Intro, UNIX, Bash, C

CS 5007: Systems

Adrienne Slaughter

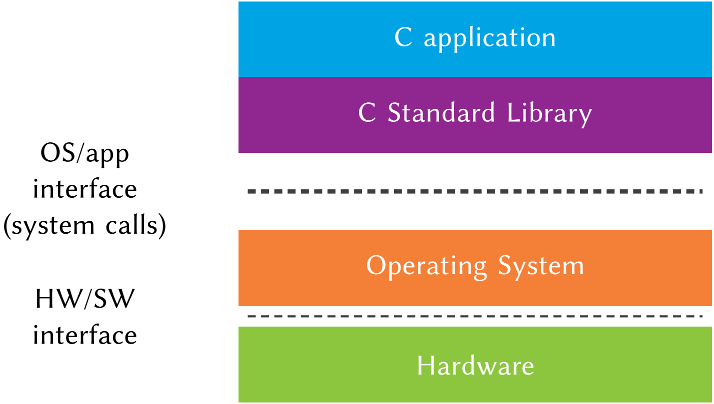
Northeastern University

March 13, 2019

Section 1

Intro to CS 5007

The Big Picture: What is a System?



Outline

- Systems:
  - File Systems
- C Skills:
  - Libraries, Building shared libraries
  - C I/O
- Intro to Architecture

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes of 5007:

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes of 5007:
  - Become a more effective programmer

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes of 5007:
  - Become a more effective programmer
  - Able to find and kill bugs more efficiently

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes of 5007:
  - Become a more effective programmer
  - Able to find and kill bugs more efficiently
  - Prepare for later "systems" classes:

[1] Bryant and O'Halloran

Abstraction is good, but don't forget Reality[1]

Course Theme:

- Most CS courses emphasize abstraction
  - Abstract datatypes
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes of 5007:
  - Become a more effective programmer
  - Able to find and kill bugs more efficiently
  - Prepare for later "systems" classes:
    - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems, Storage Systems, etc.

[1] Bryant and O'Halloran

Memory Matters! [2]

- Memory is not unbounded
  - It needs to be allocated and managed
  - Many applications are memory dominated
- Memory bugs are nasty
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory can greatly impact program performance
  - Adapting your program to the characteristics of the memory system can have huge impacts.

[2] Bryant and O'Halloran

Memory in C[3]

- No memory management in C, C++
  - Out of bounds in array
  - Invalid pointer refs
  - Poor use of malloc/free
- Bugs. Icky bugs. Hard to find bugs.
  - Whether the bug can be seen depends on the system and the compiler
  - "Action at a distance"
    - Bug could be observed long after it's generated
- How to deal with it?
  - Well, you could use Java, Python, ...
  - Be aware.
  - Work with tools that help you detect memory issues (e.g. Valgrind)

[3] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
  - Performance increase of 10:1 depending on how your code is written

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
  - Performance increase of 10:1 depending on how your code is written
  - Optimization happens at multiple levels: algorithm, data representation, procedures, loops, ...

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
  - Performance increase of 10:1 depending on how your code is written
  - Optimization happens at multiple levels: algorithm, data representation, procedures, loops, ...
- Must understand the system to optimize performance

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
  - Performance increase of 10:1 depending on how your code is written
  - Optimization happens at multiple levels: algorithm, data representation, procedures, loops, ...
- Must understand the system to optimize performance
  - How programs are compiled and executed

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
  - Performance increase of 10:1 depending on how your code is written
  - Optimization happens at multiple levels: algorithm, data representation, procedures, loops, ...
- Must understand the system to optimize performance
  - How programs are compiled and executed
  - How to measure programs and bottlenecks

[4] Bryant and O'Halloran

There's more to performance than asymptotic analysis[4]

- Constant factors matter too!
- Exact operation counts do not always predict performance
    - Performance increase of 10:1 depending on how your code is written
    - Optimization happens at multiple levels: algorithm, data representation, procedures, loops, ...
- Must understand the system to optimize performance
    - How programs are compiled and executed
    - How to measure programs and bottlenecks
    - How to improve performance without destroying code modularity and generality

[4] Bryant and O'Halloran

Memory System Performance Example[5]

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms 2.0 GHz Intel Core i7 Haswell 81.8ms

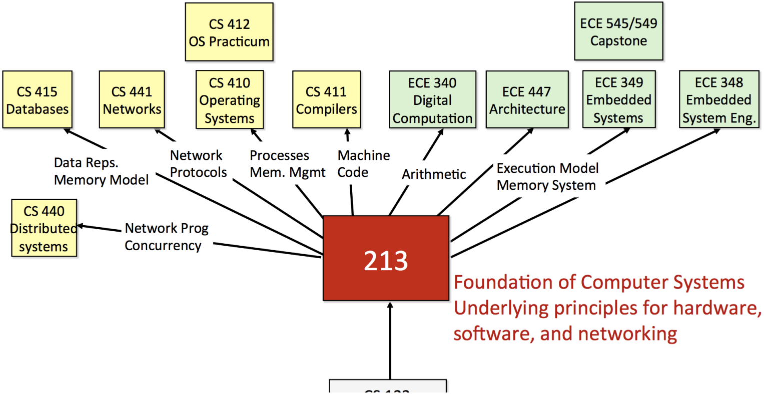
[5] Bryant and O'Halloran

Computers do more than execute programs [6]

- They need to get data in and out
  - I/O system is critical to performance
- They communicate with each other over networks
  - System-level issues arise when working with networks
  - Concurrent operations by autonomous processes
  - Coping with unreliable media
  - Cross platform compatibility
  - Complex performance issues

[6] Bryant and O'Halloran

CS Curriculum[7]



This is CMU's undergrad curriculum, but helpful for context.

[7] Bryant and O'Halloran

(Expected) Course Progression

- Week 1: Getting Started. What are systems. What is Unix/Linux/*nix. Bash.
- Week 2: More C. Libraries. File systems.
- Week 3: Memory Hierarchy. Architecture.
- Week 4: Multi-threading. Concurrency.
- Week 5: Networking
- Week 6: Synchronization. Deadlocks.
- Week 7: Wrap up, overflow. Final project help.

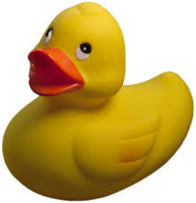
Assignment Overview

Assignments build on each other

- Assignment 6: Experiment with BASH and *nix. Start building a more advanced library of data structures.
- Assignment 7: Write Chained Hashtable in C. Build library of data structures.
- Assignment 8: Build a File System indexer using your C implementations from A1 and A2.
- Assignment 9: Make your FS Indexer disk-based rather than memory-based.
- Assignment 10/Final project: Client-server project.

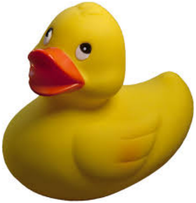
Collaboration and Academic Integrity

- You can talk to others about the ideas, but all write-ups and answers must be your own.



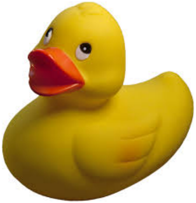
Collaboration and Academic Integrity

- You can talk to others about the ideas, but all write-ups and answers must be your own.
- Collaboration is GOOD!



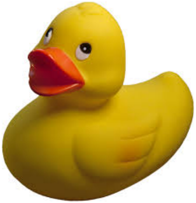
Collaboration and Academic Integrity

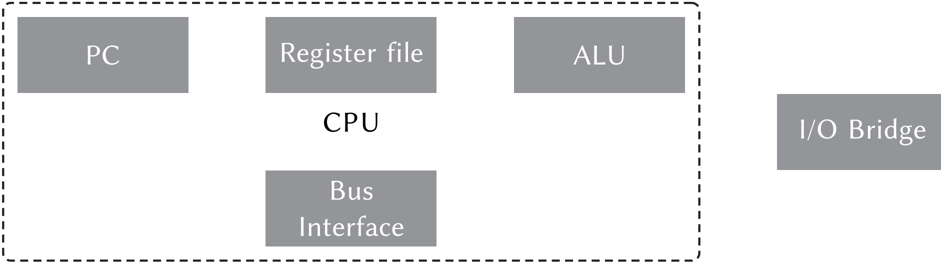
- You can talk to others about the ideas, but all write-ups and answers must be your own.
- Collaboration is GOOD!
- Cheating is BAD!



Collaboration and Academic Integrity

- You can talk to others about the ideas, but all write-ups and answers must be your own.
- Collaboration is GOOD!
- Cheating is BAD!
- Talk about how to do things, ask questions, use the rubber ducky.




PC
Register file
ALU
CPU
Bus Interface
I/O Bridge

Section 2

Intro to Unix

What is Unix?

An Operating System!

- Built on the philosophy of composing simple tools together
- Each tool does one thing
  - Usually really well, with lots of different options
- Do something more complicated by ***piping*** results (output) of one tool as the input of another

Useful Commands

`cat`	Prints out the contents of a file to the terminal
`less, more`	Shows a file without opening it
`tail/head`	Gives you the first/last few lines of a file
`grep`	Search for instances of a string. Can use regex!
`awk`	Simple processing of a line of a file
`sed`	“Stream EDitor”: Simple processing lines in a file
`sort, uniq`	Miscellaneous
`locate, find`	Find files/directories
`cut, paste, join`	Manipulating files
`comm, diff`	Find differences/similarities
`man, apropos`	How to get help and find a command

Useful Commands

`cat`	Prints out the contents of a file to the terminal
`less, more`	Shows a file without opening it
`tail/head`	Gives you the first/last few lines of a file
`grep`	Search for instances of a string. Can use regex!
`awk`	Simple processing of a line of a file
`sed`	"Stream EDitor": Simple processing lines in a file
`sort, uniq`	Miscellaneous
`locate, find`	Find files/directories
`cut, paste, join`	Manipulating files
`comm, diff`	Find differences/similarities
`man, apropos`	How to get help and find a command

Do the tutorial!

Section 3

Intro to Bash

What is Bash?

- Bourne Again SHell
- "The Terminal" or "The Command Line"
- It's usually bash; sometimes different
  - csh/tcsh (c)
  - ksh (korn)
  - zsh (z)– a great one!
- What shell are you running?
  - `echo $SHELL`

Some notes about working on the command line

- Piping: `|` pipes the output of one command to another
- Dump to a file: `>`

Section 4

Bash Scripts

What is a shell script?

A file containing a group of commands to perform a task.

What is a shell script?

A file containing a group of commands to perform a task.

- The shell is two things:
  - Interface to the system (via the command line)
  - Scripting language interpreter

What is a shell script?

A file containing a group of commands to perform a task.

- The shell is two things:
  - Interface to the system (via the command line)
  - Scripting language interpreter
- This means we can put the commands into a file, which is then interpreted by the shell, executing the commands.

What is a shell script?

A file containing a group of commands to perform a task.

- The shell is two things:
  - Interface to the system (via the command line)
  - Scripting language interpreter
- This means we can put the commands into a file, which is then interpreted by the shell, executing the commands.
- Things done on the command line can be done in a script, and vice versa

Writing a Shell Script

Writing a Shell Script

1. Write a script.
   - It's an ordinary text file
   - Using a text editor with syntax highlighting is nice :)

Writing a Shell Script

1. Write a script.
   - It's an ordinary text file
   - Using a text editor with syntax highlighting is nice :)
2. Make the script executable
   - Not all text files can be treated as programs!
   - Change permissions to make it executable

Writing a Shell Script

1. Write a script.
   - It's an ordinary text file
   - Using a text editor with syntax highlighting is nice :)
2. Make the script executable
   - Not all text files can be treated as programs!
   - Change permissions to make it executable
3. Put the script where it can be found
   - The shell looks in certain directories for executables (if not otherwise specified)

Script File Format

```
#!/bin/bash

# This is a first script

echo "Hello, CS5007!"

```

Listing 1: hello_world

```
[me@mybox]$ chmod 755 hello_world

```

Making Executable

```
[me@mybox]$ ./hello_world

```

Running

Running the Script

```
[me@mybox]$ ./hello_world

```

Running

Note: Here, we're specifying the path to the script name.

To make it executable without specifying the path, put it in the $PATH.

Try: `echo $PATH`

You should get a listing of directories on the path, including something like `:/bin:`

You might have to create that directory: `mkdir ~/bin`

Running the Script

```
[me@mybox ~]$ mkdir ~/bin
[me@mybox ~]$ mv hello_world ~/bin
[me@mybox ~]$ hello_world
Hello, CS5007!

```

Listing 2: Running

By copying the script to the bin directory, we can run without specifying the path (that is, with the `./`)

To make it executable without specifying the path, put it in a directory in `$PATH`.

Where to put scripts

- `~/bin` : Scripts intended for personal use
- `/usr/local/bin` : Scripts intended for everyone on a system
- `/usr/local/sbin` : Scripts intended for the system admin to use
- `/usr/local/...` : Locally supplied software (scripts or compiled programs)

Formatting Tricks

- Long Option Names:
  - In scripts, use the long version of options to improve readability
  - On commandline, use the short version
- Use `\` to continue a really long command onto multiple lines.

Naming your shell script file

`foo.sh`? `bar.bash`?

Which is right?

Well, it kinda doesn't matter... The extension doesn't have much meaning in Linux, other than indicating the type of file.

Extensions are by convention.

`*.sh` refers to a generic/portable shell script

`*.bash` refers to a script that only works properly on bash.

Not using an extension is okay.

In this class, use `.sh`

Code Along: Write a Script to Generate a Sys Report

Our Plan:

1. Create an HTML template to display a report
2. Add some data to the report
3. Using variables and constants
4. Populating variables and constants with system info
5. Generating a report

Making the HTML template

This is the framework for our report:

```
<HTML>
  <HEAD>
        <TITLE> </TITLE>
</HEAD>
<BODY>
        <H1></H1>
        <P>Data here</P>
  </BODY>
</HTML>

```

How do we write a program to generate this file?

First, create the script:

```
[me@mybox ~]$ touch ~/bin/myreport.sh
[me@mybox ~]$ open ~/bin/myreport.sh
[me@mybox ~]$ chmod 755 ~/bin/myreport.sh

```

Listing 3: Creating the script file

Writing a script to generate the .html content

```bash
#!/bin/bash

# Sys Report Generation

echo "<HTML>"
echo "   <HEAD>"
echo "         <TITLE> </TITLE>"
echo "</HEAD>"
echo "<BODY>"
echo "         <H1></H1>"
echo "         <P>Data here</P>"
echo "   </BODY>"
echo "</HTML>"

```

Listing 4: report.sh

`$ report.sh` dumps the HTML to the terminal window.

Running `$ report.sh > report1.html` produces an HTML file that can be opened in a browser.

Can we make it simpler?

Can we simplify?

```
#!/bin/bash

# Sys Report Generation

echo "<HTML>
  <HEAD>
        <TITLE> </TITLE>
 </HEAD>
 <BODY>
        <H1></H1>
        <P>Data here</P>
  </BODY>
 </HTML>"

```

Listing 5: report2.sh

Look ma, one `echo`!

Still works.

Time to put some content in our template.

Let's add some content

```
#!/bin/bash

# Sys Report Generation

echo "<HTML>
  <HEAD>
        <TITLE>System Information Report</TITLE>
 </HEAD>
 <BODY>
        <H1>System Information Report</H1>
        <P>Data here</P>
  </BODY>
 </HTML>"

```

Listing 6: report3.sh

Subsection 2

Variables and Constants

Don't replicate strings— Use variables!

```
#!/bin/bash

# Sys Report Generation

title="System Information Report"

echo "<HTML>
  <HEAD>
         <TIME>$title</TITLE>
 </HEAD>
 <BODY>
         <H1>$title</H1>
         <P>Data here</P>
  </BODY>
 </HTML>"

```

Listing 7: report4.sh

Variables in Shell Scripts

- To use a new variable, just use a new variable!
  - Creating: `myVar=5`
  - Using later: `echo $myVar`
  - Don't need to declare, as in C
- Naming rules
  - Consist of alphanumeric characters, and underscores
  - First character must be a letter or underscore
  - No spaces or punctuation
  - Constants should be all caps (by convention)
  - There is no way to enforce a constant

Make those constants uppercase

```
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"

echo "<HTML>
  <HEAD>
        <TITLE>$TITLE</TITLE>
 </HEAD>
 <BODY>
        <H1>$TITLE</H1>
        <P>Data here</P>
  </BODY>
 </HTML>"

```

Listing 8: report5.sh

Add in some more details

Add some more details...

```
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<HTML>
  <HEAD>
        <TITLE>$TITLE</TITLE>
 </HEAD>
 <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
  </BODY>
 </HTML>"

```

Listing 9: report6.sh

Let’s run it!

```
[me@mybox ~]$ report
<HTML>
  <HEAD>
        <TITLE>System Information Report for Adriennes-MacBook-Pro.local</TITLE>
 </HEAD>
 <BODY>
        <H1>System Information Report for Adriennes-MacBook-Pro.local</H1>
        <P>Generated 02/28/2018 01:33:16 PM PST, by ahslaughter</P>
 </BODY>
 </HTML>

```

Listing 10: output

Redirect to a file to create a .html file we can open:

`report6 > report.html`

`open report.html`

“Here” document

```
command << token
text
token

```

`command` is the name of a command

`token` indicates the end of the text to send to the command

Everything between the tokens is text sent to the command.

Token must be alone on the line with no trailing spaces

Using a here document with our example:

```
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
<BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
</BODY>
</HTML>
_EOF_

```

Listing 11: report7.sh

Adding more info

We've got a great start!

We'd really like a little more useful info.

Can we add:

Adding more info

We've got a great start!

We'd really like a little more useful info.

Can we add:

- **System uptime and load.** Amount of time the system has been up, and average number of tasks running over a time period.

Adding more info

We've got a great start!

We'd really like a little more useful info.

Can we add:

- **System uptime and load.** Amount of time the system has been up, and average number of tasks running over a time period.
- **Disk space.** Overall use of space on system's storage devices.

Adding more info

We've got a great start!

We'd really like a little more useful info.

Can we add:

- **System uptime and load.** Amount of time the system has been up, and average number of tasks running over a time period.
- **Disk space.** Overall use of space on system's storage devices.
- **Home space.** Amount of storage being used by each user.

Adding more info

We've got a great start!

We'd really like a little more useful info.

Can we add:

- **System uptime and load.** Amount of time the system has been up, and average number of tasks running over a time period.
- **Disk space.** Overall use of space on system's storage devices.
- **Home space.** Amount of storage being used by each user.

If we had a command for each of these things, it would be super easy...

We could do something like this:

```
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
<BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
    <P>$(report_uptime)</P>
    <P>$(report_disk_space)</P>
    <P>$(report_home_space)</P>
</BODY>
</HTML>
_EOF_

```

Adding commands...

There are 2 ways we can add commands to our script for this functionality:

Adding commands...

There are 2 ways we can add commands to our script for this functionality:

1. Write a new script and put it in one of the directories on our path

Adding commands...

There are 2 ways we can add commands to our script for this functionality:

1. Write a new script and put it in one of the directories on our path
2. Write a ***shell function*** in our script

Shell Functions

2 ways to declare:

```
function name{
  commands
  return
}

```

```
name(){
  commands
  return
}

```

They are equivalent and interchangeable.

BUT: a function ***must*** contain at least one command. (`return` counts)

Let's start working on it...

```
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime(){
  return
}

report_disk_space(){
  return
}

report_home_space(){
  return
}

cat << _EOF_
<HTML>
  <HEAD>
```

Back to working on our report: Stubbing out functions

```bash
#!/bin/bash

# Sys Report Generation

TITLE="System Information Report for $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime(){
  echo "uptime report generated"
  return
}

report_disk_space(){
  echo "disk space report generated"
  return
}

report_home_space(){
  echo "home space report generated"
  return
}

```

We stub out functions to help us make sure that we ***always have a working script***.

`apropos` : Helps you find the command that you need right now but can't remember the name of.

Try:

```
[ahslaughter@adriennes-mbp:~]\$ apropos uptime

```

Implementing the functions

```
report_uptime(){
  # echo "uptime report generated"
  cat << _EOF_
    <H2>System Uptime</H2>
    <PRE>$(uptime)</PRE>
_EOF_
  return
}

report_disk_space(){
  # echo "disk space report generated"
  cat << _EOF_
    <H2>Disk Space</H2>
    <PRE>$(df -h)</PRE>
_EOF_
  return
}

report_home_space(){
  # echo "home space report generated"
  cat << _EOF_
    <H2>Home Space</H2>
    <PRE>$(du -sh ~)</PRE>
```

Let's look at the whole thing...
(on the command line)

Local versus Global Variables

So far, we've only looked at global variables.

How do we use local variables in shell scripts?

- Local variables only exist within the shell function they are defined
- They cease to exist when the function terminates
- Specify a local variable by using the keyword `local`

Using local vars

```bash
#!/bin/bash

# local-vars

foo=0 # Global foo

func_1 () {
  local foo
  foo=1
  echo "func_1: foo = $foo"
}

func_2 () {
  local foo
  foo=2
  echo "func_2: foo = $foo"
}

echo "global: foo = $foo"
func_1
echo "global: foo = $foo"
func_2
echo "global: foo = $foo"
```

Using local vars

```
ahslaughter@adriennes-mbp:~$ local-vars
global: foo = 0
func_1: foo = 1
global: foo = 0
func_2: foo = 2
global: foo = 0
```

Listing 17: running local-vars

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command
- Using variables in our shell script

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command
- Using variables in our shell script
- Piped output to a file, which we could then use

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command
- Using variables in our shell script
- Piped output to a file, which we could then use
- Used `apropos` to find commands

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command
- Using variables in our shell script
- Piped output to a file, which we could then use
- Used `apropos` to find commands
- Added shell functions

What have we done?

We've just learned:

- A shell script is just a text file with shell commands.
- How to make a script file executable
- Where to put a shell script to make it easy to run
- Made a script to create a system report
- Using "here" files to pipe text to a command
- Using variables in our shell script
- Piped output to a file, which we could then use
- Used `apropos` to find commands
- Added shell functions
- Learned about local vs global variables

Subsection 3

Flow Control: Branching/Conditionals

The command `if` can be used two ways:

```
test expression
```

```
[ expression ]
```

Another thing you'll see:

```
[[ expression ]]
```

where *expression* evaluates to either true or false. It's like the single bracket `test`, but includes `string1 =~regex`

Integers use `(( $var == 0 ))`: If the result of the arithmetic evaluation is non-zero, then true.

`if` in action

```
x=5

if [ $x=5 ]; then
  echo "x = 5"
else
  echo "x does not equal 5"
fi

```

Listing 18: if

Comparisons

Refer to Bash Conditional Expressions to see more about the comparisons and flags you can use:
https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html

- Does a file exist?
- How do two numbers compare?
- Compare or evaluate a string
- ...

Reading Keyboard Input

read [-options] [variable]

```
echo -n "Please enter an integer"
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then
  if [ $int -eq 0 ]]; then
    echo "$int is zero"
  else
    ...
  fi
else
  echo "Input is not an integer" >&2
  exit 1
fi

```

Listing 19: read

Subsection 4

Flow Control: Looping

Looping with `while`

while commands; do commands; done

```
count = 1

while [ $count -le 5 ]; do
  echo $count
  count=$((count + 1))
done
echo "Finished"

```

Listing 20: while

Breaking out of a loop: `break` , `continue`

`break`

`continue`

```
count = 1

while [ $count -le 5 ]; do
  echo $count
  count=$((count + 1))
done
echo "Finished"

```

Listing 21: read

Subsection 5

Parameters

Positional Parameters

- You can provide parameters to a shell script on the command line
- In the script, refer to them via number, or position
- The i^{th} parameter is referred to as `$i` in your script
- Access parameters 0 - 9 by using `$0`, `$1`, etc.
- Access parameters in position 10 and up by using braces: `${55}`
- Determine number of arguments passed to your script: `$#`

Parameters

```
#!/bin/bash

echo "
\$0 = $0
\$1 = $1
\$2 = $2
"
```

Listing 22: parameters

```
ahslaughter@adriennes-mbp:~$ touch ~/bin/params
ahslaughter@adriennes-mbp:~$ chmod 755 ~/bin/params
ahslaughter@adriennes-mbp:~$ params

  $0 = /Users/ahslaughter/bin/params
  $1 =
  $2 =
```

Listing 23: parameters

Positional Parameters

- You can provide parameters to a shell script on the command line
- In the script, refer to them via number, or position
- The i^{th} parameter is referred to as `$i` in your script
- Access parameters 0 - 9 by using `$0`, `$1`, etc.
- Access parameters in position 10 and up by using braces: `${55}`
- Determine number of arguments passed to your script: `$#`
- `$0` will always contain the first item on the command line, which is the pathname of the program being executed.

Parameters

```
ahslaughter@adriennes-mbp:~$ params foo bar

  $0 = /Users/ahslaughter/bin/params
  $1 = foo
  $2 = bar

```

Listing 24: Passing 2 arguments

Parameters

```
#!/bin/bash

echo "
Num Params: $#
  \$0 = $0
  \$1 = $1
  \$2 = $2
  "

```

Listing 25: Getting the number of params provided

```
ahslaughter@adriennes-mbp:~$ params *

Num Params: 31
  $0 = /Users/ahslaughter/bin/params
  $1 = AndroidDev
  $2 = Applications

```

Listing 26: Results (* is the files in this directory)

Section 5

Review of C

C: A refresher

- “Low-level” language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly

C: A refresher

- "Low-level" language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly
- Procedural (not object-oriented)

C: A refresher

- “Low-level” language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly
- Procedural (not object-oriented)
- “Weakly-typed” or “type-unsafe”

C: A refresher

- "Low-level" language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly
- Procedural (not object-oriented)
- "Weakly-typed" or "type-unsafe"
- C was developed as the system programming language for Unix.

C: A refresher

- "Low-level" language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly
- Procedural (not object-oriented)
- "Weakly-typed" or "type-unsafe"
- C was developed as the system programming language for Unix.
  - The kernel (core part of the system) and supporting tools and libraries were written in C

C: A refresher

- "Low-level" language that allows us to exploit underlying features of the architecture, but easy to fail spectacularly
- Procedural (not object-oriented)
- "Weakly-typed" or "type-unsafe"
- C was developed as the system programming language for Unix.
  - The kernel (core part of the system) and supporting tools and libraries were written in C
- It's a small, simple language

C File Format

```
#include <system_files>
#include "local_files"
#define macro_name macro_expr
/* declare functions */
/* declare external variables & structs */
int main(int argc, char* argv[]) {
/* the innards */
}
/* define other functions */

```

Listing 27: C Program Layout

C Syntax: main

- Command line arguments

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)
  - `argc` needed because C doesn't know how long an array is and needs to allocate for it

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)
  - `argc` needed because C doesn't know how long an array is and needs to allocate for it
  - `argv` is an array of pointers to the arguments as strings

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)
  - `argc` needed because C doesn't know how long an array is and needs to allocate for it
  - `argv` is an array of pointers to the arguments as strings
- For our program executable `hello`, we can pass a parameter like: `./hello answer 42`

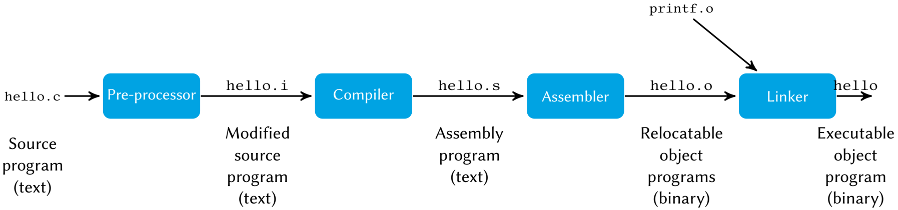
C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)
  - `argc` needed because C doesn't know how long an array is and needs to allocate for it
  - `argv` is an array of pointers to the arguments as strings
- For our program executable `hello`, we can pass a parameter like: `./hello answer 42`
  - `argc` = 3

C Syntax: main

- Command line arguments
  - To provide command-line arguments to your C program use `int main(int argc, char* argv[])` instead of `int main()`
  - `argc` is the number of strings on the command line (one for the name of the program/executable, plus one for each argument)
  - `argc` needed because C doesn't know how long an array is and needs to allocate for it
  - `argv` is an array of pointers to the arguments as strings
- For our program executable `hello`, we can pass a parameter like: `./hello answer 42`
  - `argc` = 3
  - `argv[0] = "./hello", argv[1]="answer", argv[2]="42"`

Compiling

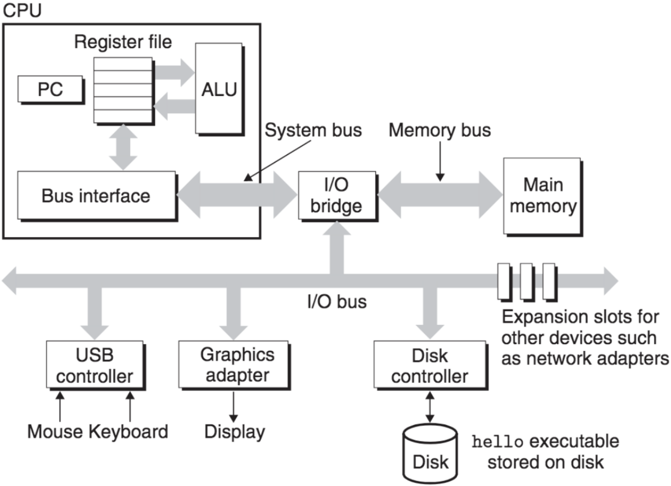

printf.o
hello.c
Pre-processor
hello.i
Compiler
hello.s
Assembler
hello.o
Linker
hello
Source program (text)
Modified source program (text)
Assembly program (text)
Relocatable object programs (binary)
Executable object program (binary)

Compiling

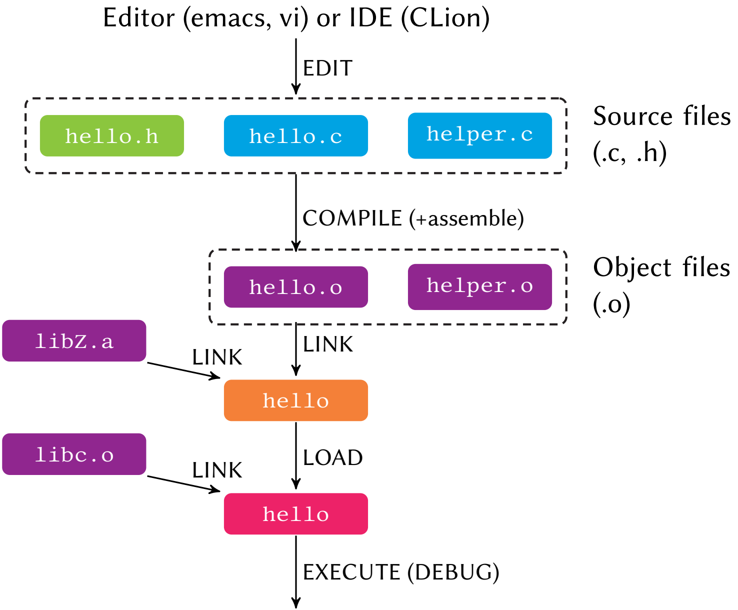
When C gets complicated...

- Errors and Exceptions
  - No `try/catch`
  - Errors are returned as error codes from a function
- Remember seg faults?
  - Preferable, because then you have a chance of figuring out what went wrong

What happens when we run our program?



C programs with multiple files


Editor (emacs, vi) or IDE (CLion)
EDIT
hello.h
hello.c
helper.c
Source files (.c, .h)
COMPILE (+assemble)
hello.o
helper.o
Object files (.o)
libZ.a
LINK
LINK
hello
libc.o
LINK
LOAD
hello
EXECUTE (DEBUG)

Shared Libraries

- Compiles functions into a shared library that other programs can link to
- Functions are defined in a header file
  - The calling program will include that header file, and link against the shared library

  library
- Can be either ***shared*** or ***static***

  shared Linked dynamically; only provide the address (linked at runtime)

  static Single file to be built; includes all files in one (big!) (linked at compile time)

Code for the next examples

```c
#include<stdio.h>
#include "goodbye.h"

int main()
{
  printf("Hello world\n");
  GoodbyeMoon();
  return 0;
}

```

Listing 28: “hello.c”

```c
// goodbye.c
#include<stdio.h>

void GoodbyeMoon() {
  printf("Good bye moon\n");
}
```

```c
// goodbye.h
extern void GoodbyeMoon();

```

Building a Static Library

```
[ahslaughter@adriennes-mbp:~]\$ ar -rc libGoodbye.a *.o

```

Listing 29: To build a static library

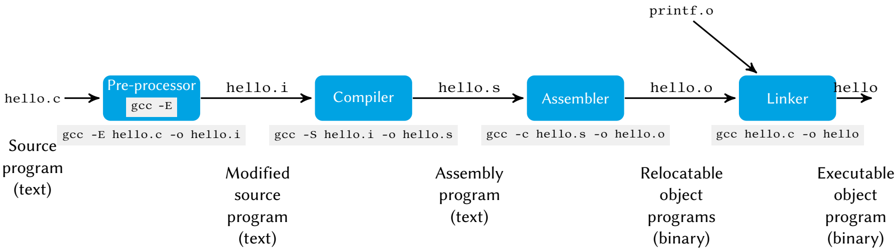
- `ar` builds an archive
- `-r` adds the given files to the archive
- `-c` creates the archive
- `libGoodbye.a` is the name of our output library/archive. (By convention, we put the ‘lib’ in front.)

```
[ahslaughter@adriennes-mbp:~]\$ gcc -g -o hello hello.o libGoodbye.a

```

Listing 30: To use a static library

Makefile

```
all: hello

hello: hello.c libGoodbye.a
  gcc -o hello hello.c libGoodbye.a

libGoodbye.a: goodbye.c goodbye.h
        gcc -c goodbye.c -o goodbye.o
        ar -rc libGoodbye.a goodbye.h *.o

run: hello
  @echo LD_LIBRARY_PATH $(LD_LIBRARY_PATH)
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./mylib ./hello

run_broken: hello
  @echo LD_LIBRARY_PATH $(LD_LIBRARY_PATH)
  ./hello

clean:
  rm *.o  hello *.a
```


printf.o
hello.c
Pre-processor
gcc -E
hello.i
Compiler
hello.s
Assembler
hello.o
Linker
hello
gcc -E hello.c -o hello.i
gcc -S hello.i -o hello.s
gcc -c hello.s -o hello.o
gcc hello.c -o hello
Source program (text)
Modified source program (text)
Assembly program (text)
Relocatable object programs (binary)
Executable object program (binary)

Building a Dynamic Library

```
[ahslaughter@adriennes-mbp:~]\$ apropos uptime

```

Code for the next examples

```c
#include<stdio.h>
#include "goodbye.h"
#include "wave.h"

int main()
{
  printf("Hello world\n");
  GoodbyeMoon();
  wave();
  return 0;
}
```

Listing 31: “hello.c”

```c
// wave.c
#include <stdio.h>

void wave() {
  printf("   \\m/    ( > < )     \\m/\n")
     ;
}
```

```c
// wave.h
extern void wave();
```

Makefile

```
all: hello

hello: hello.c libgoodbye.so libwave.so
  gcc -o hello hello.c -lgoodbye -lwave -L./mylib

libgoodbye.so: goodbye.o
  gcc -shared -o mylib/libgoodbye.so goodbye.o

goodbye.o: goodbye.c
  gcc -c -fpic -Wall -Werror goodbye.c

libwave.so: wave.o
  gcc -shared -o mylib/libwave.so wave.o

wave.o: wave.c
  gcc -c -fpic -Wall -Werror wave.c

run: hello
  @echo printing LD_LIBRARY_PATH $(LD_LIBRARY_PATH)
  export LD_LIBRARY_PATH=$(LD_LIBRARY_PATH):./mylib
  @echo printing LD_LIBRARY_PATH $(LD_LIBRARY_PATH)
  LD_LIBRARY_PATH=$(LD_LIBRARY_PATH):./mylib ./hello

```