2 Introduction to C

C Modules

- file translated into obj. file, which gets linked by linker to other object files and std libraries
- can refer to global variables/functions of other modules via. externs

Char input/output

- `getchar(void)`
- `putchar(c)`

`printf()` string formats

- `%.2f` floating pt to 2dp
- `%p` pointer

`scanf()`

- reads from std input
- returns number of read items
- parameters must be pointers

2 Introduction to Unix

`file`

- determines type of file
- e.g. ordinary, directory, device, 'special'

Shell environment

- at login, reads from `/etc/profile`
- gets `.bash_profile`, `.profile`

Permissions

- user, group, other
- read, write, execute
- use magical numbers (r: 4, w: 2, x: 1)

Redirections

- `a.out < data >res 2>errors`
- appending: `>>`

Shell scripts

- `#!/bin/bash`
- default search path `$PATH`
- to execute a script, `./scriptname`, otherwise if the current dir is in `$PATH` it can be executed using `scriptname`

Shell

- UNIX cmd interpreter
- reads in cmds, runs appropriate programs

I/O redirection

- when prog runs, 3 std files opened
  0 std input
  1 std output
  2 std error

Shell variables

- stored in environment of the program
- setting: `VARNAME = value`
- using: `$VARNAME`
- script arguments: `$1, $2` etc.

`if` statement

-
  ```
  if <cmd>
  then
      <cmd>
  fi
  ```

`while` loop

-
  ```
  while <condition>
  do
      <cmd>
  done
  ```

`for` loop

-
  ```
  for <condition>
  do
      <cmd>
  done
  ```

`case`

-
  ```
  case $selector in
  1) <cmd> ;;
  2) <cmd> ;;
  esac
  ```

UNIX cmds

- `test`: tests a condition, exists with true/face
  `if test $1 == "blah"`
- `sort`: sorts lines of text in a file
- `cut`: cuts selected parts of lines of text in a file, and sends result to output
- `tr`: changes or removes chars from a file
- `comm`: compares files and prints lines that exist in only one or both files
- `grep`: searches text file/output, matching each line against specified regex, and prints all lines that match
- `diff` and `sdiff`: comparing files

Cmd substitution

- arg enclosed in backquotes indicates that a command is to run, and the output used as the actual argument(s)
- `prog ‘cat argfile‘`

Subshells

- run cmds in another copy of the shell
- environment copied from parent - subshell can change environ., but it will be reverted when the subshell exits
- `tar cf  mydir | (cd *loc*; tar xf -)`

Collecting output

- `(echo data; cat filename) > output`

Arithmetic

- `expr` evaluates its args as an expression
- `let` for assignment of variables
- `let count = count +1`

Read text from shell

- `read x`: reads in line from std input, and stores as x
- ”here document”

Finding files

- `find`: starts at curr dir and searches recursively
- `locate`: prints the full path names of all files that match
- `du`: prints disk usage starting at curr dir

Strange file names

- to open file named `-x`, use `nano ./-x`

3 Pointers

- a memory address
- obtain address of variable with `&`
- create pointer to the address of `initial`
  `char initial = ’A’;`
  `char *initial_ptr = &initial;`
- `*` pointer to variable of specific type
- `**` unravels indirection
- `char msg[] = "message";`
  `char *string = &msg[0];` (or `msg`)
  $\therefore$ `msg[1] == *(string+1)`
- Iterating through a string with a pointer
  `while (*str != ’\0’) { str++ }`

Dynamic data structures

- dynamically allocate memory
- `malloc`, `realloc` etc.

Indirection operator

- declaration: pointer to specified type
  `int * ptr`
- dereferencing: dereferences the pointer to mean the content/value of the variable being pointed to

Array processing

- `int array[10]`
  `array == &array[0]`
- variables can change their values, but not their addresses
- pointer’s value is the address of another variable, $\therefore$ arithmetic ops permitted on pointer

Pointer scalars

- mathematical operations on pointers work regardless of the data type being pointer to
- ptr accesses to arrays will always move the correct number of bytes

Pass by reference

- swaps addresses of initial variables
-
```
void swap (int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Pointers to pointers

- multiple indirection
- `argv[][] == *argv[] == **argv`

`void` pointers

- no associated scalar value
- can recieve/return ptrs of any type
- `void *malloc(size_t size);`

Function pointers

- refer to 12. string handling
- allows for selection of program behavior

`NULL` pointers

- pointer with value '0'
- denotes invalid pointer, not a ptr to something at address '0'

3 Aggregate Data Structures

enums

- associates name to a value
- maps to an int
- `enum day_name {`
  `sun, mon, tue, wed, thur, fri, sat, sun`
  `}`
  maps to ints 0..7
- can then use `sun++;`

Structures

- for a collection of data items of different types
- `struct <tag> {`
  `<member-declarations>`
  `}`
- declare: `struct <tag> <identifier-list>;`
- access: `<tag>.<element-required>;`
- if a pointer to a struct is used, `->` operator is used to get an element in a struct

4 Source Code Control

Issues

- version control
- managing several versions of a program
- allows you to maintain current version whilst working on the next

Control

- checkin/checkout system
- e.g. svn, git, hg

Mercurial: hg

- distributed
  1. make copy of existing repo
  2. push changes to others
  3. pull changes from others
- `hg init`: creates repo
- `hg diff -r2 -r3`: diff b/w revision 2 and 3
- `hg revert -r2 code.c`: revert file to r2
- `hg push/pull/clone <repo>`
- repo: another dir/URL to a remote repo

4 `make`

- program can use many `.c`, `.h` files that require compiling
- time consuming to compile lots of files separately
- object file: machine language, but not yet linked with other parts of the program
- several `.c`, `.o` files can be combined to give an executable program via. linkage
- after changing one `.c` file, you need to recompile affected file and relink ∴ `make` is sexy

Rules

-
```
prog.o: prog.c prog.h     dependencies
    gcc -c prog.c         action
```
- `<target>`: name of file to be made
- `<1+ dependencies>`: files the target file depends on
- `<action>`: shell cmd that creates target
- default rules are the bomb
- can combine rules when targets have common dependencies and actions
- can create rules without dependences
```
clean:
    rm *o
```

`make` variables

- assignments: `variable_name = value`
- use: `$(variable_name)`

Predefined variables

- `CC`: default C compiler
- `CFLAGS`: flags passed to the C compiler

Libraries

- gives you the ability to store the object code versions of the functions in one place and have them linked into your program
- stdlib automatically searched when prog is linked
- use functions from other libraries using `-l` flag when linking
- the C compiler will search for the library in standard directories: `/lib, /usr/lib`
- create own library using `ar`, which makes library mylib.a and will contain specified .o files
  `ar c mylib.a readit.o util.o`
- can then use created library when compiling
  `gcc myprog.o mylib.a -o myprog`

5 Memory Management

Memory areas

- code: program instructions
- global/static: global/static variables
- stack: local variables, function arguments, return addresses, temporary storage
- heap: dynamically allocated memory