CS 246 Spring 2019 - Tutorial 2

May 22nd, 2019

1 Summary

- Exit Codes
- Bash Variables and Scripting
- Bash Loops and If Statements
- Testing

2 Program Exit Codes

- When a program completes, it always returns a status code to signify if the program ran successfully.
- This is true of any C program you have written before now. The exit code is the value returned from main, hence the contract `int main();`. In C and C++, if you do not return from main, the exit code is 0.
- In bash, if a program is successful, the exit code is 0. Otherwise, the exit code is non-zero. The exit code is stored in the variable `?`.
- The exit code cannot be larger than 255. In bash if you return some return code larger than 255, you will get the code modulo 256.

3 Bash Variables

- In bash, a variable is declared as follows: `var=42`.
  Note: There cannot be spaces on either side of the equals symbol.
- All variables are stored as strings.
- Unlike C variables, bash variables persist outside of the scope of if statements, loops, and scripts.
- Accessing the value in a variable: `$var`
- `${var%<end>}` removes the suffix `<end>` from the string stored in var. If `<end>` is not at the end of var, the string is unchanged.
- We can store a command in a variable and call it later.

4 Bash Scripting

- A bash script is a series of commands saved in a file so that we can accomplish the same task without having to manually type all the commands
- The first line of every shell script is the "shebang line" - `#!/bin/bash`. This line is telling the shell what program the file should be invoked with. This line is optional, but is a good idea.
- To call a bash script, give the file excutable permission using `chmod` and call the file by giving either an absolute path or a relative path to it. You can also invoke a bash script without making it executable by calling `bash <script_name>`
- Command line arguments are $1, $2, etc. The number of command line arguments is stored in $#.

4.1 Subroutines in Bash Scripts

- Format:

```
subroutine(){
    ...
}
```

- A subroutine is a series of commands which can be called at any time in a bash script.
- They can be given command line arguments similarly to how a program would be given command line arguments. A subroutine cannot access the command line arguments to the script. All other variables can be accessed. Note that the parameters of a bash subroutine are **not** listed with the ().
- **Exercise:** Write a bash script which takes in two arguments, `ext1` and `ext2`. For each file (not directory) in the current directory which ends with an `.ext1`, rename the file to end with `.ext2`.

5 Bash Loops and If Statements

- For the condition in both if statements and while loops, the result is checked, and if it's 'true', the program will go into the body of the if statement or while loop.
  - Form of an if statement in bash:

```
if [ cond1 ]; then
   ...
elif [ cond2 ]; then
   ...

.........
else
   ...
fi
```

  - Form of a while loop in bash:

```
while [ cond ]; do
   ...
done
```

  - Form of a for loop in bash:

```
for var in words; do
   ...
done
```

- "`words`" is a list of whitespace separated strings. The loop runs once for each string in "`words`".
- You can use the `seq` command to generate a space-separated list of numbers.

```
for var in $(seq 1 10); do
    echo $var
done
```

```
#Alternate syntax
for var in {1..10}; do
    echo $var
done
```

- Side note: '`[ cond ]`' can be replaced by any command and the exit code will be checked.

5.1 Test Command

- Test is a bash command. The program is `[` and is called in the form `[ cond ]` whose exit code is 0 if `cond` is true and 1 if `cond` is false. It may be useful to review the `man` page for test.
- A short non-exhaustive list of conditions for test:
  - `num1 -gt num2`: num1 > num2
  - `num1 -lt num2`: num1 < num2
  - `num1 -eq num2`: num1 == num2
  - `num1 -ne num2`: num1 ! = num2
  - `str1 = str2`: str1 == str2 (string equality)
  - `-e file`: file exists
- Note that bash does not use Korn shell syntax ("&&" and "||"), but rather uses -a for "and" and -o for "or".

6 Testing

Testing your program is a very important aspect of programming. You want to ensure that you are satisfied that your program is correct. Most, if not all, of future assignments will ask you to create a test suite which will act as your primary source for verifying the correctness of your code.
(Marmoset is **not** designed for this purpose)

When designing your test cases for your test suite, keep the following things in mind:

- Your test suite should pass any correct implementation of a program, but it should also reject/fail any incorrect or buggy implementation as well.
- There is a balance between quantity and quality for test cases. There should be enough tests in your test suite to ensure that your test suite is robust (see above point), but at the same time each test should test for something meaningful— there should not be 'redundant' tests.
- Both **whitebox testing** (Testing knowing the specific implementation) and **blackbox testing** (Testing without knowing how the program is implemented) should be performed as deemed necessary.

6.1 Types of Test Cases

You should also consider a wide coverage of different test case types:

- **General/Basic cases:** These can act as "sanity checks" to check for very basic mistakes (e.g. formatting mistakes).

- **Equivalence Classes:** Different sizes of inputs (small/large inputs), different 'amounts' of inputs (few/a lot of inputs), or even the empty input if necessary.
  - For example, tax brackets.[1]
- **Boundary/Edge cases**
- **Corner and Weird Cases**

But remember, no number of tests can **guarantee** the correctness of a program.

6.2 Testing Exercises

Exercise: We have a compiled program `test` which is invoked as follows:

```
./test <testnum>
```

Where `<testnum>` is an integer between 1 and 8 denoting the number for the test you want to run. The program then expects from standard input lines of integers separated by whitespace. The program takes a line of integers and tests if they satisfy a certain property (which will depend on `<testnum>`). The program will return 'true' for each line of numbers which satifies the property and 'false' for all other input. For example, for test number 1, the input:

```
2 4 6
```

returns '`true`'.

In `check.h`, for each test number there is a "Claim" claiming what that test number's property is. Some of these claims are correct, and others are incorrect. Come up with some test cases (i.e. inputs) and determine which claims are correct and which are incorrect.

Exercise: Consider how you would test the bash script we wrote in the last tutorial (changing extensions). What different test cases should you consider?

[1] `https://www.canada.ca/en/financial-consumer-agency/services/financial-toolkit/taxes/taxes-2/5.html`

7 Tip of the Week: Bash Script Debugging

- A debugging mode can be activated when running a bash script by placing '-x' at the end of the shebang line, or calling it using `bash -x`.
- When running the script, each command, with variables being expanded, is printed to the screen.
- If a script is not doing what you expect it to do, using this debugging mode can be an easy way to see what is happening in the script.

8 Vi Tip of the Week: More Vi Commands

- `dd`: Delete the current line.
- `C`: Delete contents from current character to the end of the line and enters insert mode.
- `D`: Deletes contents from current character to the end of the line.
- `cw`: Deletes all characters from the current character to the next whitespace and enters insert mode.
- `dw`: Deletes all characters from the current character to the next whitespace.
- `r`: Replace the current character with the next character typed.
- `R`: Enter replace mode. This is like insert mode except characters will be replaced when you type.
- `s`: Delete the current character and enter insert mode.
- `S`: Deletes contents of the current line and enters insert mode.