

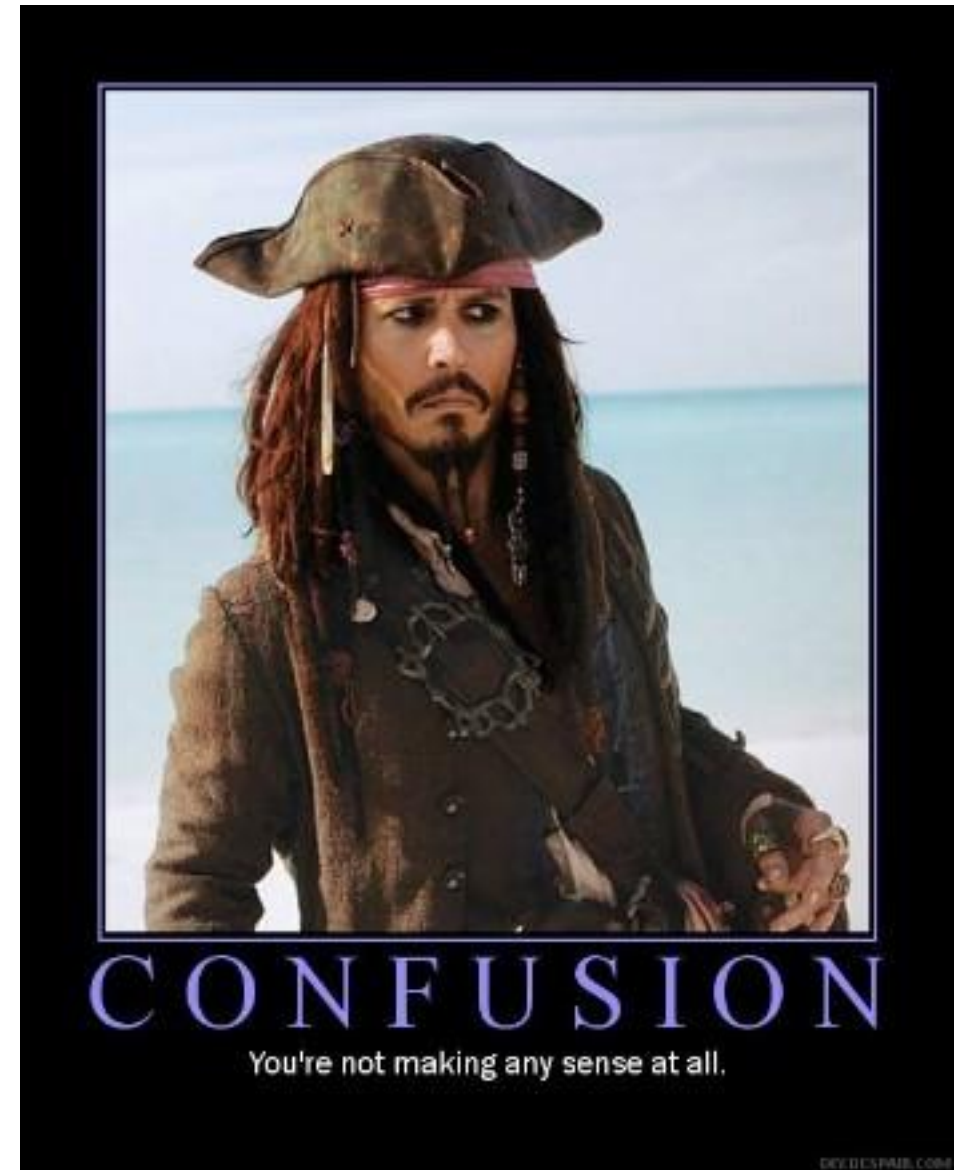
Solving Problems & Debugging

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,
with attributions given whenever available

The Challenge

- You have been asked to write a piece of software
 - You have been given specifications
- You are not sure where to start
 - Perhaps you don't understand the underlying technology
 - Perhaps you don't understand all the specifications
 - Perhaps you are not sure how the different components work together



Advice to Follow

- The following advice applies to every single software project that you will encounter in your life!
- And certainly much more than software.

[Warning: philosophy]



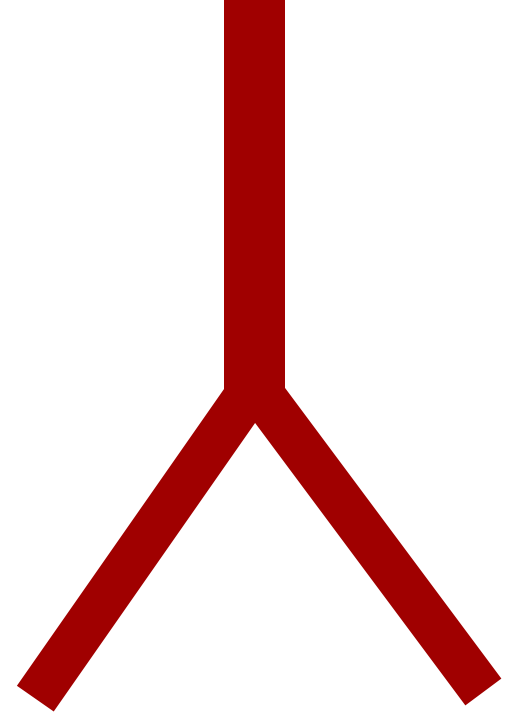
Making a Large Problem Seem Easy

- Divide and conquer!
- Break the problem into smaller and smaller pieces - ask yourself about each piece:
 - Do I already know how to do this?
 - If not, does this seem easy?



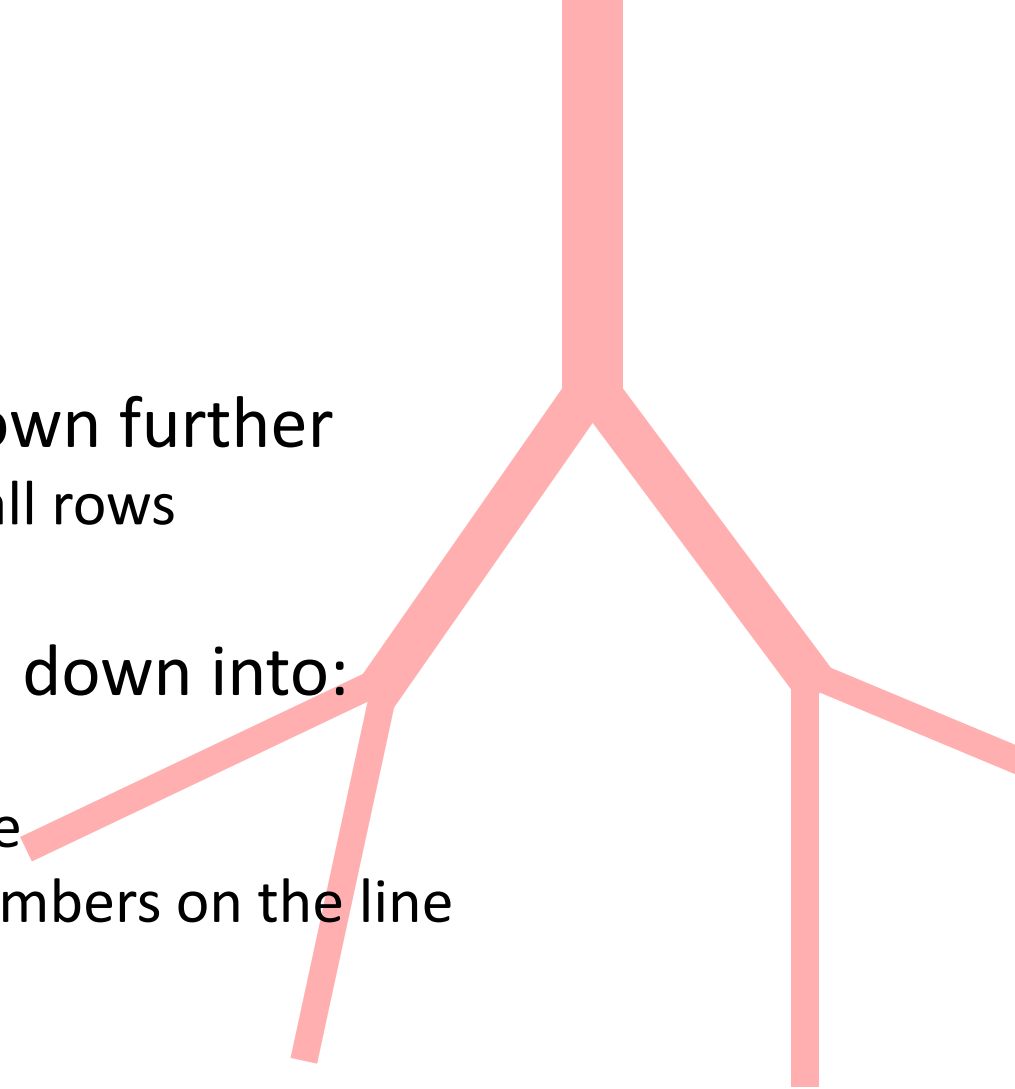
Example: Program 1

- Identify the core task
 - Compute statistics
 - Ignore the rest (handling signals, etc.)
- Break the core task into pieces
 - Need to compute statistics on rows AND on columns
- Focus on one of the sub-pieces
 - How do we compute stats *just* for rows?



Example: Program 1

- Still not sure what to do, so let's break it down further
 - Try and compute stats for one *single* row, not all rows
- Computing stats for one row can be broken down into:
 1. We need to read one line from a file
 2. Then we need to sum the numbers in the line
 3. Then we need to divide by the number of numbers on the line
 4. Then we need to print out the result
- Each of *these* look doable!



Searching with a Mission - Step 1

- **How do we read one line from a file?**
- Let's read through all of the sources available to you, this time looking for a specific solution to reading a single line
- Re-read technical reference documentation once you have a compact problem
 - You'll find that you'll pay closer attention
 - You'll remember more of what you read
 - You'll find the answers to your problem
 - Before, you didn't have a goal, so your brain didn't bother to internalize or remember the information



Common UNIX Commands – Reading Data

- **read** – Get data from either stdin or a file (with -u option)

```
$ cat readtest
#!/bin/bash
echo "Enter in a string, then hit enter:"
read myvar
echo "You entered: $myvar"
echo $myvar > "tempfilename$$"
echo $myvar >> "tempfilename$$"
echo $myvar >> "tempfilename$$"
echo "The temp file contents are:"
cat "tempfilename$$"
echo "The first line of the file is:"
read entireline < "tempfilename$$"
echo "$entireline"
read firstword restofline < "tempfilename$$"
echo "First word of first line: \"$firstword\""
echo "Rest of line: \"$restofline\""
rm -f "tempfilename$$"
```

```
$ readtest
Enter in a string, then hit enter:
THIS SENTENCE IS FALSE
You entered: THIS SENTENCE IS FALSE
The temp file contents are:
THIS SENTENCE IS FALSE
THIS SENTENCE IS FALSE
THIS SENTENCE IS FALSE
The first line of the file is:
THIS SENTENCE IS FALSE
First word of first line: "THIS"
Rest of line: "SENTENCE IS FALSE"
```


Learning to Read - Step 1

- You figured out how to read a single line from stdin into a variable:
 - `read singlelinevar`
- Test your knowledge
 - Write a tiny shell script to test it
 - You also figure out how to print the line to the screen to test it

```
$ cat readtest
```

```
#!/bin/bash
```

```
read singlelinevar
```

```
echo $singlelinevar
```

```
$ cat data
```

```
6 4 4 7 7
```

```
$ cat data | readtest
```

```
6 4 4 7 7
```

Summing up - Step 2

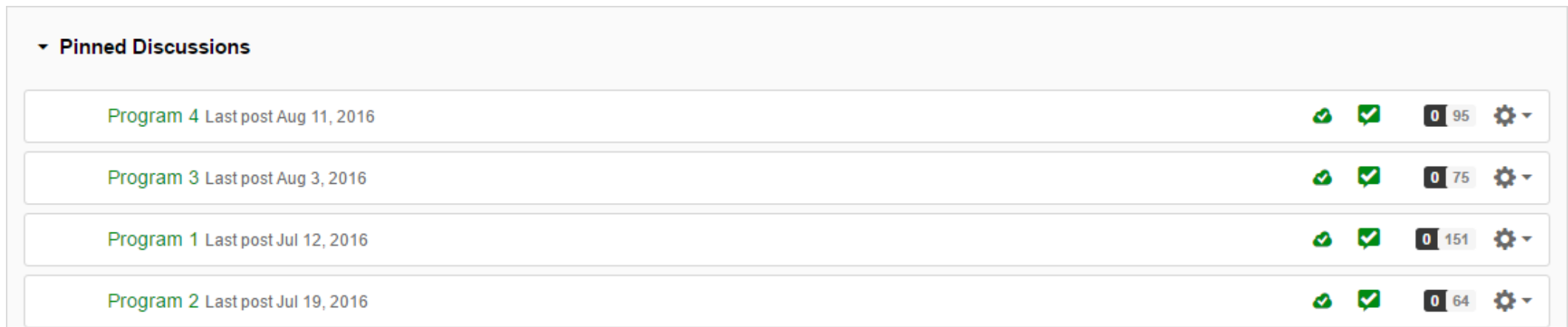
- Next step: **sum the numbers on a line**
- Decompose further:
 - Forget the line, how do I add two numbers?
 - Then worry about getting those numbers from the line
- You read the docs (finding it faster this time!), and figure out how to add two numbers and store them in a variable

```
sum=`expr 3 + 4`
```

```
sum=`expr $var1 + $var2`
```

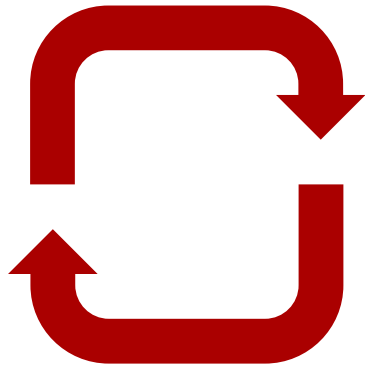
Lotsa Summing - Step 2

- Now, how to take those numbers from the line you read?
- And - a new challenge: what if you don't know how many numbers are on a line?
 - `read` X reads the entire line into a variable - how do you break it up?
- Raw documentation might not be as useful as putting the question up on a discussion board, asking a friend, or searching online



Loop it - Step 2

- Let's loop it - we've learned about the bash **for** loop:



```
$ cat data
```

```
6 4 4 7 7
```

```
$ cat readlooptest
```

```
#!/bin/bash
```

```
read myLine
```

```
sum=0
```

```
for i in $myLine
```

```
do
```

```
    sum=`expr $sum + $i`
```

```
done
```

```
echo "sum is: $sum"
```

```
$ cat data | readlooptest
```

```
-bash: ./readlooptest: Permission denied
```

```
$ chmod +x readlooptest
```

```
$ cat data | readlooptest
```

```
sum is: 28
```

Dividing and Printing - Step 3 & 4

- Next step: **divide by the number of numbers on the line**
- Decompose further:
 - How can we tell how many numbers are on a line?
 - How do we divide two numbers?
- We can simply add a count variable to our previous code, and use the division operator instead of addition
- Then use printf to combine it all!

```
$ cat data
6 4 4 7 7

$ cat countdividelooptest
#!/bin/bash
read myLine
sum=0
count=0
for i in $myLine
do
    sum=`expr $sum + $i`
    count=`expr $count + 1`
done
mean=`expr $sum / $count`
echo "$count entries sums to: $sum"
echo "mean average is: $mean"

$ cat data | countdividelooptest
5 entries sums to: 28
mean average is: 5
```

Huh. `expr` only works with integers!

Problem Solving Summary

- Break down the problem until you have a tiny problem that looks solvable
- Then use docs/friends/online searches to find the solution
 - You will learn much more if you are reading with a well-defined problem
- Solve the tiny problem
 - Make sure that you write the code!
 - Feel good about the success – have a donut:
- Now work upwards
 - Integrate your solution into the (slightly) larger problem
- Repeat!
- See the online Shell Script Compendium page for an example script (“bigloops”) that reads ALL lines from a file, sums them, and computes the averages



Introduction to Debugging



- Debugging is the art of figuring out why something has gone horribly, awfully wrong
- Testing is done to *find* bugs, debugging *removes* them

Debugging Process

1. Reproduce the problem reliably

- Simplify input and environment until the problem can be replicated at will
 - e.g. Wolf Fence algorithm (next slide)
- Challenges:
 - Unique environment (space station, aunt Edna's PC in Hoboken, NJ, etc.)
 - Particular sequence of events leading up to the error are unknown or difficult to do more than once (lightning strike, aunt Edna tries to watch Netflix through her toaster, etc.)

2. Examine the process state at the time of error; we'll cover 3 types:

1. Live Examination
2. Post-mortem Debugging
3. Trace Statement

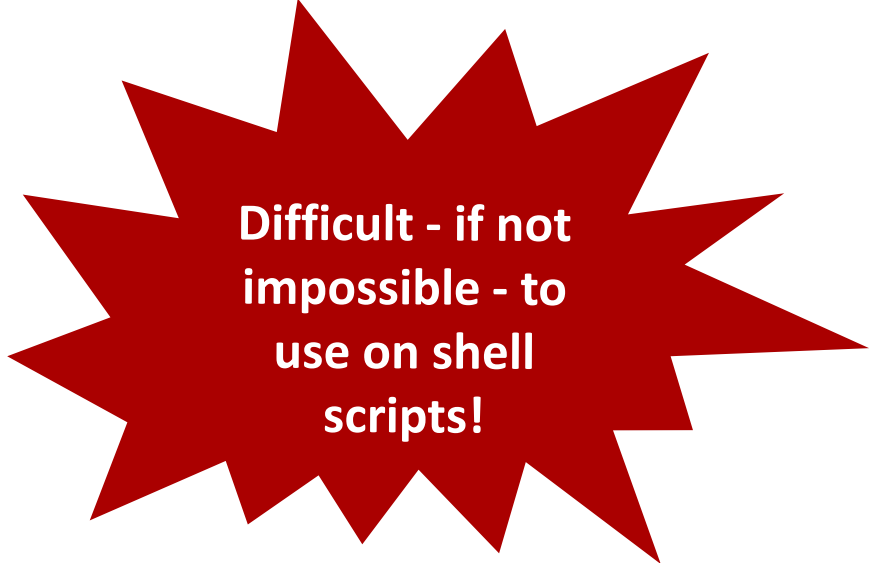
Wolf Fence Algorithm



- If you can't find the source of the problem, or can't reproduce it, narrow things down by dividing and conquering:
 - Bifurcate the code by commenting out half of it, and running it again
 - Change the code to ignore all of the input but one piece and run it again
- "There's one wolf in Alaska; how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf."
 - E. J. Gauss (1982). "Pracniques: The "Wolf Fence" Algorithm for Debugging", in Communications of the ACM

Technique: Live Examination

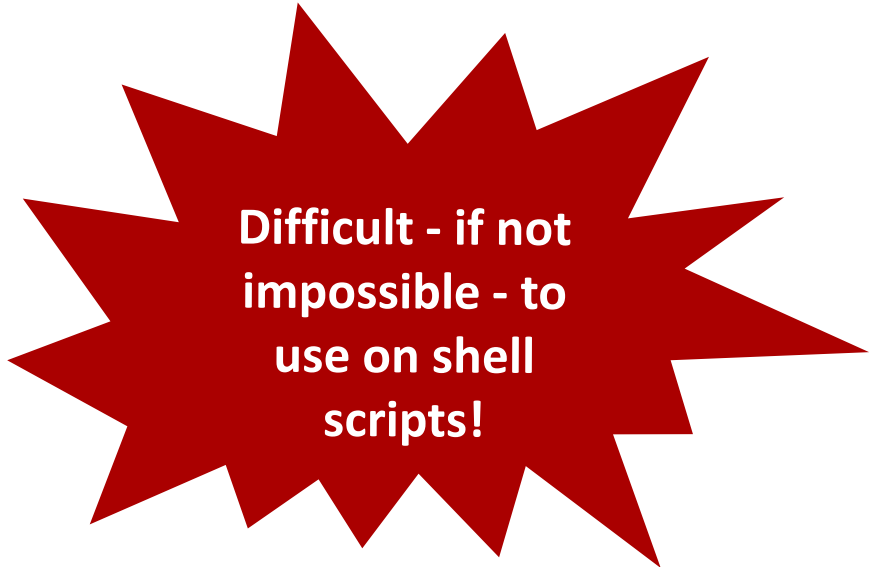
- A separate application allows you to pause process execution and view the contents of variables and stack trace directly
- Common tools:
 - IDEs: Microsoft Visual Studio, Eclipse, Monodevelop, etc.
 - Apps: gdb, valgrind
- Variable values can even be edited on the fly!
- Debugger applications and IDEs can be connected to processes on other machines (aka remote debugging)

A red starburst graphic with multiple points, containing white text.

**Difficult - if not
impossible - to
use on shell
scripts!**

Post-mortem Debugging

- After the program crashes, UNIX will leave behind a core dump file that can be examined
- The core file stores the contents of that program's virtual memory contents, including CPU registers, program counter, stack pointer, and other OS info
- gdb is used to analyze core files on UNIX:
`$ gdb myprogram -c coredumpfile`



**Difficult - if not
impossible - to
use on shell
scripts!**

Trace Statements



- Trace statements - *what* and *where*
 - Print out the value of your variables as you go (what)
 - Print out where you currently are (where)

- What and where:

About to Begin Loop: `j = 0`

Now At Start of Loop: `i = 0, j = 0`

Sum() function ran: `i = 1, j = 1844835813859`

Executed file read: `i = 1, j = ^^^^ ^^°ØĆ@«`

Contents of file read: `fj8283jJ*#Jf8j32@fj`

Now at End of Loop: `i = MDKMDKMDK, j = 42941492`

Start missile laun`^C ^C ^C ^C`

Garbage! Sum() is busted!

Hull breach!

The horror!

No survivors!

Halt and catch fire!

Debugging Bash Shell Scripts

- Enable printing of command traces before executing each command:

- Add -x to the end of the first line of the script:

```
#!/bin/bash -x
```

OR

- Launch the script with a shell that has command traces enabled:

```
$ bash -x sumloop
```



Debugging Bash Shell Scripts - Example

```
$ sumloop
```

```
In Loop  
num: 8  
sum: 8  
End of Loop
```

```
In Loop  
num: 7  
sum: 15  
End of Loop
```

```
In Loop  
num: 6  
sum: 21  
End of Loop
```



```
$ bash -x sumloop
```

```
+ sum=0  
+ TMP1=./TMP1  
+ echo -e '8\n7\n6'  
+ read num  
+ echo 'In Loop'  
In Loop  
+ echo 'num: 8'  
num: 8  
++ expr 0 + 8  
+ sum=8  
+ echo 'sum: 8'  
sum: 8  
+ echo -e 'End of Loop\n'  
End of Loop
```

```
+ read num  
+ echo 'In Loop'  
In Loop  
+ echo 'num: 7'  
num: 7  
++ expr 8 + 7  
+ sum=15  
+ echo 'sum: 15'  
sum: 15  
+ echo -e 'End of Loop\n'  
End of Loop
```

```
+ read num  
+ echo 'In Loop'  
In Loop  
+ echo 'num: 6'  
num: 6  
++ expr 15 + 6  
+ sum=21  
+ echo 'sum: 21'  
sum: 21  
+ echo -e 'End of Loop\n'  
End of Loop  
  
+ read num
```

(Quick Note: You cannot pipe to read)

```
$ readpipetest
Contents of readpipetest file:
#####
#!/bin/bash
echo "Contents of readpipetest file:"
echo '#####'
cat readpipetest
echo '#####'
echo
echo "Contents of readpipetest being piped through read:"
cat readpipetest | read firstline
echo '#####'
echo "\"$firstline\""
echo '#####'
#####
```

```
Contents of readpipetest being piped through read:
#####
""
#####
```

- The pipe operator runs the next command in a subshell, whose created variables exist only while that subshell is being executed - thus `read` needs to be stuffed full of data and used immediately:

