CS 246 Spring 2018 — Tutorial 2

May 16, 2018

Summary

1 Bash Variables

- In bash, a variable is assigned a value as follows: `var=42`. You do not need to declare a variable before assigning a value.

  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: `$var` or `${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.
- In addition to using variables as arguments, we can also treat the value of a variable as a command and run it:

```
greet="echo hello"
$greet
```

2 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.
- To call a bash script, give the file executable 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>`.

  Note: if the relative path consists of only the file name (e.g. `script_name`), we need to add `./` before the path to call it: `./script_name`.
- Command line arguments are `$1`, `$2`, etc. The number of command line arguments is stored in `$#`.

2.1 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 the same way 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.
- **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`.

2.2 Debugging

- 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 is printed to the screen with variables expanded.
- 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.

3 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
```

  Where `<words>` is a list of whitespace separated strings. The body of the loop runs once for each string in `<words>`.

- You can use the `seq` command to generate a list of numbers.

```
# prints 1 to 10
for num in $(seq 1 10); do
    echo $num
done
```

```
# alternatively
for num in {1..10}; do
    echo $num
done
```

- Note: `[ <cond> ]` can be replaced by any command and the exit code will be checked. For example:

```
# prints "done" if cat succeeded
if cat file.txt; then
    echo done
fi
```

3.1 Test Command

- `test` is a bash command. The program is implicitly referred to using `[` (though it can also be explicitly referred to using `test`) 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` (`man [` brings up the same page).
- A few conditions you can use for test:

`num1 -gt num2` $\quad$ `num1` $>$ `num2`

`str1 = str2` $\quad$ `str1` $==$ `str2` (string equality)

4 Program Exit Codes

- When a program completes, it always returns a status code to signify if the program was a success.
- 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 explicitly 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 `$?`.

  Remember: this is opposite from the definition of true in C. In C a non-zero integer represents true, while in bash zero represents success.
- 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.

5 Make and Makefiles

- With single-file programs, compilation is a breeze:

  ```
  g++14 change.cc -o change
  ```
- However, when we have a project across multiple files, compilation may become a pain to type out. Surely there is a better way to compile a project without typing all `.cc` files.
- You should use separate compilation which looks something like

  ```
  g++14 -c main.cc
  g++14 -c book.cc
  ...
  g++14 book.o main.o textbook.o ... -o main
  ```
- When we do this, we only have to recompile the modules that change. This means less time compiling but more time remembering what we have recently compiled.
- Surely there must be a better way to keep track of changes. This is a bigger issue when we would constantly be recompiling everything when we don't have to.
- `make` can help here. It allows us to specify the dependencies of targets (the files produced by the build process) and the command for producing each target in a `Makefile`. `make` will automate the building process and avoid unnessesary compilation by keeping track of changed files based on last modified time [1].

[1] i.e. if a target is newer than its dependencies, then there is no need to rebuild this target

A `Makefile` will look something like:

```
# example1/Makefile

# means main depends on these dependencies
# if all of the dependencies are up to date, then execute
# the build command below
main: main.o book.o textbook.o comic.o
    # specifies how to build main
    g++ -std=c++14 main.o book.o textbook.o comic.o -o main

book.o: book.cc book.h
    g++ -std=c++14 -c book.cc
textbook.o: textbook.cc textbook.h book.h
    g++ -std=c++14 -c textbook.cc
comic.o: comic.cc comic.h book.h
    g++ -std=c++14 -c comic.cc
main.o: book.h textbook.h comic.h main.cc
    g++ -std=c++14 -c main.cc
```

- The whitespaces before the build command (in this case, `g++ ...`) **MUST** be a tab.
- On the command line, run `make`. This will build our project.
- If `book.cc` changes, what happens?
  - compile `book.cc`
  - relink `main`
- What happens when we execute the command `make`?
  - Builds first target in our Makefile, in this case `main`.
  - What does `main` depend on?
    - `book.o textbook.o comic.o main.o`
  - If `book.cc` changes:
    - `book.cc` is newer (timestamp) than `book.o`; rebuilds `book.o`
    - `book.o` is newer (timestamp) than `main`; rebuilds `main`
- **Tip**: We can build specific targets using `make`:

  ```
  make textbook.o
  ```

- Common practice: put a `clean` target at the end of a makefile to remove all binaries[2]

  ```
  clean:
      rm *.o main
  ```

[2]The description found in `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`

```
# clean is a "phony target": it is not name of a file but
# a recipe to be executed when an explicit request is made
.PHONY: clean
```

- To do a full rebuild:

```
make clean && make
```

- While Makefile can make our compilation process easier, writing out all of the dependencies and individual compilation commands can be time consuming.
- Conveniently, we can generalize a Makefile with variables.

```
# example2/Makefile
CXX=g++                                                    #compiler name
CXXFLAGS=-std=c++14 -Wextra -Wpedantic -Wall -Werror -g  #options to pass
...
book.o: book.cc book.h
    ${CXX} ${CXXFLAGS} -c book.cc
...
```

- **Shortcut:** For any rule of the form `x.o: x.cc a.h b.h`, we can leave out the build command. `make` will guess that it is `${CXX} ${CXXFLAGS} -c book.cc -o book.o`

```
# example3/Makefile
CXX=g++                                                    #compiler name
CXXFLAGS=-std=c++14 -Wextra -Wpedantic -Wall -Werror -g  #options to pass

# This one is not in the generic form, so have to list it
${EXEC}: ${OBJECTS}
      ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

...
book.o: book.h book.cc

textbook.o: textbook.h textbook.cc book.h
...
```

- Issue: how to track dependencies and updating them as they change
  - `g++` can help. `g++ -MMD -c comic.cc` will create `comic.o comic.d`.
  - What will `comic.d` contain?

```
comic.o: comic.cc book.h comic.h
```

- Looking at this `.d` file, we can see it is exactly what we need in our `Makefile`. We just need to include all `.d` files in our `Makefile`. This means our `Makefile` will look like

```
# example4/Makefile
```

```
CXX=g++
CXXFLAGS=-std=c++14 -Wextra -Wpedantic -Wall -Werror -MMD -g
OBJECTS=main.o book.o textbook.o comic.o
DEPENDS=${OBJECTS:.o=.d}
EXEC=main

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

clean:
    rm ${OBJECTS} ${DEPENDS} ${EXEC}
.PHONY: clean
```

- Testing: Assuming there is a testing folder in the directory the makefile is in, we can add a `test` target similiar to `clean`

```
# example5/Makefile
CXX=g++
CXXFLAGS=-std=c++14 -Wextra -Wpedantic -Wall -Werror -MMD -g
OBJECTS=main.o book.o textbook.o comic.o
DEPENDS=${OBJECTS:.o=.d}
EXEC=main

# feel free to change these directories to your liking
# this setup assumes that in the directory with the Makefile, there is
# 1) A directory called tests where all the tests are
#    a) Inside that directory, there is a suite.txt to be used
#    b) There is also the scripts produceOutputs and runSuite
# 2) A directory called executables one level above this one
#    that contains the given executable with the same name
TESTDIR=test
EXECSDIR=../executables/
EXECGIVEN=${EXEC}
SUITE=suite.txt

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

clean:
    rm ${OBJECTS} ${DEPENDS} ${EXEC}
.PHONY: clean
```

```
# @ silences output
test:
    @cp ${EXECSDIR}/${EXECGIVEN} ${TESTDIR}                  # copy over given exec
    @mv ${TESTDIR}/${EXECGIVEN} ${TESTDIR}/sol_${EXECGIVEN} # prefix it with sol_
    @cp ${EXEC} ${TESTDIR}                                   # copy over your exec
    @(cd ${TESTDIR}\                                         # go into test dir
    && ./produceOutputs ${SUITE} ./sol_${EXECGIVEN}\         # run produceOutputs
    && ./runSuite ${SUITE} ./${EXEC})                        # run runSuite
.PHONY: test
```

- To test:

  `make test`

- This is the final version of our `Makefile`. Altering the variables of this `Makefile`, we can use this exact `Makefile` for basically any program we want to create.

6 Vim Tips of the Week: More Normal Mode Commands

- In vim nearly all the most useful and commonly-used commands are in normal mode. Try to get into the habit of pressing `Esc` to go back to normal mode whenever you finish inserting text.
- Some more normal modes command useful for editing code:

  `yy` copies the current line.

  `dd` copies and deletes the current line.

  `cc` copies and deletes the current line, then enter insert mode.

  `cw` deletes all characters from the current character to the end of the word and enters insert mode.

  `dw` deletes all characters from the current character to the end of the word.

  `p` pastes the copied content. If the content is full lines then it will be placed after the current line. Otherwise it will be after the current character.

  `*` searches the current word under cursor. Similar to pressing the `/` key followed by the current word and pressing `Enter`, but only finds words that match as a whole (i.e. not substrings).

  `.` repeats the last change.

  `C-r` (Ctrl-R) redo (undo the last undo operation by the `u` key).