Testing

- We talked about testing earlier in the course
- Reminder:
  - Testing is about determining what is broken
  - Testing is never complete
- Approaches to testing
  - Black box vs. glass box

Warning: These notes are not complete, it is a Skelton that will be modified/add-to in the class. If you want to us them for studying, either attend the class or get the completed notes from someone who did

CSE3201

Tetsing

These slides are based on slides by Prof. Wolfgang Stuerzlinger at York University

testing and Debugging

- How programmer ensures that a program "works"
  - First creates set of test cases according to spec
  - Then creates implementation
  - Then tests implementation
  - Debugging if not all tests pass
  - Repeat until compliance with specification
- Note *all* of the above is the job of the programmer
  - Not customer, nor spec author, nor compliance tester
  - Includes creation of adequate test cases

Testing

- We need to pick our tests and perform them
- A good test should be:
  - Specific
  - Repeatable
  - Deterministic

Using Shell Scripts

- Shell scripts can help with the “repeatable” and
- “deterministic” parts
- We want to encapsulate the test in a script
- Running the script will perform the test for us and
- tell us the result
- Running the test again later should be as easy as running the script

Testing

- So let’s put together a script for the test we’ve come up with
- Assume that the program we are testing is called

“**convert**”

- The command

convert

- executes the program with stdin being the keyboard and stdout being the screen

Using Redirection

- We could perform the test ourselves by typing in the test but we want to make it easier
- Put the input in a file (called test1.input)
- Use redirection to use that file as input:
- **convert <test1.input**

Using Redirection

- This will run the program and give it the input and then print the output on stdout
- We could check the output by hand
  - This is easy in this particular case
- However in general, it is much better to let the computer checks it.

Using redirection

- We create another file containing the expected output (call it test1.output)

test1.input

```
0
12
123
4
5
```

test1.output

```
0
12
123
4
5
```

Comparing Output

- First of all we need to capture the output
- **convert <test1.input >tmpfile**
- This will put the output of the program in a file called “tmpfile”
- We can now compare the two files “tmpfile” and
- “test1.output” to see if the program passed or failed the test

Comparing Output

- **diff tmpfile test1.output**
- If they are the same, diff outputs nothing and exits with a status of 0
- If they do not match, you see something like:
- **5c5**
- **< 000**
- **---**
- **> 0**
- and exit status is non-zero

Putting it Together

- Here's our first testing script:

```
#!/bin/sh
f=tmpfile
convert <test1.input >$f
diff $f test1.output
```

- If the program passes the test we see nothing
- If the program fails the test we see an error

Avoid Silent Success

- We have a problem here though:

Silent success

- If the test is passed we see nothing
- In the event of some other failure (e.g. a bug in your test script) we may have a "false success"

Avoid Silent Success

- We should be explicit about success or failure
- Use the exit status of ‘diff’:

```
if diff $f test1.output; then
    echo test1 passed
else
    echo test1 failed
fi
```

Scripting our Script

- We've assumed that our output will be viewed by a human
- What if this is one of many tests that are being
- executed by another script?
- Should be able to signal success or failure to another program!
- Use exit status of our script

Scripting our Script

```
#!/bin/sh
f=tmpfile
convert <test1.input >$f
if diff $f test1.output; then
        echo test1 passed
        exit 0 # "true"
else
        echo test1 failed
        exit 1 # "false"
fi
```

Finally

- We can now distinguish between success and failure
- Output of our script is either
- **test passed**
- Or a number of lines (from diff) followed by :
- **test failed**
- The output of diff helps for diagnosis of failures

Regression testing

- Recall: keep *all* of tests we come up with
- It's also a good idea to make a test for each bug you encounter while debugging
- Keep all these tests together
- After you change the program/fix a bug - run all tests again

Regression testing

- We can do this by using one script to run others:

```
#!/bin/sh
fail=0
for x in test1 test2 test3
do
        $x || fail=`expr $fail + 1`
done
echo "$fail tests failed"
```

Temp Files

- Script has another limitation
  - Output goes to "**tmpfile**"
- Don't need to keep this file around after script terminates
- Bigger problem: can't use the same name in other scripts (may overwrite file!)

Temporary Files

- Option 1: Current working directory and delete file after you are done
- Option 2: Under Unix, the path **/tmp** is reserved
- for temporary files
  - Anybody can write there
  - As the name implies these files are temporary
  - On most systems, there is no guarantee the file
- will be kept around
  - Almost always automatically deleted after 1 week

Temporary Files

- **#!/bin/sh**
- **convert <test1.input >/tmp/tmpfile**
- …
- A little better, but we are not sure that nobody else is using the name "**/tmp/tmpfile**"
- A unique name would be better

Unique Temporary Files

- A common approach is to use the process id of the shell (the special variable **$$**)
- This is an integer which is unique in the system We typically pick a name (test1) and append the
- process id to make it unique:
- **/tmp/test1.$$**
- **or use the method we discussed before**

Another Way

- An approach which checks for conflicts:
- **gettemp() {**
- **id=0**
- **while [-f $1.$$.$id]; do**
- **id=`expr $id + 1`**
- **done**
- **echo $1.$$.$id**
- **}**

sed

- Sed: Stream editor is an editor to modify files.
- If you want to write a program to modify files, sed is the solution
- Here is a brief introduction to sed, practice is the best help.

sed

- sed s/day/night <old >new
- Substitute the word day in the file old by the word new and store the results in a file called new
- preferably sed ‘s/day/night/’
- If the string contains “/” then you have to escape it or use another delim.
  - sed ‘s/\/usr\/local\/bin/\/common\/bin/’ <old >new
  - sed ‘s_/usr/local/bin_/common/bin_’ <old >new

sed – Using &

- The special character & corresponds to the search pattern.
- For example to sed ‘s/[0-9]*/& &/’ doubles a number at the beginning of a line
- “123 cat” → “123 123 cat”

- If you have many commands and they won't fit neatly on one line, you can break up the line using a backslash:
  - sed -e 's/a/A/g' \
    -e 's/e/E/g' \
    -e 's/i/I/g' \
    -e 's/o/O/g' \
    -e 's/u/U/g' <old >new

- If you have a large number of *sed* commands, you can put them into a file and use
  - sed -f sedscript <old >new
- where *sedscript* could look like this:
  - # sed comment - This script changes lower case vowels to upper case
    s/a/A/g
    s/e/E/g
    s/i/I/g
    s/o/O/g
    s/u/U/g