Introduction to Bash

- Ian Miell
- Twitter: @ianmiell
- Ian.miell@gmail.com

Bash and Me

- **Used throughout career**
- **Never learned formally**
- **Stumbled around, lots of mistakes**
- **<u>Slowly</u> learned concepts and key points**
- **Wrote a book**

This Course

- Live Walkthroughs
  - Encourage you to follow - ‘Hard Way’ Method
- Exercises
- Group chat
- Materials:
  - https://github.com/ianmiell/introduction-to-bash

Pre-Requisites

- **Familiar with command line**
- **Bash version 4+**
  - $ echo $SHELL
  - $ bash --version
  - <4 is still ok
- **Using zsh (eg on Mac)**
  - **Type ‘bash’ to get a bash shell**
- **Basic shell utilities (eg grep, cat, ls)**
- **Any editor (I use vim)**

Why This Course?

- **Bash is everywhere**
- **Shells are everywhere**
- **Work with it every day**
- **Taken for granted that it’s known**
- **Studying it pays massive dividends**
  - Gateway to deeper OS concepts

Bash is under-served

- Man page is hard to follow if you don’t know the jargon
- One-liners are easy to find but concepts give you real power
- Guides that assume knowledge you may not have

Target Audiences

- **No knowledge assumed**
  - Advanced questions outside the course please
- **‘Hardly/never used bash’**
  - Coverage of 90% of bash features
- **‘Used bash casually for a while’**
  - Refresher on some topics, learn some new things
- **‘Used bash for years, but never studied’**
  - A-ha moments

Ever been confused by…?

- Diffference between ‘[‘ and ‘[[‘
- Globs vs regexes
- Single vs double quotes
- Difference between `` and $()
- How a bash script is created

Recently I've used bash to…

- Fix a Terraform script
- Write and debug various CI/CD pipelines
- Robustly apply changes in a cloud-init VM script
- Automate the renaming of files with spaces in my backup folders
- Setup environments at work

Poll - Experience

- Never used bash
- Used bash for <2 years
- Used bash for >2 years
- Used bash for >5 years
- Studied bash seriously

Structure of Course

- Part I – Bash Basics
- Part II – Further Bash Basics
- Part III - Scripting

Discussion

- What do you want to achieve in bash?
  - Any specific goals?
  - What have you been frustrated by with bash?

Part I – Bash Basics

- 1.1 Bash background
- 1.2 Variables
- 1.3 Globs
- 1.4 Pipes and Redirects

1.1 What is Bash?

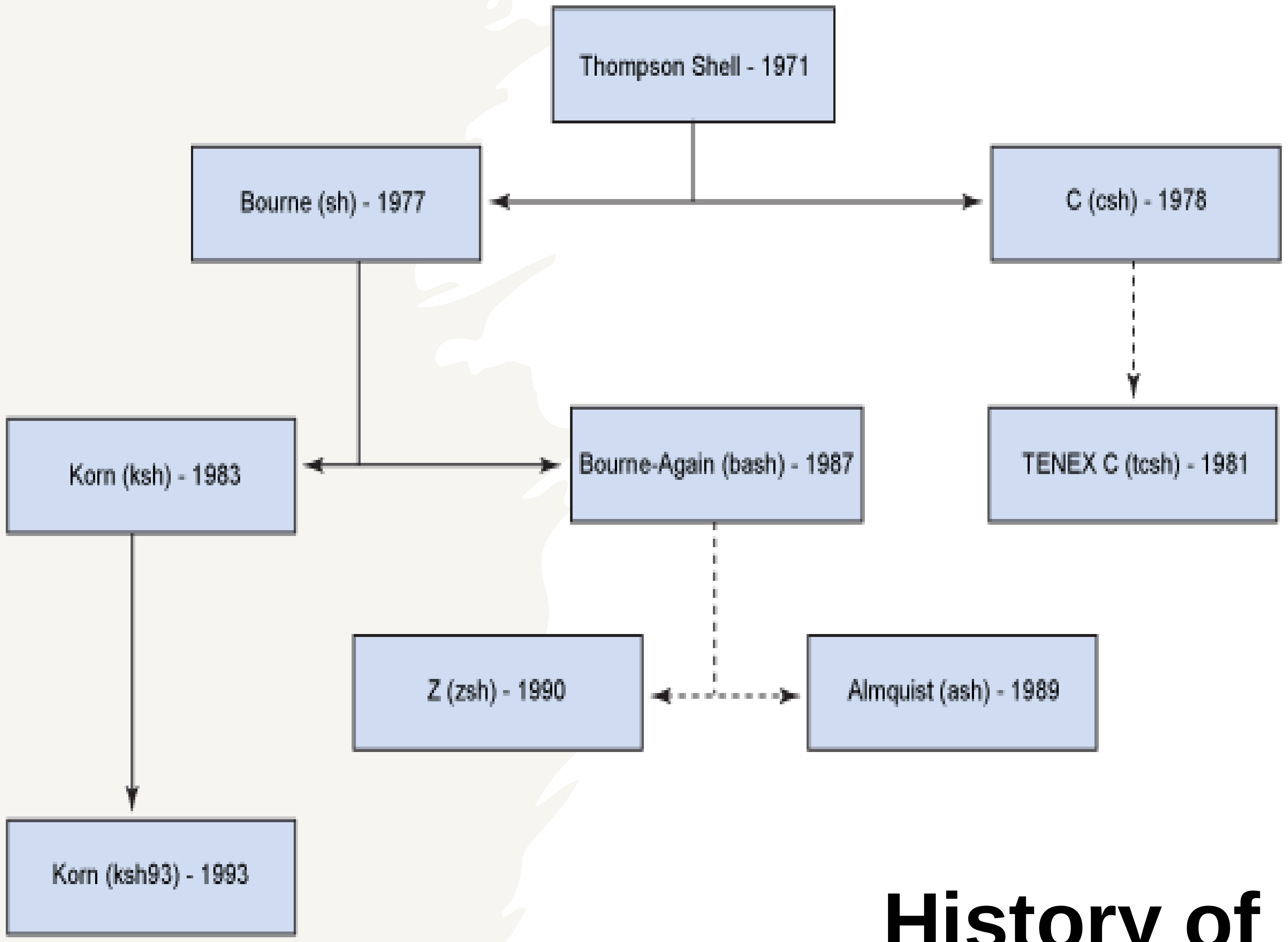
- What is a shell?
- A program takes input from a terminal
- Translates input into:
  - System calls
  - Calls to other programs
  - Computation within the bash program
- Bash excels at ‘gluing’ other commands together

Other shells

- sh
- ash
- ksh
- tcsh
- zsh

What is Bash? - Walkthrough

- **Run tcsh from bash**



History of Shells

Bash in the Market

- Most popular shell
- Lots of competition:
  - zsh now default on mac
  - fish is also popular
- Very rarely, you find servers that don’t have bash on still

1.2 Variables

- Basic variables
- Quoting variables
- ‘env’ and ‘export’
- Simple arrays

Variables - Recap

- $ dereferences
- Variables in double quotes are interpreted, single quotes not
- Exported variables are passed to programs run within the shell
- Env shows exported variables, ‘declare’ shows all variables

1.3 Globbing

- What is a glob?
- What does ‘*’ mean?
- Differences to regular expressions
    - Not familiar with regexes?
- Dotfiles

Recap - Globs

- What a glob is
- What a dotfile is
- Special directory files
- Globs, regexps and dots

1.4 Pipes and Redirects

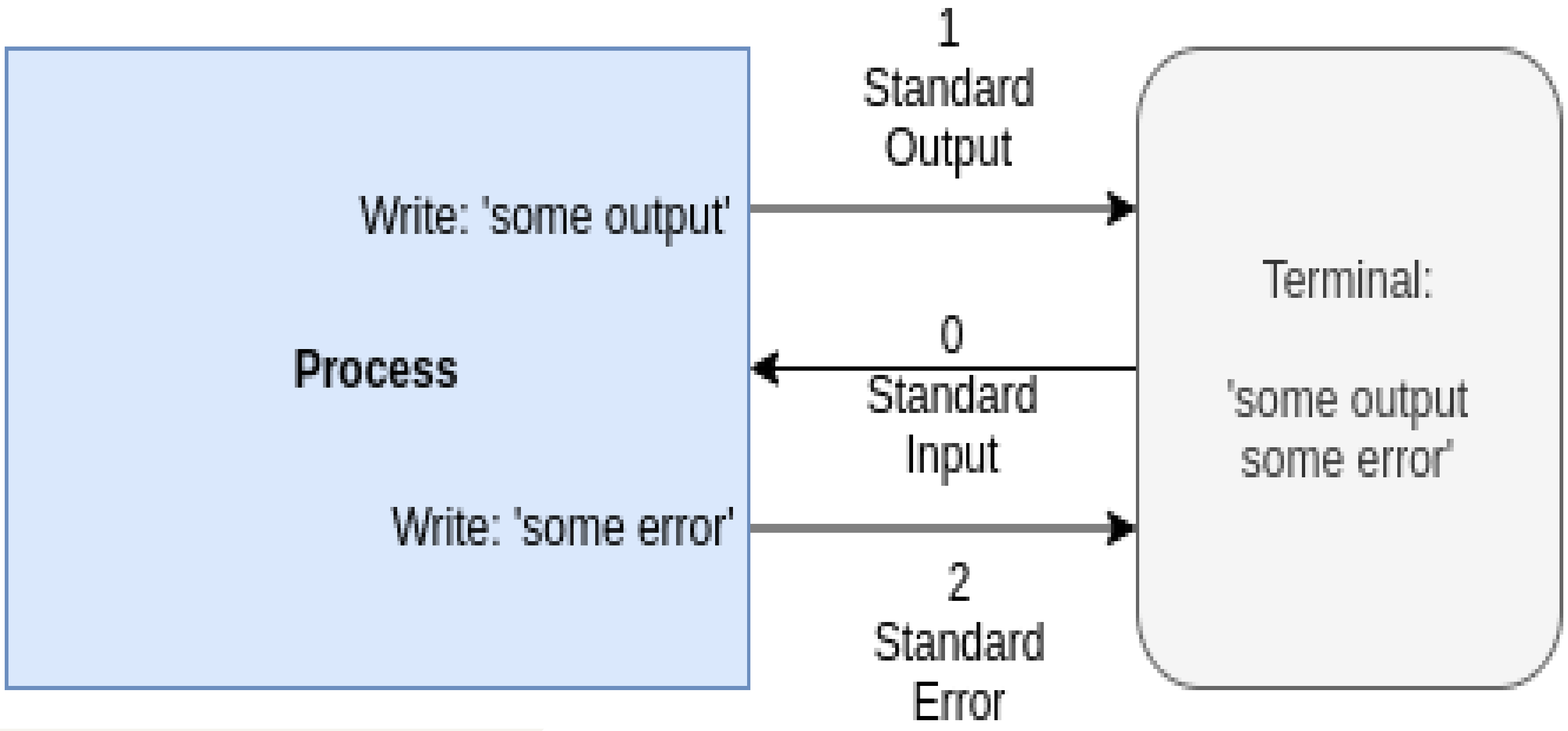
- Basic redirects
- Basic pipes
- File descriptors
- Special files
- Standard out vs standard error

Pipes and Redirects - Walkthrough

- Simple pipes and redirects
- Standard in/out/error
- File Descriptors

Default

No redirects



Pipes and Redirects

Pipes and Redirects - Walkthrough

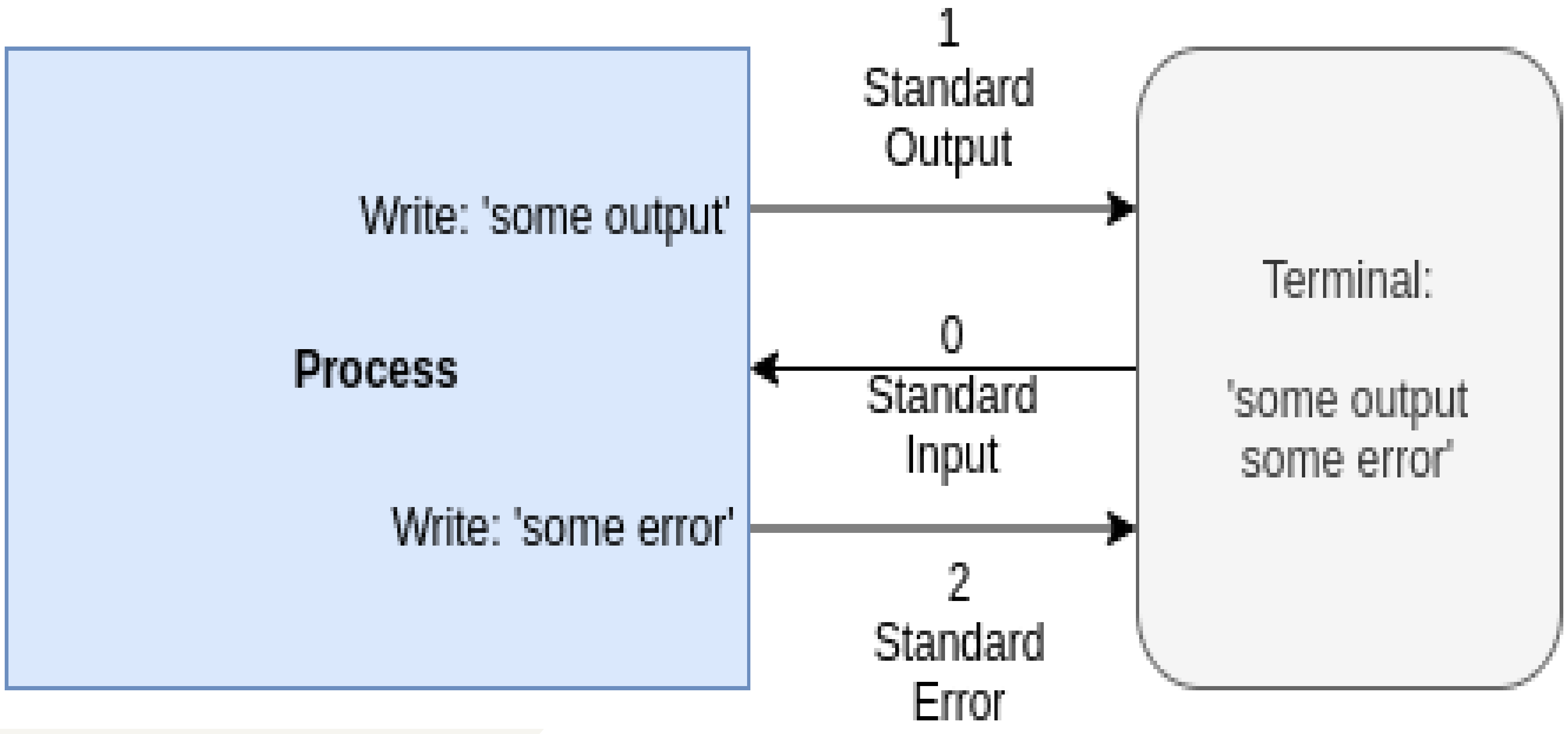
- Simple pipes and redirects
- Standard in/out/error
- File Descriptors

File Descriptors (I)

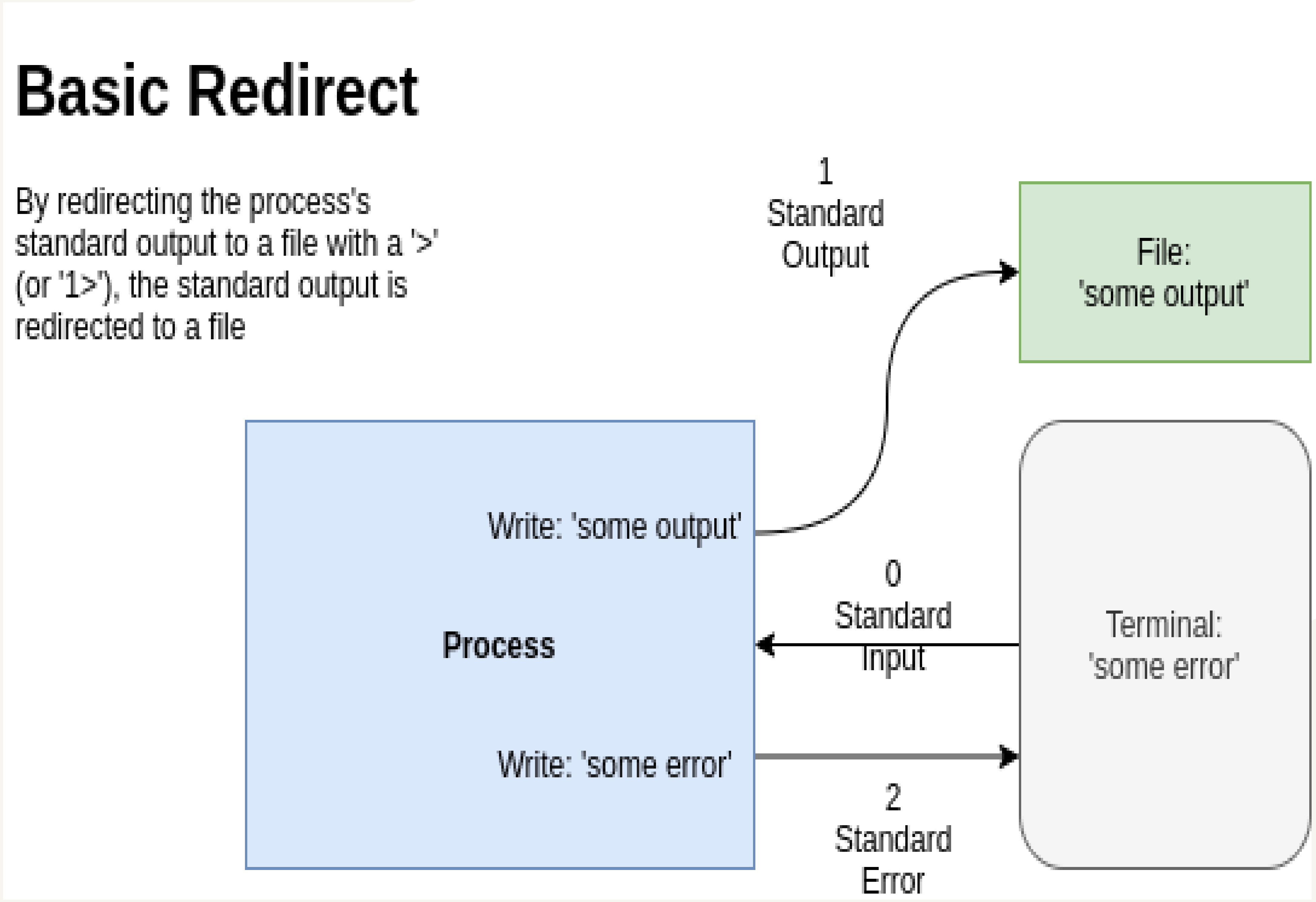
- Every process gets three file descriptors:
  - 0 - ‘standard input’
  - 1 - ‘standard output’
  - 2 - ‘standard error’
- ‘Normal’ output goes to file descriptor 1
- Programs generally output errors to file descriptor 2
- Normally ‘stderr’ and ‘stdout’ both go to the terminal – but you can change that!

Default

No redirects



Pipes and Redirects



Basic Redirect

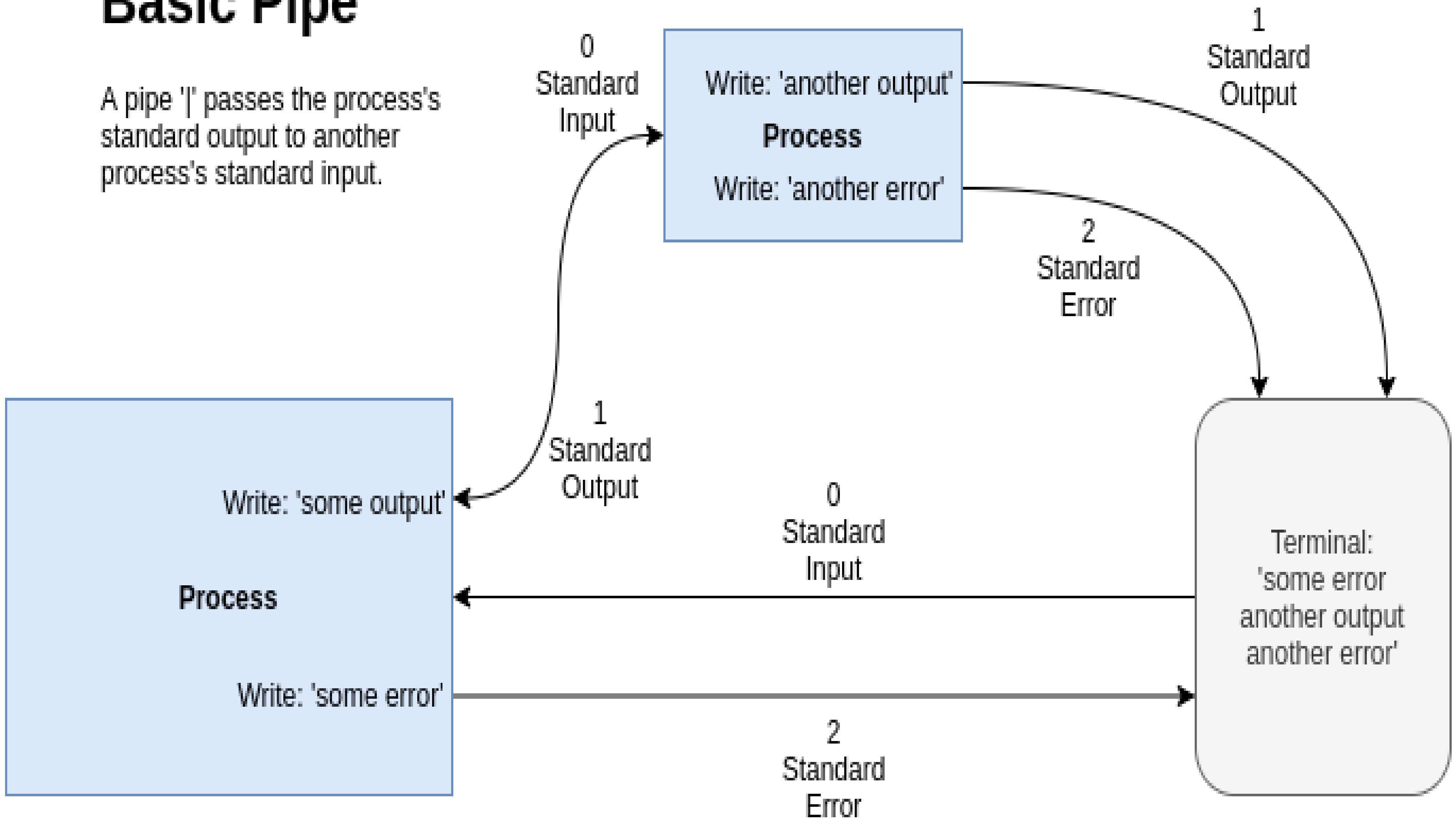
By redirecting the process's standard output to a file with a '>' (or '1>'), the standard output is redirected to a file

Pipes and Redirects

Pipes and Redirects

Basic Pipe

A pipe '|' passes the process's standard output to another process's standard input.



File Descriptors (II)

- ‘>’ operator sends standard output to a file
  - ‘1>’ is the same (1 is assumed)
- ‘2>’ sends standard error to a file
- ‘|’ sends standard output to a process
- Advanced, but often seen:
  - 2>&1 sends standard error to whatever standard output is pointed at
  - A way of sending ‘all’ output to a file

Recap – Pipes vs Redirects

- The main 3 file descriptors
- ‘>’ vs ‘|’
- *n*> and standard error
- 2>&1 and ordering

Part I Recap

- Globs
  - vs regexps
- Variables, arrays
- Pipes and redirects
- File descriptors

Part II – Further Bash Basics

- 2.1 Command Substitution
- 2.2 Functions
- 2.3 Tests
- 2.4 Loops
- 2.5 Exit Codes

Discussion

- Is bash a programming language?
- What is a programming language?
- Why has bash lasted so long?

2.1 Command Substitution

- The ‘$()’ operator
- $() vs ``
- Nesting

2.2 Functions in Bash

- Four types of command:
  - Function
  - Alias
  - Program
  - Builtin

2.3 Tests

- Bash tests
- Different ways of writing tests
- Logical operators
- Binary and unary operators
- ‘if’ statements

2.4 Loops

- ‘C’-style for loops
- ‘for’ loops over items ‘in’ lists
- ‘while’ loops
- ‘case’ statements

2.5 Exit Codes

- What an Exit Code is
- The ‘$?’ variable
- How to set one
- Exit Code conventions
- Other ‘special’ parameters

Standard Exit Codes

- **0 – OK**
- **1 – General Error**
- **2 – Misuse of shell builtin**
- **126 – Cannot execute**
- **127 – No file found matching command**
- **128 – Invalid exit value**
- **(128 + n) – Process killed with signal ‘n’**

Recap – Exit Codes

- Standard exit codes
- Exit code usage (eg grep)
- Setting exit codes
- ‘return’ing from functions
- Special parameters

Discussion / Recap – Part II

- Bash more as programming language:
  - Functions
  - Tests / ifs
  - Loops
  - Return/Exit codes
  - Process and command substitution

- $() vs ``

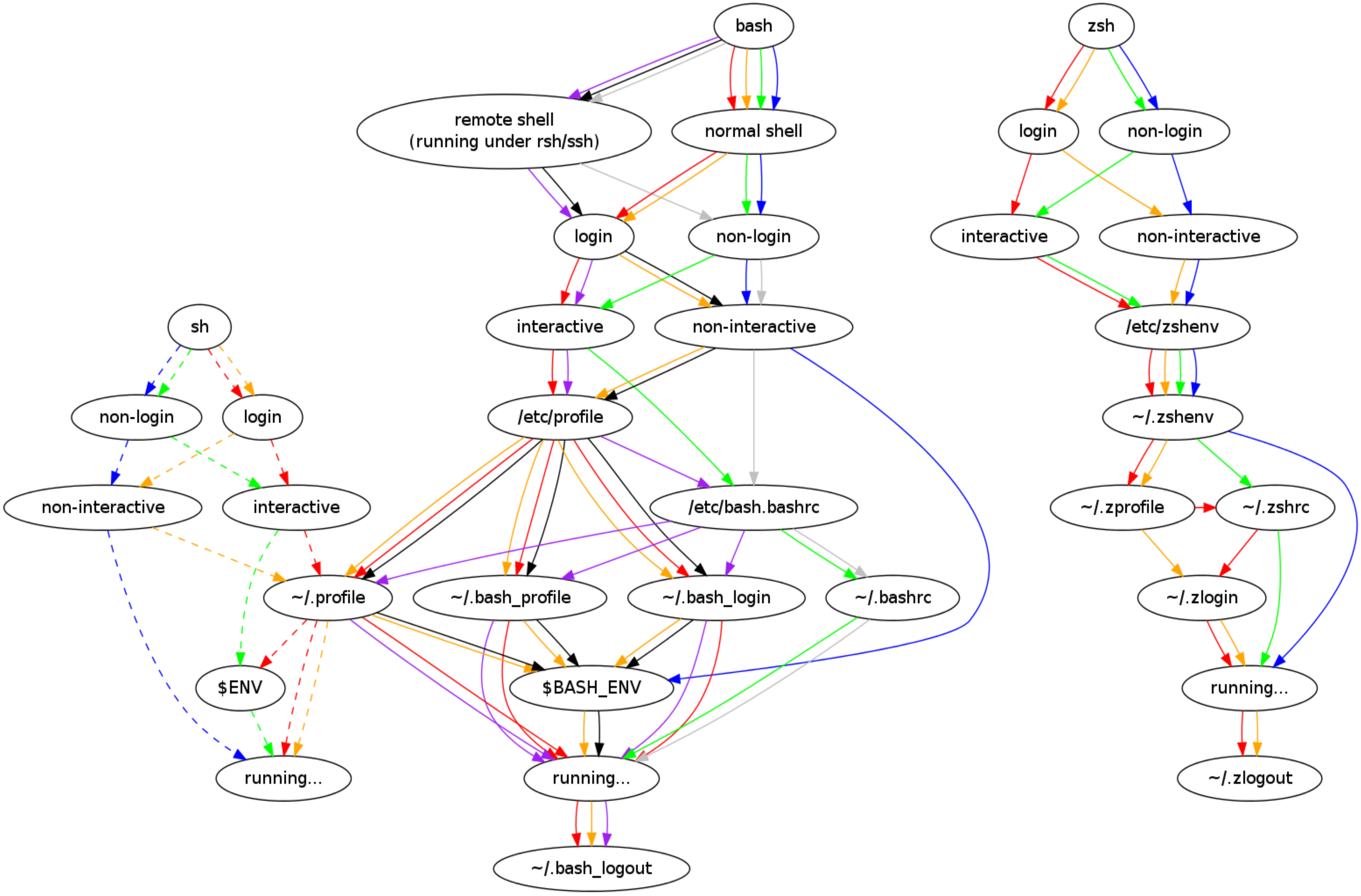
Part III - Scripting

- Scripts and Startup
- The ‘set’ Command
- Debugging in bash
- Subshells
- IFS

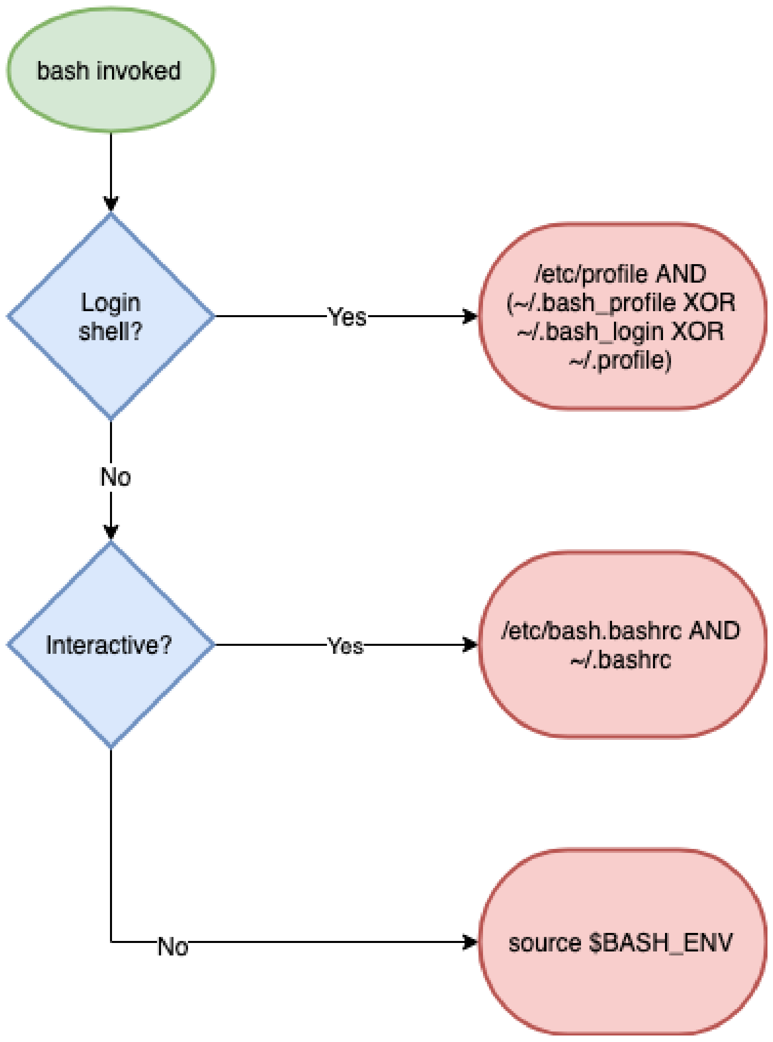
3.1 Scripts and Startup

- What shell scripts are
- What happens on bash startup
- This has cost me many hours!
- Executable files
- ‘source’ vs ‘./’

Walkthrough – Startup Explained



Walkthrough – Startup Explained (simpler)


bash invoked
Login shell?
Yes
/etc/profile AND (~/.bash_profile XOR ~/.bash_login XOR ~/.profile)
No
Interactive?
Yes
/etc/bash.bashrc AND ~/.bashrc
No
source $BASH_ENV

Recap - Scripts and Startup

- What shell scripts are
- How complex bash startup can be
- Keep diagram handy!

3.2 The ‘set’ builtin

- Setting options in bash
- What POSIX is
- Most useful options:
  - nounset
  - xtrace
  - errexit
- ‘set’ vs ‘shopt’

Recap - ‘set’

- Options: + off, - on
- POSIX
- Most common options
- shopt and set
- xtrace, nounset, errexit

3.3 Subshells

- What is a subshell?
- How to create a subshell
- Why they are useful
- () vs {}

3.4 Internal Field Separator

- aka IFS
- Why it’s important
- How to use it

Walkthrough – Spaces in Filenames

- ‘for’ looping over files
- The IFS shell variable
- The $” construct

Walkthrough – Spaces in Filenames

- Setting IFS
- The ‘find’ command and ‘xargs’
- find, xargs and the null byte separator

Part III – Discussion / Recap

- Shell Startup
- Practical bash usage
  - Shell options
  - Shell debugging
  - IFS

Optional – Advanced Bash

- Traps
- String manipulation
- Autocomplete
- Walkthrough a ‘real’ script

4.1 Jobs and Traps

- Background jobs
- Traps and signals
- The ‘kill’ command
- The ‘wait’ builtin
- Trapping signals
- Process groups

Standard Exit Codes - Refresher

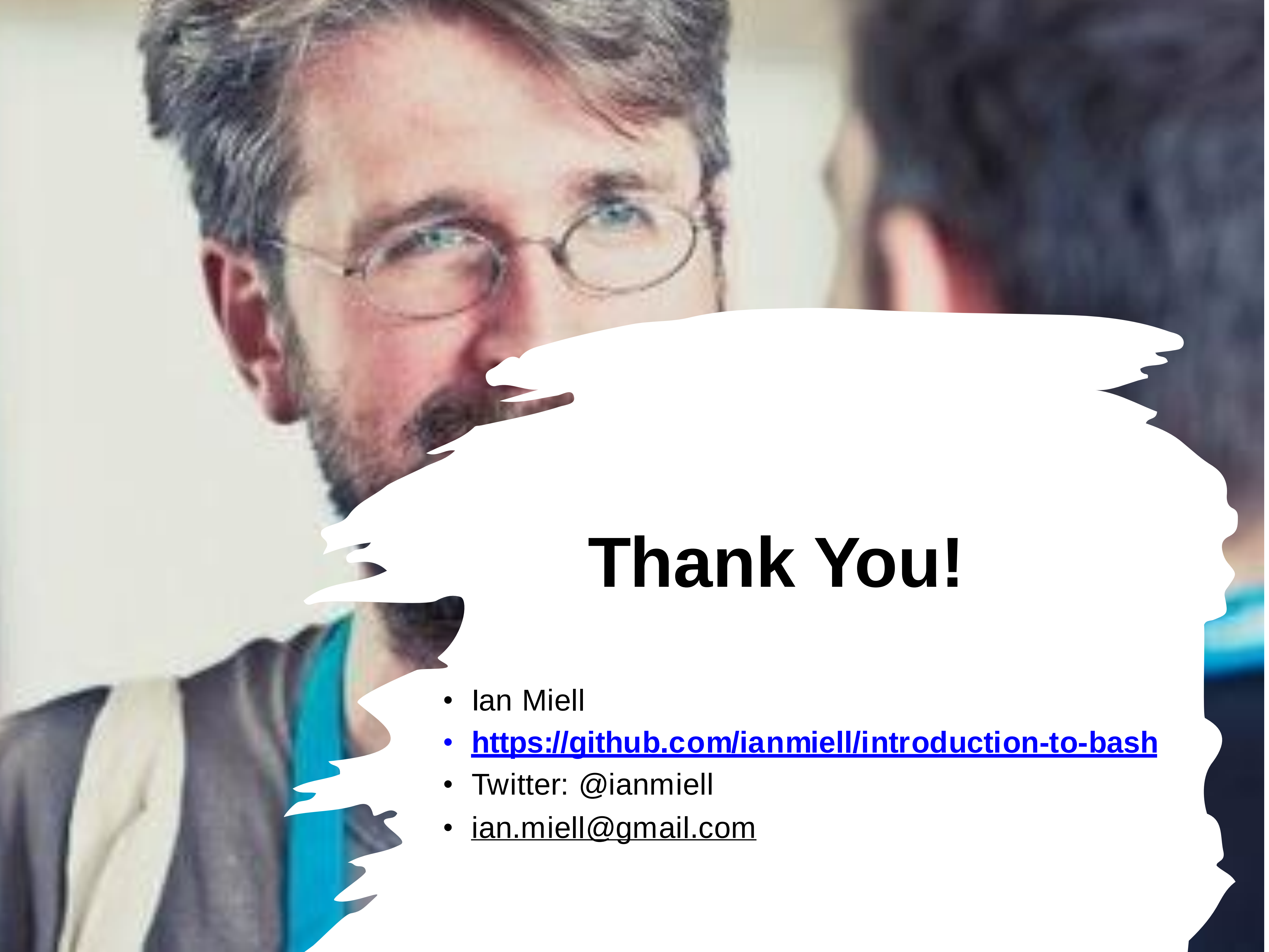
- **0 – OK**
- **1 – General Error**
- **2 – Misuse of shell builtin**
- **126 – Cannot execute**
- **127 – No file found matching command**
- **128 – Invalid exit value**
- **(128 + n) – Process killed with signal ‘n’**

4.2 Process Substitution

- The ‘<()’ operator
- Substitution of file arguments

- The ‘<()’ operator
- Substitution of file arguments

Process Subsitution - Walkthrough



Thank You!

- Ian Miell
- https://github.com/ianmiell/introduction-to-bash
- Twitter: @ianmiell
- ian.miell@gmail.com