 0
```

### Recursive Function

```bash
#!/bin/bash
# fact.sh
# find factorial of given command line argument with recursion

factorial() {
    local i=$1
    local f

    if [[ $i -le 2 ]]; then
        echo $i
    else
        let f=$(i-1)
        let f=$(factorial $f)
        let f=$(( f * i ))
        echo $f
    fi
}

# display usage
if [[ $# -eq 0 ]]; then
    echo "Usage: need a number for input"
    exit 1
fi

# call factorial
factorial $1
exit 0
```

### Break command

- Exit from a for, while or until loop

**SYNTAX**

```
break [n]
```

- If `n` is supplied, the `n`th enclosing loop is exited. `n` must be greater than or equal to 1.

```bash
for myloop in 1 2 3 4 5 do
    echo "$myloop"
    if [ "$myloop" -eq 3 ]
    then
        break
    fi
done
```
Continue Command

- Resume the next iteration of an enclosing for, while, until, or select loop.

**SYNTAX**

```
continue [n]
```

- If `n` is supplied, the execution of the `n`th enclosing loop is resumed.
- `n` must be greater than or equal to 1.

---

**Other Commands: eval**

**Indirect Variable References**

- `eval`: allows you to evaluate argument expressions.

```bash
foo=10
x=foo
y=${foo}  # it is same as y='$foo'
echo $y

foo=10
x=foo
y=${foo}  # it is same as y='$foo'
echo $y
```

```bash
x=foo
y=${x}  # it is same as y='$x'
echo $y
```

---

**Other Commands: export**

- Set an environment variables.
- Mark each name to be passed to child processes in the environment.

---

**Other Commands: exit n**

- `exit` command cause the script to exit with exit code.

- In shell programming,
  - `exit 0`: exit with success
  - `exit 1 ~ 125`: exit with an error
  - `exit 126`: reserved code the file was not executable
  - `exit 127`: A command was not found
  - `exit 128`: a signal occurred

---

```bash
#!/bin/sh
# import.sh gr1, gr2, gr3 might be imported from parent script
echo "$gr1 $gr2 $gr3"
exit 0
```

```bash
#!/bin/sh
# import1.sh gr1 might be imported from it's parent
echo "$gr1 Birthday!"
exit 0
```

---

```bash
#!/bin/sh
# export.sh: export variables to subscripts
export gr1="Happy"
export gr2="Fear"
export gr3="Year"
./import.sh
./import1.sh
exit 0
```

```bash
#!/bin/sh
# export.sh: export variables to subscripts
export gr1="Happy"
export gr2="Fear"
export gr3="Year"
./import.sh
./import1.sh
exit 0
```
Other Commands: expr

- Evaluate expressions, evaluates an expression (arithmetic, logical) and writes the result on standard output.

- Syntax
  - `expr expression`

```
#!/bin/sh
# exprArith.sh: expr example with arithmetic operators
a=5+3
echo "a is $a"
echo "Arithmetic Operators"
echo "-------------------"
a=`expr 5 + 3`
echo "new a = 5 + 3 = $a"
e=`expr $a + 1`
echo "a + 1 = $a"
a=`expr 5 % 3`  # same as let a=5/2
# modulo
echo "5 mod 3 = $a"
exit 0
```

Other Commands: printf

- Syntax
  - `printf "Format strings", parameter list`

```
printf "\%d, \%s, \%c\"a\" 2 "Hi" 'a'
```

Other Commands: printf

- Escape Sequence in printf
  - \b : backspace
  - \f : form feed
  - \n : newline
  - \r : carriage return
  - \t : tab
  - \v : vertical tap
Other Commands: set

- The set command sets the parameter variables for the shell script.
- Position parameter can change inside a script.
- We can use the result of a command as an input to other commands.

Other Commands: shift

- The shift command shifts all positional parameter variable down by one. $4 become $3, $3 become $2...
- The previous value of $1 is discarded $0 (name of script) remains.