CS 246 Fall 2013 - Tutorial 2

September 24, 2013

1 Summary

- Shell Scripting

- Testing

- C++ I/O

2 Shell Scripting

2.1 Basic Scripting

- The following is a simple shell script that you may have seen before

```
#!/bin/bash
date
whoami
echo "Hello World!"
```

- Suppose that this script is contained in the file "simple.script". How do we run this script?

- Answer 1: Use chmod to set the user executable bit

- Answer 2: Invoke bash with the script as a parameter

2.2 Command Line Arguments

- To make more complex shell scripts, we likely want to accept arguments from the command line

- Recall the following special shell variables:

  - `${#}` provides the number of arguments, excluding $0
  - `${0}` **always** contains the name of the script
  - `${1}`, `${2}`, `${3}`,... refer to the arguments by position (not name)
  - `${@}` all arguments supplied to the currently running script (except $0), as separate strings

- Let's very quickly see these in action - see paramExample.script

2.3 More Complex Scripting

- Recall from lecture that bash shell scripts can have if-statements, routines, for-loops, while loops, variables, and possibly other things

- Let's write a shell script that determines the factorial of a given argument (> 0).

- Let's start by writing the main body of the program

```
#!/bin/bash
i=$((${1}-1))
total=$1
while [ "${i}" -gt 1 ]; do
    total=$((${total} * ${i}))
```

```bash
6       i=$((${i}-1))
7    done
8    echo "${1} factorial is ${total}"
```

- But what if `${1}` isn't > 0? We should check this.

```bash
1    #!/bin/bash
2
3    if [ "${1}" -lt 1 ] # line *
4    then
5      echo "${1} is not > 0" 1>&2
6      exit 1
7    fi
8    # Insert body of function here
```

- Why do we encapsulate `${1}` in quotes on line *?

- What else should we check?

  – That we have a single parameter? Yes.

  – That the single parameter is a number? Yes. But that's more complicated so we won't

```bash
1    #!/bin/bash
2    if [ ${#} -ne 1 ]
3    then
4      echo "Incorrect number of parameters" 1>&2
5      echo "Usage: ${0} n, where n > 0"
6      exit 1
7    elif [ "${1}" -lt 1 ]
8    then
9      echo "${1} is not > 0" 1>&2
10     echo "Usage: ${0} n, where n > 0"
11     exit 1
12   fi
13   # Include body here
```

- But now we're duplicating lines of code. How to solve? Create a usage routine and call that!

```bash
1    #!/bin/bash
2    usage(){
3      echo "Usage: ${0} n, where n > 0" 1>&2
4      exit 1
5    }
6    if [ ${#} -ne 1 ]
7    then
8      echo "Incorrect number of parameters" 1>&2
9      usage
10   elif [ "${1}" -lt 1 ]
11   then
12     echo "${1} is not > 0" 1>&2
13     usage
14   fi
15   # Include body here
```

- Note that we could make this more robust and let our usage routine take a parameter but that's for you to figure out.

3   Testing

We're going to perform a miniature case study of testing and determine some possible test cases for a problem.

Problem: Given a program that reads from stdin a list of integers with the goal of determining if some combination of a list of integers can sum to the last integer given, where integers are $>= 0$, e.g. input is of the form

```
n
x_1
x_2
..
x_n
y
```

where n specifies the number of possible summands, `x_i` is a possible summand in ascending order, and y is the target value. If no combination of integers can sum to the target value then "Impossible!" should be printed.

What are some possible test cases?

- Test containing 0 as target and no 0 summand should print "Impossible!"

- Test containing 0 as both, should print "Impossible!"

- Test which fails if you start from low and go to high

- Test which contains even integers and an odd target should print "Impossible!"

- Target smaller than all integers should print "Impossible!"

- Target less than largest integer but target still attainable

4   C++ I/O

- C++ I/O is radically different than the C I/O you may be used to

- For this course, you should not use C I/O unless told otherwise

- Recall, that C++ has three default input and output streams:

    - **cout** - standard output
    - **cerr** - standard error (unbuffered - prints immediately)[1]
    - **cin** - standard input

- Let's see an example that will take in a number and output a phrase. (phrases.cpp)

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
  int choice;
  int numChoices = 5;
  string phrases[] = {"More Vespene Gas required.","The sun is shining. But the ice is slippery.",
                      "Gotta go fast!", "Autobots, roll out!", "Do or do not. There is no try."};
  cout << "Please choose a number from 1-5: ";
  while(cin >> choice) { // cin needs to be read at least once before it can hit eof or fail
    if(choice > numChoices)
      cerr << "Invalid number" << endl;
    else
      cout << phrases[choice-1] << endl;
    cout << "Please choose a number from 1-5: ";
  }
}
```

- This program will end when either eof is reached or invalid input is given (e.g. non-integer input).

- Accordingly, this program is not very robust. How could we make it more so?

    - Explicitly checking for failure/eof by using `cin.fail()` and `cin.eof()` and not using the implicit conversion to boolean value

---

[1]Technically, there are four. The fourth is clog and is basically bufferred cerr

- If we are in a failure case then we could use `cin.ignore()` to ignore the next character of input and then `cin.clear()` to reset the failure flag
- Why do we reset the failure flag?

- Recall that cin ignores any and all whitespace (unless you use an I/O manipulator to tell it otherwise).

- Suppose we wanted to get an entire line. How could we do this?

  - By using **getline**, e.g. `string s; getline(cin, s)`
  - Thus we take a line from cin and store it in the string s.
  - But how do we process this line now? Using **stringstreams**! But that's next week.