# Testing & Scripting

Vineet Seth

24th March 2006

## Two types of testing Approach

- **White Box** Requires Understanding of the Code
  - Memory Leaks
  - Logical Flow
  - Syntactical and Semantic elements
- **Black Box** Requires Understanding of the System
  - System installation
  - System Configuration
  - System runtime behavior
  - Regression testing

## Associated with testing is Reporting component

Any testing performed should be able to produce reports for further analysis

- bugzilla - Web based End-to-End Bug tracking tool
- GNATS - Bug tracking system
- gurgle - Report generator Language can be connected to database or text output of Testing cycle
- + tons of other tools

## Limitation of White Box testing

- Depends upon knowing the Language and its problem areas
- The tools used are specialized tools meant for the Language
- White box testing might not be very feasible for a project e.g OS, compliers, Assembly coding projects
- Project with Performance issues or Distributed components may have problem

## Memory Leaks - Major component of White Box Testing

- Memory Leak - malloc/free mismatch
- Overrun the end of dynamically allocated heap memory
- Under-run a memory buffer
- Freeing of same buffer multiple times
- Clobbering of statically allocated stack and global memory

## Using mcheck

- Comes with libc
- Easy to Use
- Environment variable **MALLOC_TRACE**=filename
- Call to mtrace installs handlers for function malloc, realloc, free
- Cannot detect memory corruption errors
- Also good for trace when program does not run in debug mode
- **mtrace** filename can be used to analyze traces

```
#include <mcheck.h>
 int
 main(int argc, char **argv)
 {
   #ifdef DEBUGGING
     mtrace();
   #endif
 }
```

## Using mpr - a memory allocation profiler

- Is almost equal to mcheck, except that it gives the amount of leaks
- Link your program with **-lmpr** (libmpr.a)
- Dynamically load library using LD_PRELOAD
- Environment variable **MPRFI** is to be set .eg export MPRFI='cat a.log'
- Run the program, a.log is formed
- **mprleak** < a.log to see the leak

## Using Electric Fence

- Does not detect memory leaks, but detects Buffer overruns
- Works by allocating memory after the requested allocation that the process is not permitted to access
- The offending program will Seg fault, and then the core file can be analyzed
- Link the program with **-lefence**
- **EF_ALIGNMENT**=0, so that smaller overruns can be caught
- **EF_PROTECT_BELOW**=1 for buffer under-run
- **EF_PROTECT_FREE**=1 for detecting access to free memory

## Using Valgrid

- Valgrid can detect the the memory leaks, buffer overruns etc. . .
- It can run directly with the executables
- For usage just run **valgrid** "executable name"
- It detects uninitialized memory, besides under-run, overrun, leaks
- Simple to use, easy to understand output

## Problems with powerful tools

- Execution becomes slower
- Time to analyze the log becomes bigger
- Tools assume the architecture e.g CPU instructions, MMU unit etc, so cannot be used portability or for cross platform development

## Using LCLint/SPLint

- Statical checking of C code
- Good as a review replacement
- Gives warning and reason on possible memory leaks
- Gives warning on unused variable, invalid flows, common security vulnerabilities
- Expands upon gcc -Wall option

## Using C error Handling

- assert (assert.h)
  - In C code put assert(int EXPRESSION), after any logical block
  - Designed for detecting Internal inconsistency, not invalid inputs
- errno (errno.h)
  - **perror**(const char *MESSAGE) - reports error for system call failures based on errno
  - fputs(**strerror**(errno),stderr) - Is the a better method than perror, but perror is more portable

## Unit testing

- Using GDB
  - Make a dummy main file
  - Link your .o file with main to produce executable
  - Call gdb <exec> , break on main , and break on your function
  - run and the set (function name), to single trace test function
  - make use of **gdb -x** script exefile for automation
  - Script file is same as commands used during gdb session
- Other unit testing tools e.g CUnit,CPPunit, JUnit...
  - Provides set of assets to check your functions
  - Provides framework to maintain test-cases
  - More realistic than GDB, as GDB executes in virtual environment e.g Signals, Memory, File limit...

## Problems with White Box testing

- Validates only part of the framework, hence cannot validate the requirements
- Adds to overhead to the developing framework, so cannot be streamlined into the development process
- Slows the expandability of the projects
- Requires larger investment, as almost equal amount of work is required

## When to use White Box Testing

- Special purpose Libraries whose interface needs to be validates e.g Math Libraries
- Certain special algorithms, which acts as interfaces e.g High speed string manipulation functions
- Optimized code to validate the performance for that segment of code, e.g scheduler

## Minimize the White Box Testing work

- Add -DDEBUG flags to Makefile and in the Code log the debug statement to output file
- Use LOGLEVEL approach to get the detailed logs
- Use -lefence or -lmbr method during linking when -DMDEBUG flag is given

## Problems with Black Box testing

- No general consensus on how to to test, due to varying nature of projects
- Too many different frameworks to test

## Importance of scripting Language

Scripting languages are easy to learn and have a faster work around time.
Scripts are easy to modify and support

## Standards

- POSIX 1003.3 standard, defines what testing framework need to provide
- TET (Testing Environment Toolkit) by Open Software provides testing framework
- POSIX provided a set of assertions, on basic of which the test will pass of fail
- The output of the test case can be

| | |
|---|---|
| PASS | *Test passed* |
| FAIL | *Test Failed* |
| UNRESOLVED | *Test executed in an unexpected manner* |
| UNTESTED | *Test was not run* |
| UNSUPPORTED | *No support for tested case* |
| XPASS/XFAIL | *Not in Posix, as there is no notion of expected success/failure* |

## Writing Test-Cases

- Each framework poses its own restrictions on how the test case results are presented
- Framework should does not pose rustications on how the test case should be validated
- Framework do support certain languages that are better suited, though
- TeT framework(TeTware) is well suited for perl, DejaGNU is well suited for TCL
- Most of these framework have binding for different languages
- Knowledge of scripts is a added benefit if you use any framework, or even write you own framework
- Most common scripting languages used are Shell(ksh/bash), perl, tcl, python

## Autoconf support for check target

- Autoconf framework support check target by default
- make check in earlier autoconf example did not produce any output, but the target was valid
- To add make a test dir, generally names are of type tests, testsuite
- Variables supported are **TESTS**, **TESTS_ENVIRONMENT**, **XFAIL_TESTS**

## Example - adding test to previous autoconf example

```
# mkdir test
# Add following to Makefile.am
TESTS = hello-1
EXTRA_DIST = $(TESTS) # Since we want to execute it
                      # and be able to distribute it
# Append test/Makefile to AC_CONFIG_FILES(configure.ac)
# Append test to SUBDIRS in Makefile.am
# Run autoreconf -i --force
```

## Shell script

- touch hello-1; chmod +x hello-1
- make; make check
- Output

```
PASS: hello-1
==================
All 1 tests passed
==================
```

- Actual Shell script

```
#!/bin/sh
trap 'rm -fr $tmpfiles' 1 2 3 15
tmpfiles="hello-test1.ok"
cat <<EOF > hello-test1.ok
Hello World
EOF
tmpfiles="$tmpfiles hello-test1.out"
: ${HELLO=../src/hello}
${HELLO} > hello-test1.out
diff hello-test1.ok hello-test1.out
result=$?
rm -f $tmpfiles
exit $result
```

## Features of Scripts for Testing

- Should be able to handle processing of text files
- Should have good report generation capabilities
- Should have well supported Regular Expression support
- Should have good level of Sting manipulation functions
- Should be able to execute external commands easily
- Should be easy to modify and be maintainable
- Should be as independent as possible, and should not be too much dependent upon the exiting installed system
- Should be easy to port, and well connected with other languages for expandability e.g Can be called from C functions (using libraries), Should be able to write functions in C and connect to the language

## Languages discussed

- Idea is to be able to understand the code written in the language, rather than able to write the code
- With the coverage given, should be able to understand a project with medium complexity
- More Build up of the language is required by using the manual

## Perl Introduction

- Perl is closest to Natural Language, so sources written can be very difficult to Understand
- Variables : Use qw(quote word) instead of "" e.h @a = qw(ab a), Other quotes q(single quotes), qq(Double quotes), qx(back quotes)

| **Scalar** | **Array** | **Associative Array** |
|---|---|---|
| $scalar = value | @array = ("val1" , "val2" , . . . ) | %hash = ( "key1" => "val1", "key22" => "val2", . . . ) |
| ($a, $b) = ($b, $a) | | |
| ($a, $b) = @array | $array[index] = val | $hash{$key} = value |
| Undef: $x = **undef** | Size: $#array | Delete: **delete**($hash{$key} |
| Check **defined**($x) | Size : $s = @array | **exists**($hash{$key} |
| $x = **join**(" ", @arr) | **sort** [func] @array | ($key, $value) = **each**(%hash) |
| @arr = **split**(" ", str) | **reverse** @array | @array = **keys** %hash |

- Arrays Operations : **push**(@array, $val), $x= **pop**(@array), **unshift**(@array, $val), $x = **shift**(@array), **chop** – removes last element in each array element , **chomp** – removes last null element in each array element
- **sort** { $a **cmp** $b } @array (for string) , **sort** { *a***<=>**b } @array
- String: $s = **[r]index**("string","search string", skip), $s = **substr**("string", start, length) and **substr**("string", start, length) = "New string"
- functions: **sub** func { arg1=**$_[0]**; arg2=**$_[1]**; statements; **return** value} Private variable : **my** ($var) = data; **local** ($var) = data ( Also available to sub-functions)

## Using Perl

- Special Variables usage is very important to understanding Perl syntax **$_** ($ARG – Default input and pattern searching space) , **$.** ($NR or $INPUT_LINE_NUMBER) **$/** ($RS or $INPUT_RECORD_SEPERATOR), **$,** ($OFS or $OUTPUT_FIELD_SEPERATOR), **$** ($ORS or $OUTPUT_RECORD_SEPERATOR), **$!** ( $ERRNO or $OS_ERROR), **$@** ($EVAL_ERROR), **$$** ( $PID or $PROCESS_ID), **$0** ($PROGRAM_NAME), **$ARGV** ( Name of program), **@ARGV** ( Program Arguments), **$&** ($MATCH), **$‘** ($PREMATCH), **$’** ($POSTMATCH), **$|** ($OUTPUT_AUTOFLUSH), **@INC** (Include directory), **%ENV** (Environment variables), **%SIG** (Signal handler)
- Control Structures

| | | | |
|---|---|---|---|
| **if** (expr) { statements } **elsif** (expr) { statements } **else** { statements } | **while** (expr) { statements } **until** (expr) { statements } **last** - break | **for** (initial_expr; text_expr; re-init_exp) { statements } **next** - continue | **foreach** $var (@array) { statements } **do** { statements } **while** or **until** (expr) **redo** - Reevaluate |

- Can create labels **abc:** and last, next, redo can jump to it, even outside the loop
- For simple statements : if (expr) {expr2}: expr1&&expr2 or !expr1||expr2 or expr2 if (expr1) Can be used for other constructs using until, while var = expr1 ? assign : assign

## Other Operations in Perl

- File : **open**(HANDLE, ""[> or » or < or « ]filename") or **die/warn** "String $!" Operations on file **close**(HANDLE). Check operation -r, -w, -x , -e , -s , -z on filename
  while (**<HANDLE>**) { . . . }; print HANDLE "String" , **<>** = <STDIN>
- Execute Commands : **system**("command") or back-quotes ‘**command**‘ or open(HANDLE,"**command**|") for reading "|**command**" for writing or Using **fork() & exec("command")** e.g exec("command") unless fork() ;
- Globing : @array = **glob**("file*") or @array = **<file*>**
- Regular Expressions: if (/**abc**/) { statement } ( executes on **$_**) otherwise use $var **=~** /RE/, for substitution use s/RE/"string"/
- References : $refs = \$val or $refa = \val or $refh = \%val. $reff = \&func. Use reference like a pointer -> e.g $$refs, $refa->[0], $refh->$key, $reff->(@args)
- To use external package : **use** PackageName or **use** Path::PackageName. For specific function use **use** PackageName qw(func1 . . . ). Second method **require** Module (Routines should be used with package Name)
- Date and Time: use Time::localtime; $tm = localtime; ($DAY, $MONTH, $YEAR) = ($tm->mday, $tm->mon, $tm->year). For high resolution timer **use** Time::HiRes qw(gettimeofday)
- Sleep: **sleep(value)** or **select(undef,undef,undef,value)**
- Signals: $SIG{’INT} = ’my_han’ | ’DEFAULT’; sub my_han { . . . }

## Advanced features of Perl

- Writing Modules (**.pm**) file

```
package Path::Module;
use vars qw(@ISA @EXPORT @EXPORT_OK
            %EXPORT_TAGS $VERSION);
use Exporter;
@ISA = qw(Exporter)
@EXPORT = qw($var %hash @arr \&funci1); # Autoexported
@EXPORT_OK = qw(\&func2); # Exported on request
%EXPORT_TAGS = ( );
$var;
%hash;
@arr;
 sub func1 { ... }
 sub func2 { ... }
1;
```

- Hashes and array can have multiple levels of nesting, Full OO feature in Perl is an example of this
- Object Oriented:

```
 package myclass;
 # Registers the class with bless
 sub new { my $class = shift; my $self = { }; return bless $self,$class }
 sub func1 { my $self = shift; # Always the 1st arg
     $self->{VAR} = shift if @_; return $self->{VAR};
 }
sub DESTROY { my $self = shift; ... }
use myclass;
my $inst = myclass->new(); $inst->func1("value")
```

## Advanced features

- Overloading

```
use overload ( '+'  => \&func );
# Any operator can be linked to xyz function
sub func { my ($left, $right) = @_;
           ... ;
            return $ansclass;
         }
```

- If Regular Expression is not the feature for which Perl is used for then it is **tie** function, which makes Perl worth using. tie binds the variable to a class which provides access methods for that variable. So accessing a tied variable automatically calls the method calls in that class. Methods constructors should be of form **TIESCALAR, TIEARRAY TIEHASH, TIEHANDLE**
**tie VARIABLE, CLASSNAME, LIST** ; $object = tied VARIABLE; **untie VARIABLE** Methods like FETCH (When variable is accessed), STORE (When variable is assigned), DESTORY(When variable is destroyed) For Handle methods WRITE, PRINT, PRINTF, READ, READLINE, GETC, CLOSE, DESTROY exists e.g tie *STDIN, 'MyFileLogger' <– In this case logger takes care of the details tie *STDIN, 'MyFileLogger', File1, File2, File3

```
#!/usr/bin/perl
use Tie::Tee;
use Symbol;
@handles = (*STDOUT);
for $i ( 1 .. 10 ) {
    push(@handles, $handle = gensym());
    open($handle, ">/tmp/teetest.$i");
}
tie *TEE, 'Tie::Tee', @handles;
print TEE "This␣lines␣goes␣many␣places.\n";
```

## TCL introduction

- It is an interpreted language
- Setting a variable is `set variable ?value?` e.g set a "hello"
- All variables are treated a string
- Quotes are either `" "` `( allows substitution )` or `{ }` `(does not allow substitution)`
- Results of the command are represented in [ ] e.g set a 2;set b [expr {0x2 * sqrt($a)}] , while { [gets $file line] >= 0 } { . . . }
- **if** expr ?then? body **elseif** expr ?then? body **?else?** body e.g if {$x ==2} {puts $x } {puts "no"}
- **switch** string { pattern1 body1 ?pattern2 body2?...?default bodyN? } e.g switch $x "O" "puts 0" "1" "puts 1" "default" "puts Rest""
- **`while`** `test body` and **`for`** `start test next body`
  Every test should be placed within braces, as everything is a command and goes though same substitution phase
  for incrementing: **`incr`** `varname ?increment?` can be used
  Usual **break** and **continue** can be used
- For functions use **`proc`** `name {required {default1 a} {default2 b} args} body`. If no return statement is there then return value of the last command is the return value
  Variables used are within the local scope, can be changed to global scope by using **global** or **upvar** (pass by reference)

# Operation performed using TCL

- File operations: set fileid [**open** filename ?access? ?permission?], **close** $fileid , for binary use **fconfigure** $fileid -translation binary
  Other operations **gets**, **puts**, **read**, **write**, **seek**, **tell**, **flush**, **eof**
- For running external programs use **open** ¨cmd"
- Making list: set list { {item1} {item2} {item3}} or set list [**split** "stings" "charsep"] or set list [**list** arg1 arg2 . . . ]
  **llength** list (length of list), **lindex** list index (returns element at index), **foreach** varname list body (iterates the list) **concat**, **lappend**, **linsert** list index arg1 ?arg2, **lreplace** list,
  **lsearch** list globpattern, **lsort** list, **lrange** list first last
- String subcommands : **string length** $string, **string index** string index, **sting range** string first last
  **string compare** s1 s2, **string first** s1 s1, **string match** pattern string **tolower** string, **toupper** string, **trim** string ?chars?, **format** formatstr args
- Regular Expressions: **regex** ?switches? exp string ?matchVar? ?subMatch1 ... subMatchN? e.g
  set result [regexp {([A-Za-z]+) +([a-z]+)} "There is way" match sub1 sub2]
  # result=1, match="There is", sub1=There sub2=is
  **regsub** ?switches? exp string subSpec varName e.g regsub "no" "There is no way" "a" sample2
- Associative arrays: set name(value) "Data" and operations on array like **exits, names, size, get, set**

## Using command-line arguments

**$argv0** – File name, **$arg1 . . . $argN** – arguments, **$env** associative array e.g $env(PATH)

## Extending TCL – Using Packages

- Using external packages – **package require package-name version**
  Calling function mypack::fun1
  Another method is to source the file **source** filename
- Making Packages – **package provide mypack** and create NameSpace **namespace eval::mypack{namespace export fun1 fun2; variable myvar}**
  proc ::mypack::fun1{} { . . . }

## Using Expect

- Automation tool for interactive applications
- Usage is either **#!/usr/bin/expect** or in tcl program **package require Expect**
- **spawn** – Starts a program, **expect** – Wait for some input (used like switch statement), **send** – Send some input, **interact** – allow user interaction
- Variables **$spawn_id** – Spawned process id, **$user_spawn_id** – Self process ID, **$timeout** – Timeout variable(default 10sec), **$expect** – Associative array e.g $expect(output)

## Expect example – To be explained line by line

```
#!/usr/bin/tcl
package require Expect;
proc telnet { host } {
  spawn telnet $host
  expect {
        "login*"    {send "guest\r\n" ; exp_continue ;}
         "Password*" {send "guest\r\n" }
         timeout    {exit}
         eof        {exit}
         default    {exit}
   }
   send "ls\r\n"
   # PAIN statement - Problem is that it is line buffer
   expect -re "ls.*\.*@.*" { send_user "# : $expect_out(buffer)  #" };
   return $spawn_id;
}

set host1 "192.9.201.84"
set host2 "192.9.201.63"

set ses1 [ telnet $host1  ]
set ses2 [ telnet $host2  ]

interact  {
    -i $ses2
       ~l { return }
   }
```

## Example Continued

```
#set timeout -1 # For infinite timeout
while (1) {
     send_user "\nid, ls, date or q> "
     expect {
        -i $user_spawn_id
             "id\n" { send -i $ses1 "hostname\n" ;
                      send -i $ses2 "hostname\n" }
             "ls\n" { send -i $ses1 "ls\n" ;
                      send -i $ses2 "ls\n" }
             "date\n" { send -i $ses1 "date\n" ;
                        send -i $ses2 "date\n" }
             "q" { break }
             "\n" { }
             timeout { puts "Timeout occurred ..." ; continue; }
        -i $ses1 -re ".+" { send_user "\n**1> $expect_out(buffer)" }
        -i $ses2 -re ".+" { send_user "\n**2> $expect_out(buffer)" }
      }
}
```

## Using TK

- TK is the GUI extension, either use #!/usr/bin/wish or package require Tk

  *package require Tk is broken in Redhat 9.0*

## Using TK

- TK can communicate to any of the Messaging systems, Windows, XWindows, KDE etc. . .
- TK can connect with expect to perform GUI testing
- TK is based on **.** language e.g .menu.file.open
- GUI widgets like **frame**, **entry**, **label**, **scale**, **listbox**, **checkbutton**, **radiobutton**, **button**, **scrollbar**. **text**, **panedwindow**, **canvas**, **spinbox** exists
- Dialogs box like Message Box – **tk_messageBox**, Open file – **tk_getOpenFile**, Choose Directory – **tk_chooseDirectory**, Colour box – **tk_chooseColor** exists
- Keyboard and Mouse events can be binded using **bind** e.g bind .button <Double-ButtonPress-1> { proc call }
- Display will happen once all the components created have been packed using **pack** or **grid** command
- Example

```
#!/usr/bin/wish
proc push_button {} {
        .ent insert 0 "Hello "
}
frame .frm -relief groove
label .lab -text "Enter name:"
entry .ent
button .but -text "Push Me" -command "push_button"
pack .lab -in .frm
pack .ent -in .frm
pack .frm
pack .but
```

## Python Basics

- Object oriented interpreted language
- Forces indentation
- Variable assignment is **variable = value**, value can either be number, string – multiple assignment is possible e.g a,b = 2,3
- Stings can be index like in C, e.g str[4], str[4:12], str[4:], str[:4], str[-1], str[:-1]
- Lists **list = [ it1 , it2, it3 . . . ]** – reference e.g list[0], list[0:2] = [it1, it2] – replace, list[0:2] = [] – remove, list[1:1] – insert, Nested list is possible
- expr can be chained e.g a < b < c m Boolean operations like (a and (not b)) or c, assignments cannot happen in expr
- Loops

  **if** expr:
  commands
  **elif** expr:
  commands
  **else:**
  commands

  **while** expr:
  commands
  **break**

  **for** val **in** list[:]:
  commands
  **continue**

  **for** val **in range(param)**:
  **pass** #blank statement
- **range** is a general function Usage `range(till) or range(from,to) or range(from,to, iterator)`
- Functions **def** func(arg1, arg2=default, *args): <– args used like variable arguments, range(*args). Valry can be returned using **return**. Called like func(val1,val2), func(val1), func(arg2=val,arg1=val)

## Operations performed using Python

- File operations: f = **open(filename, type)**, **f.read()**, **f.readline()**, **f.readlines()** <– return list per line or use **for i in file**
  Other Operations: **f.write(string), f.tell(), seek(offset, from_what), f.close()**
  Helper Module: **pickle** Usage: pickle.dump(x,f), x = pickle.load(f)
- Executing External commands: **import os** and **out = string.join(os.popen(file).readlines())** or **os.spawnvp(os.P_WAIT,file,[list of args])**
- Modules : Writing modules : Name **module.py**, Inmodule.py declare global variable **__name__**
  Using Modules: import module and for function access use module.f1() or **from** module **import** f1, f2 or * and use directly like f1()
  **import** is a command way e.g import sys (for system) and use sys.argv, sys.stderr.write , import os (for os), import shutil ( for system utils) e.g shutil.copyfile or shutil.move. Depending upon the code can use other modules like math, datetime, zlib etc. . .
- Wild-cards: import glob and **glob.glob**(*.*)
  for Regular Expressions : import re and **re.findall**(r'expr', string) , **re.sub**(r'expr', r'expr', 'string')
- list methods: **append(x), extend(L), insert(i,x), remove(x), pop([i]), index(x), count(x), sort(), reverse(), del, set** (sets in set operation e.g data = set(list), lst_elm **in** data = true/false

## Other Python Elements

- Some of functional programming elements : list = **filter**(func, list,. . . ) or range(from, to), list= **map**(func, list or range(),. . . ), element = **reduce**(func_for_operation_on_2_elem, list)
- Associative array : dictionary arr = { index:value, . . . }, usage : arr[index] = val, arr.keys(), has_key(index), for k,v in arr.iteritems(), Other : zip(list1,list2), reversed(list), sorted(list)
- For formating output use print 'C style input e.g print '%s' % (values,ldots)
- Exception Handling: **use try:** commands **except errtype1:** commands **except errtype2:** . . . **else:** commands To create exceptions use **raise**
- Small and Anonymous functions use **lambda** e.g lambda a,b: a+b
- For Documentation use """ . . . """ in beginning of class,def etc. To get documentation use func.__doc__
- Uses classes : x = myclass(args), For data member access : x.data = val, For Method access: x.f1(args)

```
class myclass(Base1, Base2, . . . ):
""" Doc """
__prv_data = val
def __init__(self, args):
self.data = []
def __iter__(self) and def next(self) # <– or yield(iterators)
def f1(self,args) # 1st arg is always self
commands
```

## 1st Start with #!/bin/sh

- # is comment except magic-number **#!** in 1st line
- Variable assignment **variable=anything** No space before and after = Anything can be a string, number, list - Essentially they are untyped for usage use $variable
- For arithmetic operations use **(( ))** or let or expr e.g

```
a=1
let a=a+1 # also accepts -=,+=, *=, /=, %=
(( a=a+1 )) # and also accepts bitwise operations
a=$((a+1))
a=$(($a+1))
let "a = 2#1010 , b = 0xa" # Use of , and base
a=`expr $a + 1` # Spaces in between and back-quotes
```

- Scripts **exit** with 0 on success

| | |
|---|---|
| 1 | general errors |
| 2 | Misuse of shell builtins |
| 126 | cannot execute command or not executable |
| 127 | command not found |
| 128 | invalid argument to exit |
| 128+n | Fatal signal |
| 255* | exit status out of range |

# test in Shell

- if syntax

```
if list then; [ elif list ; then list; ] ...
[else list;] fi
```

- test syntax - **test** , **[ ]** , **[[ ]]**, for arithmetic **(( ))** also
  - file check operation **-e, -f, -s, -d, -b, -c, -p, -h, -S, -r , -w, -x**
  - file compare operations **-nt, -ot**
  - file integer comparisons **-eq, -ne, -gt, -ge, -lt, -le, <, <= , > , >=**
  - string comparisons **=, ==, !=, >,** < ( " " for [ ] ), **-z, -n**
  - compound comparisons **-a, -o ( for [ ] ), && ,** ||
- Ways to write tests

```
[ "10" –ne "11" ] && echo "Impossible"
if cd /home; then echo "Now␣in␣new␣directory"
[[ –d /vinod ]] || echo "Directory␣does␣not␣exist"
```

- Some special characters **:** (Null command), * (Wild card or multiply), **!** (negate), **( )** (command group), **{ }** (block of code), **>,<,»,«,|** (redirection) **$?** (Exit status), **$$** ( process id), **$!** (Last job run in background), **$_** (argument of the last command executed)
- Positional Param args **$0** (file name), **$1 (first param)**, **$2...$11 $12**, **$#** (No of param), **$*** (All param as 1 word), **$@** (Each param a quoted string)

## Shell Language

- **shift** – shift the positional parameters
- **$IFS** (default to white space), **$LINENO, $FUNCNAME, $PATH, $PPID, $PS1, $PS2, $PS3, $PWD, $REPLY** ( if variable is not given in read command)
- Parameter Substitution :
  If param not set use default :
  **${parameter-default}** - Not declared, **${parameter:-default}** - declared and null
  If param not set, set it to default:
  **${parameter=default}, ${parameter:=default}**
  If param is set use alt value, else use NULL value:
  **${parameter+alt_value}, ${parameter:+alt_value}**
  If param is set use it else print err message:
  **${parameter?err_msg}, ${parameter:?err_msg}**
- Remove: **${string#Pattern}** - shortest match, **${string##Pattern}** - Longest match
  **${string%Pattern}** - shortest match from back, **${string%%Pattern}** - Longest match from back
- Replace: **${string/pattern/replacement}** - Replace first match,
  **${string//pattern/replacement}** - Replace all match
  **${string/#pattern/replacement}** - Front-end replace ,
  **${string/%pattern/replacement}** - Backend replace

## Additional Features

- Array : arr[0]="data"; echo "${arr[0]} ${arr:0}"
  arr=(val0 val1 val2); echo "${arr[@]:1:2} prints val1 val2"
- Length: **${#var}** - str length, ${#*} or ${#@} - no pf positional params, ${#array[*]} ${#array[@]} - no of elements
- Substring Extraction : **${var:pos}** - Variable staring from position, **${var:pos:len}** - Variable expanded staring from position up-to max len
- Indirect reference : **eval var1=\$$var2**
- Loops

| | |
|---|---|
| **for** arg **in** [list]; **do**<br>commands<br>**done** | **for** (( initial_exp; test_exp; reinit_exp ))<br>**do** commands<br>**done** |
| **while** [condition]; **do**<br>commands<br>**done**<br>**break** [n] | **until** [condition]<br>**do** commands;<br>**done**<br>**continue** [n] (out to n outer loop) |

- other statement alternates

| | |
|---|---|
| **Alternate if..else**<br>**case** "$var" **in**<br>"$cond1") commands**;;**<br>. . .<br>***)** commands;; # Default case ;<br>**esac** | **Alternate for and read**<br>**select** variable [**in** list]<br>**do**<br>commands<br>[break];<br>**done** |

## Other features

- C-style operators: **(( a = b<2?0:1 ))**
- Making function : **function func** or **func()**{
  **local** arg1=$1; arg2=$2; commands; **return** val }
- **declare** -r ( readonly), -i (integer), -a (array), -f (functions), -x (export) (typeset for ksh)
- Options processing **getopts** - **$OPTIND** ( Option Index), **$OPTARG** (Option Args)

```
while getopts ":ab:cd" Option; do
   case $Option in
        a ) echo  "a";;
        b ) echo "b with arg $OPTARG" ;;
        c ) echo "c" ;;
        * ) echo "Unimplemented option";;
    esac
done
shift $(($OPTIND - 1 ))
```

- Redirecting : **>** file or **1>** file (stdout to file), » file (Append to file), **&>** file ( both stderr & stdout to file), **2>&1** (stderr to stdout) , **<>** file (Input and output from file) , **<** file (Input from file instead of stdin), **n<&-** (close n input file descriptor, 0 is stdin), **n>&-** (closing n output file descriptor)
- Command execution: back-quote ‘CMD‘ or $(CMD) to get input into a var

## Other features

- Here Document: For controlling interactive Applications
  COMMAND «**InputfromHere**
  . . .
  **InputfromHere**
  Use «**-InputfromHere** for use of Tabs (to increase readability)
- Build in Commands: **echo, printf, read** variable (-t timed , -n no new line, input), **cd, pwd, pushd, popd, dirs, let, eval, set, unset, export, declare, typeset, readonly, source, exit, exec** ( replaces the current process), **shopt, caller** (echo about the caller of that function), **true, false, type, hash, bind**( readline key bindings),**jobs, disown, fg, bg, wait, suspend, logout, times, kill, command** "COMMAND", **builtin** "Built-in Command" ( enable -n/-a command - enable or disable a builtin command), **autoload**
- External Command: **expr** – All purpose expression evaluator
  math : **y=‘expr $y + 1‘**
  **expr length $string**
  **expr match $str ’$RE’**
  **expr match $str ’\(\RE\)’**
  **expr substr $str $pos $len** #Returns substr
  **expr index $str $substr**
  **expr $str : ’$RE’** # no of match from start
  **expr $str : ’\(\RE\)’** #Substr match from start
  **’.*$RE’** – Matches from end of string
- External Command: **seq** – prints sequence of integer, seq [-s separator] LAST or FIRST LAST or FIRST INC LAST
- **basename**, **pathname**, **cut** -d delimiter -ffield1,field2

## Other external Command

- **find** path -exec COMMAND \; if COMMAND contains {} ( substitutes the full path name )
- **xargs** - filter for feeding arguments to commands ( -n - no of columns), -O is verbatim option - good for giving file name with special characters
- **grep** – RE files
- **bc** – Base calculator can be used for floating point calculation
- **sed** -e "expression" or -f "filename" files, -n suppress automatic printing of pattern space.
  - Expression are of form **[Range] [!]Command**. Range can be line no or Regular expression e.g 1 1,5 /RE/ or /RE/,/RE/ $ (last line), ! means to execute command on every line expect the match Commands can be **d** (delete), **p** (print), **=** (prints the line no), **a** **"text"** (append), **i** **"text"** (insert) , **c** **"text"** (change) , **n** ( go to next line), **w file** (write to file)
  - Substitution commands have the pattern **s/RE/rep-string/** – p - print, g - global replace. / can be replaced by any other char depending upon convenience e.g
    sed -e "s@/home/vinod@/vinod@g" file1
  - To give multiple commands use **";"** Grouping of commands use **{ }**
  - Pattern spaces is the current pattern and Hold Space is the buffer. There are commands to manipulate it. Check the man page for Usage

## Other Tools and Languages

- **awk** – Is a full text processing language and used in conjunction to shell scripts. e.g
  awk -F: 'BEGIN print "Check file"  print NR,$1' /etc/passwd – Begin executes before starting, -F: (FS - Field Separator), NR is number of record, $0 - all the fields, $[n] - Individual fields
- **DejaGNU Testing framework** : In Makefile.am add AUTOMAKE_OPTIONS = dejagnu AND RUNTESTDEFAULTFLAGS = –tool name CALC='pwd'/binexe –srcdir ./testsuite <– Testsuite directory. Create .exp file which is an expect script
  This tool is used widely in GNU framework for testing, examples include gcc, gdb etc.... Due to expect it can handle both interactive, batch operations and GUI testing
- **Autotest** - This is an integrated set that comes along with Autoconf framework based on m4 macros(AT_*).
  - Good for batch operations, cannot handle interactive applications
  - Creation of .at file which is set of m4 macros
  - This tool is slowly gaining popularity, and many of the applications are moving towards it
  - Important tool to be considered in coming years