Sample tester (bash)

- Let’s incrementally develop a simple bash script to automate testing of a program
- Simplifying assumptions:
  - program reads from standard input, writes to standard output, takes no command line arguments
  - program is normally supposed to exit with status 0, any other status represents and error
  - for each test case we have two files: one for the input to use, one for the expected output

Run a test case, check status/output

- Assume variables we use for filenames are
  - prog (program under test)
  - efile (expected output)
  - tfile (test case input data)
  - ofile (actual output)
  - dfile (differences between expected and actual output)
- Run test case, capture results

```
“${prog}” < “${tfile}” > “${ofile}”
status=$?
```

What about runaway output?

- suppose bug in prog produces huge output and we want to limit amount stored, e.g. using head -c someamount

```
“${prog}” < “${tfile}” | head -c ${climit} > “${ofile}”
```

- Problem: piping through head masks access to $?
- maybe run twice? one to capture $? (throw away output), one to capture output (through head)

```
“${prog}” < “${tfile}” > /dev/null
status=$?
“${prog}” < “${tfile}” | head -c “${climit}” > “${ofile}”
```

What about infinite loops/crashes?

- Suppose bug causes infinite loop, we want to terminate program after some time limit (cpu time), e.g. with ulimit
- set ulimit before each run and enclose in $( ) so command runs in subshell, so a program crash doesn't crash our script

```
$(ulimit -t "${cpu}"; "${prog}" < "${tfile}" > /dev/null)
status=$?
(ulimit -t "${cpu}"; ("${prog}" < "${tfile}" > | head -c "${climit}" > "${ofile}"))
```

Testing program exit status

- Should test status (non-zero means a problem), probably test after the first run?

```
$(ulimit -t "${cpu}"; "${prog}" < "${tfile}" > /dev/null)
status=$?
if [ "${status}" -ne 0 ] ; then
   echo "${tfile} failed: non-zero exit status"
fi
```

Testing output

- After second run, can use diff to compare actual output to expected, store differences in a file for later examination

```
(ulimit -t “${cpu}”; (“${prog}” < “${tfile}” > | head -c “${climit}” > “${ofile}”))
diff “${ofile}” “${efile}” &> “${dfile}”
```

- Diff actually returns 0 if files match, non-zero otherwise
- Trick: if command one succeeds run command two, otherwise run command three: `((cmd1 && (cmd 2)) || (cmd3))`

```
(((diff “${ofile}” “${efile}” &> “${dfile}”) && (echo "${tfile} passed")) || (echo "${tfile} failed") )
```

Now for multiple tests ...

- That was all for a single test case, usually we want to automate to run entire collections of test cases
- Assume one directory of test input files, another directory of expected output, and another directory for storing actual output
- Assume file names match in each directory, e.g. test case t1 has a file named “t1” in each of the three directories
- Assume we use variables to store the directory names

Iterating through test cases

- For each file in the directory of test case inputs, find the matching files in the other two directories

```
for tfile in “${indir}/”* ; do
   # extract just the name of the file
    fname=$(basename “${tfile}” )
    # generate names of other two files
    efile="${expdir}/${fname}"
    ofile="${resdir}/${fname}"
    .... code for single test case goes here ....
done
```

Temporary files

- Can use mktmp to generate a temporary file, one that is automatically deleted when script ends
- Perhaps use this to store the differences between the two files, e.g.
- dfile=$(mktemp)

The whole thing

```
#! /bin/bash
prog="./driverx"
indir="test/infiles"
expdir="test/expOut"
resdir="test/results"
cpu=10
climit=1000
for tfile in ${indir}/* ; do
    fname=$(basename ${tfile} )
    efile="${expdir}/${fname}"
    ofile="${resdir}/${fname}"
    dfile=$(mktemp)
    retv=0
    $(ulimit -t ${cpu}; ${prog} < ${tfile} &> /dev/null)
    retv=$?
    if [ $retv -ne 0 ] ; then
       echo "${tfile} failed: non-zero exit status"
    else
       (ulimit -t  ${cpu}; (${prog} < ${tfile} 2>&1 | head -c ${climit} > ${ofile}))
       (((diff ${ofile} ${efile} &> ${dfile}) && (echo "${tfile} passed")) || (echo "${tfile} failed") )
    fi
done
```

What about stderr?

- Suppose program produces stderr output as well
- Should have one test that captures them both together, making sure they are interleaved correctly (i.e. using 2>&1 then | head)
- Should have one test that separates them so we can ensure right content goes into each
- Need three expected output files: one for just stdout, one for just stderr, one for the combined