EECS 2031 E 3.0

Software Tools

Week 12: November 28, 2018

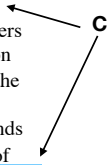
LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.

What we were supposed to cover

LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

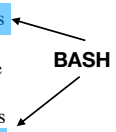
This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.



What we were supposed to cover

LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.



What we were supposed to cover

LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.

GitHub

What we were supposed to cover

LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.

GitHub

What we were supposed to cover

LE/EECS 2031 3.00 Software Tools
Tools commonly used in the software development process: the C language; shell programming; filters and pipes; version control systems and "make"; debugging and testing.

This course introduces software tools that are used for building applications and in the software development process. It covers ANSI-C (stdio, pointers, memory management, overview of ANSI-C libraries), Shell programming including Filters and pipes (shell redirection, grep, sort and uniq, tr, sed, awk, pipes in C), Version control systems and the "make" mechanism, and debugging and testing. All of the above are applied in practical programming assignments and/or small-group projects. Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes. Develop and test shell scripts of significant size. Develop and test programs written in the C programming language. Describe the memory management model of the C programming language. Use test, debug and profiling tools to check the correctness of programs.

Debugger

What we were supposed to cover

C Summary

- Programs consist of a collection of compilable units (.c files) coordinated by include files (.h files)
- Each unit is compiled separately by the compiler and linked together to produce an executable unit (a.out file)

A .c file

- Consists of a collection of functions (some local - declared static, and some global - references coordinated via .h files), and variables (again some local - declared static and some global - references coordinated via .h files)
- Each function has a signature (its collection of parameters and its return type).
- May also define local named constants and macros.

A .h file

- Synchronizes definitions across .c files (manual process, not automatic)
  - extern keyword
- Typically provides synchronization of #define'd macros and constants as well.

Variables

- C supports a number of basic types (int, char, float, double) which are tied to specific properties of the underlying hardware
- C supports more complex types including arrays, pointers, structures, unions and enumerated types.
- Typedef allows you to create 'new' types from collections of existing types.
- Within a function, a static variable is one that has one definition for the function. (Similar to the definition in Java.)

Operators

- Large collection of sophisticated operators at both the memory type level (+, -, *, /) as well as bit operators.
- No primitive string type, but a library (strings.h) provides standard set of string manipulation operators on arrays of ordered char's terminated by a null ('\0').

Pointer operators

- int *p
  - Defines p to be a pointer of type int
- p++; p- -; p+=1; etc.
  - Increments p to point to the next (or previous) thing of this type. [pointers have a type]
- p=&x;
  - Sets p to the address of x.

Pointers and arrays

- C has pointers and arrays, C cleverly and silently provides access to arrays as a pointer to the same memory locations. So it is often said that arrays in C are just pointers, but its actually more subtle than that.
  - &, *, ->, [] syntax elements
- malloc(), free(), <stdlib.h> - you are responsible for memory management in C
- Arrays are much as they are in Java and other languages
- Pointers enable recursive data structures, and passing references to variables between functions.

Functions

- Encapsulate code blocks and map arguments to parameters using call by value.
- Return type can be any type but an array.
  - For reasons of efficiency, typically return pointer to larger structures rather than copies.

C function structure

- Consists of structure elements.
- Simple statements are terminated by ;
- The block { …. } allows multiple statements to be grouped together.
- Local variable definitions must occur at the beginning of a block (ANSI requirement).
- Return value given by ‘return’ statement.

Basic C programming structures

- Selection (if, if else, switch/case)
- Repetition (while, do .. while, for)

Input/Output

- Various libraries support this (not part of the language per se).
- POSIX I/O is tied to UNIX. Generally efficient, low level.
- Stdio library provides some hardware independence, more efficient (buffering) IO.

An example

```
#include <stdio.h>
#include <stdlib.h>
#include "nextInputChar.h"

static FILE *fd;
static int lastChar = -1;

void setFile(FILE *d)
{
  fd = d;
}

int getChar()
{
  if(lastChar >= 0) {
    int temp = lastChar;
    lastChar = -1;
    return temp;
  }
  if(feof(fd))
    return -1;
  return fgetc(fd);
}

void ungetChar(int ch)
{
  if(lastChar >= 0) {
    fprintf(stderr,"ungetChar: max pushback is one character\n");
    exit(1);
  }
  if(ch < 0) {
    fprintf(stderr,"ungetChar: trying to push back a negative character\n");
    exit(1);
  }
  lastChar = ch;
}
```

Visible (and defined in the .h file)

```
void setFile(FILE *d);
int getChar();
void ungetChar(int ch);
```

nextInputChar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "nextInputChar.h"

static FILE *fd;
static int lastChar = -1;

void setFile(FILE *d)
{
  fd = d;
}

int getChar()
{
  if(lastChar >= 0) {
    int temp = lastChar;
    lastChar = -1;
    return temp;
  }
  if(feof(fd))
    return -1;
  return fgetc(fd);
}

void ungetChar(int ch)
{
  if(lastChar >= 0) {
    fprintf(stderr,"ungetChar: max pushback is one character\n");
    exit(1);
  }
  if(ch < 0) {
    fprintf(stderr,"ungetChar: trying to push back a negative character\n");
    exit(1);
  }
  lastChar = ch;
}
```

Not visible (local to this file)

```c
#include <stdio.h>
#include <stdlib.h>
#include "nextInputChar.h"

static FILE *fd;
static int lastChar = -1;

void setFile(FILE *d)
{
  fd = d;
}

int getChar()
{
  if(lastChar >= 0) {
    int temp = lastChar;
    lastChar = -1;
    return temp;
  }
  if(feof(fd))
    return -1;
  return fgetc(fd);
}

void ungetChar(int ch)
{
  if(lastChar >= 0) {
    fprintf(stderr,"ungetChar: max pushback is one character\n");
    exit(1);
  }
  if(ch < 0) {
    fprintf(stderr,"ungetChar: trying to push back a negative character\n");
    exit(1);
  }
  lastChar = ch;
}
```

An if statement

```c
#define MAX_SYMBOL_LENGTH 129
#define LEX_TOKEN_EOF         1
#define LEX_TOKEN_IDENTIFIER  2
#define LEX_TOKEN_NUMBER      3
#define LEX_TOKEN_OPERATOR    4

struct lexToken {
  int kind;
  char symbol[MAX_SYMBOL_LENGTH];
};

void freeToken(struct lexToken *token);
struct lexToken *nextToken();
void dumpToken(FILE *fd, struct lexToken *token);
struct lexToken *allocToken();
```

Token ‘kinds’

Visible functions

lexical.h

```c
#include <stdio.h>
#include <stdlib.h>
#include "lexical.h"
#include "nextInputChar.h"

#define STATE_S1  1
#define STATE_S2  2
#define STATE_S3  3
#define STATE_S4  4

/* function to allocate a new token */
struct lexToken *allocToken()
{
  struct lexToken *token;

  if((token = (struct lexToken *) malloc(sizeof(struct lexToken))) == (struct lexToken *)NULL) {
    fprintf(stderr,"nextToken: out of memory, aborting\n");
    exit(1);
  }
  token->symbol[0] = '\0';
  return token;
}

void freeToken(struct lexToken *token)
{
  (void) free(token);
}

void dumpToken(FILE *fd, struct lexToken *token)
{
  if(token == (struct lexToken *) NULL)
    fprintf(fd, "dumpToken: null token\n");
  else {
    switch(token->kind) {
    case LEX_TOKEN_EOF :
      fprintf(fd, "dumpToken: EOF token\n");
      break;
    case LEX_TOKEN_IDENTIFIER:
      fprintf(fd, "dumpToken: identifier token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_OPERATOR:
      fprintf(fd, "dumpToken: operator token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_NUMBER:
      fprintf(fd, "dumpToken: number token |%s|\n", token->symbol);
      break;
    default:
      fprintf(fd, "dumpToken: bad token type %d\n", token->kind);
    }
  }
}
```

Private defines

lexical.c (part 1)

Visible functions

```c
#include <stdio.h>
#include <stdlib.h>
#include "lexical.h"
#include "nextInputChar.h"

#define STATE_S1  1
#define STATE_S2  2
#define STATE_S3  3
#define STATE_S4  4

/* function to allocate a new token */
struct lexToken *allocToken()
{
  struct lexToken *token;

  if((token = (struct lexToken *) malloc(sizeof(struct lexToken))) == (struct lexToken *)NULL) {
    fprintf(stderr,"nextToken: out of memory, aborting\n");
    exit(1);
  }
  token->symbol[0] = '\0';
  return token;
}


void freeToken(struct lexToken *token)
{
  (void) free(token);
}

void dumpToken(FILE *fd, struct lexToken *token)
{
  if(token == (struct lexToken *) NULL)
    fprintf(fd, "dumpToken: null token\n");
  else {
    switch(token->kind) {
    case LEX_TOKEN_EOF :
      fprintf(fd, "dumpToken: EOF token\n");
      break;
    case LEX_TOKEN_IDENTIFIER:
      fprintf(fd, "dumpToken: identifier token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_OPERATOR:
      fprintf(fd, "dumpToken: operator token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_NUMBER:
      fprintf(fd, "dumpToken: number token |%s|\n", token->symbol);
      break;
    default:
      fprintf(fd, "dumpToken: bad token type %d\n", token->kind);
    }
  }
}
```

Switch statement

```c
#include <stdio.h>
#include <stdlib.h>
#include "lexical.h"
#include "nextInputChar.h"

#define STATE_S1  1
#define STATE_S2  2
#define STATE_S3  3
#define STATE_S4  4

/* function to allocate a new token */
struct lexToken *allocToken()
{
  struct lexToken *token;

  if((token = (struct lexToken *) malloc(sizeof(struct lexToken))) == (struct lexToken *)NULL) {
    fprintf(stderr,"nextToken: out of memory, aborting\n");
    exit(1);
  }
  token->symbol[0] = '\0';
  return token;
}


void freeToken(struct lexToken *token)
{
  (void) free(token);
}

void dumpToken(FILE *fd, struct lexToken *token)
{
  if(token == (struct lexToken *) NULL)
    fprintf(fd, "dumpToken: null token\n");
  else {
    switch(token->kind) {
    case LEX_TOKEN_EOF :
      fprintf(fd, "dumpToken: EOF token\n");
      break;
    case LEX_TOKEN_IDENTIFIER:
      fprintf(fd, "dumpToken: identifier token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_OPERATOR:
      fprintf(fd, "dumpToken: operator token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_NUMBER:
      fprintf(fd, "dumpToken: number token |%s|\n", token->symbol);
      break;
    default:
      fprintf(fd, "dumpToken: bad token type %d\n", token->kind);
    }
  }
}
```

Function returning pointer to a structure

```c
#include <stdio.h>
#include <stdlib.h>
#include "lexical.h"
#include "nextInputChar.h"

#define STATE_S1  1
#define STATE_S2  2
#define STATE_S3  3
#define STATE_S4  4

/* function to allocate a new token */
struct lexToken *allocToken()
{
  struct lexToken *token;

  if((token = (struct lexToken *) malloc(sizeof(struct lexToken))) == (struct lexToken *)NULL) {
    fprintf(stderr,"nextToken: out of memory, aborting\n");
    exit(1);
  }
  token->symbol[0] = '\0';
  return token;
}


void freeToken(struct lexToken *token)
{
  (void) free(token);
}

void dumpToken(FILE *fd, struct lexToken *token)
{
  if(token == (struct lexToken *) NULL)
    fprintf(fd, "dumpToken: null token\n");
  else {
    switch(token->kind) {
    case LEX_TOKEN_EOF :
      fprintf(fd, "dumpToken: EOF token\n");
      break;
    case LEX_TOKEN_IDENTIFIER:
      fprintf(fd, "dumpToken: identifier token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_OPERATOR:
      fprintf(fd, "dumpToken: operator token |%s|\n", token->symbol);
      break;
    case LEX_TOKEN_NUMBER:
      fprintf(fd, "dumpToken: number token |%s|\n", token->symbol);
      break;
    default:
      fprintf(fd, "dumpToken: bad token type %d\n", token->kind);
    }
  }
}
```

Local memory management

Summary

- An intermediate level language
  - Requires programmer to do their own memory management
  - Complex set of operators to provide low level manipulation of basic types
  - Size of types tied to hardware specifics.
  - Mechanisms to obtain addresses of types (including arrays) allowing for sophisticated memory manipulation.

Summary 2

- Has many high level language features
  - If, switch, while, do, for - constructs
  - Local and global memory structures
  - Supports compilation over multiple source files but requires programmer to coordinate functions/variables, typically through .h files.

BASH review

- Bash is
  - Written as a replacement for the Bourne shell (its the Bourne Again Shell) in the late 1980's
  - A command line interpreter (CLI)
  - A procedural programming language
  - Extremely similar to sh (many machines make sh synonymous for bash)

Simple commands

- Fall into two basic groups
  - Built in (executed within the shell itself)
  - External (separate programs that are run)
- Basic syntax is
  - command arg1 arg2 arg3 ….
  - In Unix flags are typically given to commands using dash command (e.g., -o foo.o)

```
JOB_SPEC [&]                    (( expression ))
. filename [arguments]          :
[ arg... ]                      [[ expression ]]
alias [-p] [name[=value] ... ]  bg [job_spec ...]
bind [-lpvsPVS] [-m keymap] [-f fi break [n]
builtin [shell-builtin [arg ...]] caller [EXPR]
case WORD in [PATTERN [| PATTERN]. cd [-L|-P] [dir]
command [-pVv] command [arg ...]  compgen [-abcdefgjksuv] [-o option
complete [-abcdefgjksuv] [-pr] [-o continue [n]
declare [-afFirtx] [-p] [name[=val dirs [-clpv] [+N] [-N]
disown [-h] [-ar] [jobspec ...]   echo [-neE] [arg ...]
enable [-pnds] [-a] [-f filename] eval [arg ...]
exec [-cl] [-a name] file [redirec exit [n]
export [-nf] [name[=value] ...] or false
fc [-e ename] [-nlr] [first] [last fg [job_spec]
for NAME [in WORDS ... ;] do COMMA for (( exp1; exp2; exp3 )); do COM
function NAME { COMMANDS ; } or NA getopts optstring name [arg]
hash [-lr] [-p pathname] [-dt] [na help [-s] [pattern ...]
history [-c] [-d offset] [n] or hi if COMMANDS; then COMMANDS; [ elif
jobs [-lnprs] [jobspec ...] or job kill [-s sigspec | -n signum | -si
let arg [arg ...]                 local name[=value] ...
logout                            popd [+N | -N] [-n]
printf [-v var] format [arguments] pushd [dir | +N | -N] [-n]
pwd [-LP]                         read [-ers] [-u fd] [-t timeout] [
readonly [-af] [name[=value] ...] return [n]
select NAME [in WORDS ... ;] do CO set [--abefhkmnptuvxBCHP] [-o opti
shift [n]                         shopt [-pqsu] [-o long-option] opt
source filename [arguments]       suspend [-f]
test [expr]                       time [-p] PIPELINE
times                             trap [-lp] [arg signal_spec ...]
true                              type [-afptP] name [name ...]
typeset [-afFirtx] [-p] name[=valu ulimit [-SHacdfilmnpqstuvx] [limit
umask [-p] [-S] [mode]            unalias [-a] name [name ...]
unset [-f] [-v] [name ...]        until COMMANDS; do COMMANDS; done
variables - Some variable names an wait [n]
while COMMANDS; do COMMANDS; done _{ COMMANDS ; }
```

Bash built in commands (type help)

Bash supports multiple tasks

- Only one task is in the foreground
- task & - runs the job in the background
- ^Z - suspends the current job
- jobs - lists current jobs
- fg %n - brings job n to the foreground (has control of the terminal)
- bg %n - runs job n in the background
- kill %n - kills job n

Command history

- Bash's command line is sophisticated, it has editing capabilities, a history, command completion, etc.
- history - lists your history
- Each command has a number, to re-execute it type !n
- Up and down arrow keys let you walk through the history
- Left/right arrow keys let you move through the currently selected history element (and edit)
- Hit return to execute the command
- The arrows are vi-like, emacs-like motion works too (^a,^e,^p,^n)

Variables

- The shell supports variables, two types 'environment variables' and 'shell variables'
  - Many commands use 'well known environment variables' to control their action.
- printenv - prints all environment variables
- set - prints all variables

PATH

- An environment variable
  - When you type a command junk -o foo -x bar
  - BASH first checks to see if its a built in command
    - If so, executes it
  - It then searches your path for junk that is executable by you
    - If found, executes it
    - If not found, prints an error
  - If you put a slash in the command name, then BASH just looks for the file directly
- Note: bash actually maintains a table of all executable programs to avoid having to search through this list often.

Variables

- x=2
  - Note: no spaces. None
- Want to know its value use set or
  - echo “$x” or echo $x (not echo ‘$x’)
- Variables are untyped
- Can set to null (x=)
- let command lets you manipulate variable values (but there are other ways, often better ones)
  - let “x=2+3+4”
  - echo “$x”

Exit status

- All commands has an exit status
  - 0 = good (true)
  - Anything else = bad (false)

Test

- [] or test is a command that ‘tests’ some property
- test -f foo.c
  - Tests if foo.c is a file that exists
  - $? is the last exit status

Test

- test -f test.c; echo $?
- test -f testxxx.c; echo $?

```
sh-3.2$ test -f testx.c; echo $?
1
sh-3.2$ test -f test.c; echo $?
0
sh-3.2$ test -f textxxx.c; echo $?
1
```

```
#!/bin/bash
if test -f test.c ; then
  echo "test.c exists"
else
  echo "test.c does not exist"
fi
```

Test and [ ]

- [ ] is a ‘short form’ for ‘test’
  - Requires blanks
- () executes the inner contents in a sub-shell
  - Limits side effects of the inner contents (more on this later)

Test

```
sh-3.2$ test "h" \>  "a"; echo $?
0
sh-3.2$ test "h" \<  "a"; echo $?
1
```

- Huge number of options. A few observations
  - <, > are special symbols in the shell and must be escaped \<, \>
  - test uses different operators for strings and ints

```
sh-3.2$ test 2 \> 3; echo $?
1
sh-3.2$ test 2 \> 03; echo $?
0
```

```
sh-3.2$ test 2 -gt  3; echo $?
1
sh-3.2$ test 2 -gt  03; echo $?
1
```

Let

- There are many ways of doing arithmetic expressions in bash.
- Mechanism #1 ‘let’

```
sh-3.2$ let z=3+5
sh-3.2$ echo $z
8
sh-3.2$ let z = 3 + 5
sh: let: =: syntax error: operand expected (error token is "=")
sh-3.2$ let "z=3+5"
sh-3.2$ echo $z
8
sh-3.2$ let "z = 3   +    5"
sh-3.2$ echo $z
8
sh-3.2$ let "z = z+6"
sh-3.2$ echo $z
14
```

Back tick (quote)

- In Bash, if you execute a command in `ls` then the output of the command is returned.
- You can use $(ls) as well.
  - z=$(ls) or z=`ls`

Iteration: for

- Bash as while, for and until structures
- while test; do …. done

```
#!/bin/bash
z=1
while test $z -le 10; do
  echo $z
  let "z=$z+1"
done
```

```
sh-3.2$ ./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: until

- until test ; do … done

```
#!/bin/bash
z=1
until test $z -eq 11; do
  echo $z
  let "z=$z+1"
done
```

```
sh-3.2$ ./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: for

- Not similar to similar structures in C
  - Similar structures in modern Java and (all) Python's
- for var in list; do … done

```
#!/bin/bash
z=1
for z in 1 2 3 4 5 6 7 8 9 10; do
  echo $z
done
```

```
./foo.sh
1
2
3
4
5
6
7
8
9
10
```

Iteration: for

- Iterate over all files in a directory?

```
#!/bin/bash
for z in `ls`; do
  echo $z
done
```

```
./foo.sh
foo.sh
hello
hello.c
output
```

Case statement

- case expression in case1) cmd;; case2) cmd;; esac
- case1, case2 etc are patterns
  - * matches everything
  - *.c matches c files, *.h matches h files etc.

Functions in Bash

- As Bash programs become larger it becomes prudent to break them down into smaller modules
- Two approaches in Bash
  - Have one script defined in terms of other scripts/ programs.
  - Have internal functions.

Scripts within scripts

- Given that Bash will execute commands defined outside of the script, you can clearly have one script ‘call’ another.

```
wanderereecsyorkuca:t jenkin$ cat a.sh
#!/bin/bash
echo "in a"
./b.sh
echo "back in a"
wanderereecsyorkuca:t jenkin$ cat b.sh
#!/bin/bash
echo "in b"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
in a
in b
back in a
```

Functions

- Syntax
  - function name() { … }
  - The parenthesis are optional

```
wanderereecsyorkuca:t jenkin$ cat c.sh
#!/bin/bash
function c() {
  echo "now in function c"
}

echo "about to call c"
c
echo "back"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
about to call c
now in function c
back
```

Functions: parameters

- Parameters are $1…$n
- Number is $#
- All parameters is given by $@

```
#!/bin/bash
function c {
  echo "now in function c"
  if test $# -eq 0; then
    echo "no parameters"
  else
    for z in $@; do
      echo arg $z
    done
  fi
}

echo "about to call c"
c all this and heaven too
echo "back"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
about to call c
now in function c
arg all
arg this
arg and
arg heaven
arg too
back
```

Variables in function

- By default, global
- Keyword local to identify as local
- Beware of variable hiding

```
#!/bin/bash
function c {
  echo "now in function c"
  local v1
  v1="hello world"
  v2="goodbye world"
  echo "c v1 $v1"
  echo "c v2 $v2"
}

v1="foo"
v2="bar"
echo "main v1 $v1"
echo "main v2 $v2"
echo "about to call c"
c
echo "back"
echo "main v1 $v1"
echo "main v2 $v2"
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
main v1 foo
main v2 bar
about to call c
now in function c
c v1 hello world
c v2 goodbye world
back
main v1 foo
main v2 goodbye world
```

Return values

- Functions can have a return value
  - Exit value (#?)

```
#!/bin/bash
function c {
  echo "now in function c"
  if test $# -eq 0; then
    return 0
  fi
  return 1
}

echo "going"
c
echo $?
c hello
echo $?
```

```
wanderereecsyorkuca:t jenkin$ ./c.sh
going
now in function c
0
now in function c
1
```

Arrays

- Infrequently used, but they exist in bash
- Can declare them explicitly
  - declare -a foo=(a b c)
- Or implicitly
  - foo=(a b c)
- ${foo[2]} - element 2
- ${foo[@]} - all of foo
- ${#foo[@]} - size of foo
- Arrays are 0 offset

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[0]}"
echo "${foo[1]}"
echo "${foo[2]}"
echo "${foo[3]}"
echo "${foo[@]}"
echo "${#foo[@]}"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
monday
tuesday
wednesday
thursday
monday tuesday wednesday thursday friday saturday sunday
7
```

Arrays

- Array elements can be null (so they will not print)
  - Unset foo[2]
  - foo[0]=

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[@]}"
unset foo[1]
echo "${foo[@]}"
foo[0]=
echo "${foo[@]}"
```

Arrays

- Can create arrays from arrays
- Can select parts of arrays
  - ${foo[@]:2:1}

```
#!/bin/bash
foo=(monday tuesday wednesday thursday friday saturday sunday)
echo "${foo[@]}"
foo=(${foo[@]} holiday extraday)
echo "${foo[@]}"
foo=(${foo[@]:3:2})
echo "${foo[@]}"
```

```
wanderereecsyorkuca:t jenkin$ ./a.sh
monday tuesday wednesday thursday friday saturday sunday
monday tuesday wednesday thursday friday saturday sunday holiday extraday
thursday friday
```

Bash and subshells

- It is easy (too easy) to generate subshells in bash
  - Invoke another command
  - Use a programming construct which bash implements by using subshells
- In either event you need to be aware that the (sub) shell will have its own local variables that will vanish when the sub-shell is exited

Bash and sub-shells

- If you put a command in () it is executed in a subshell.

```
#!/bin/bash
x=7
echo before $x
(echo "in parenthesis $x";x=8;echo "in parenthesis $x")
echo "after $x"
```

```
wanderereecsyorkuca:t jenkin$ ./d.sh
before 7
in parenthesis 7
in parenthesis 8
after 7
```

Bash and sub-shells

- You can cause Bash to spawn subshells whenever you pipe the output of a command

```
#!/bin/bash
x=7
echo before $x
for i in 1 2 3 4 5 6 7 8; do
  let "x=$x+1"
  echo "inside $x"
done
echo "after $x"
```

```
./d.sh
before 7
inside 8
inside 9
inside 10
inside 11
inside 12
inside 13
inside 14
inside 15
after 15
```

```
wanderereecsyorkuca:t jenkin$ cat d.sh
#!/bin/bash
x=7
echo before $x
for i in 1 2 3 4 5 6 7 8; do
  let "x=$x+1"
  echo "inside $x"
done | cat >/dev/null
echo "after $x"
```

```
./d.sh
before 7
after 7
```

Summary

- Bash - a CLI (shell) based on sh before it
  - There are other shells. Bash is free so commonly used.
- Supports standard programming language constructs, untyped variables (int, string) and arrays
- Supports functions and the ability to invoke other programs (including other bash programs)
- Utilizes value of exit (very unix-friendly) to pass a single small value integer between processes
- Variables are ‘complex’ in that different programming features can lead to the spawning/use of subshells or separate processes with their own namespace.

What’s left

- Labtest next week
  - Only on BASH (no C programming)
- Next slide is a list of things you should be able to do in Bash programming

clone.sh - outputs itself
backup.sh - make a gzip’d tar archive of your home directory, recursively
primes.sh - print out all prime numbers from 2 to the first argument
iors.sh - determine if its first argument is a string or a number
sdel.sh - safe delete. Like rm but rather moves to the file to ~/.trash don’t overwrite anything in the trash
doublespace.sh - read in stdin and write out its contents with an extra blank line between lines
rmempty.sh - remove all empty files in a directory
rmemptydir.sh - remove all empty directories in a directory
countfiles.sh - count the number of files in a directory
drawtriangle.sh - draw a triangle with n rows (n parameter)
drawsymmetrictriangle.sh - draw a triangle with n rows (n parameter) symmetric
drawtree.sh -draw a tree with n rows (parameter) and trunk
findhidden.sh - find all hidden files in a directory (recursively)
mytree.sh - print out a recursive listing of all files in a directory
rpncalc.sh - do a RPC calculator of the command line arguments
countscrips.sh - count number of shell scripts in a directory
findbig.sh - find all files in a directory (recursively) larger than a given size
rot13.sh - rot 13 stdin (move 13 spaces forward in ascii) to stdout
makeprint.sh - replace all non-printable characters in stdin and copy to stdout