e1_cheatsheet

Linux Commands

man command : Show the manual of command
ls : List all files in the directory
  -a : List includes hidden files
  -l : List include user read/write/execute permission
cat : Show the content of the file

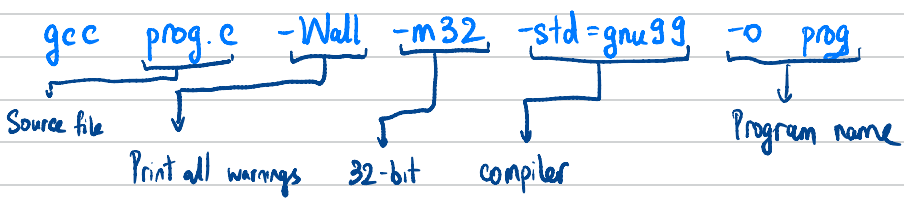
echo word : Print word
ssh : Connect to an another machine

Absolute filepath : Starting from home
  ex : /p/course/cs354/file.txt
Relative filepath : Starting from current directory.
  ex : ./cs354/file.txt
  when currently in /p/course

Andrew File System (afs) Command

cd dir : Change directory to dir
mkdir dir : Create new directory name dir
  -P : If there is no directory as mentioned, force create it.
rmdir dir : Remove/delete directory name dir
mv file1 location1 : Move file to new location
  file1 location1/file2 : Move file to new location and rename
cp : Copy files in local machine
scp : Copy files through machines
  -r : Copy folder through machines
pwd : Print working directory

GCC



gcc -E : Preprocessing Stage (.i)
↳ Remove comments, expand macros (#define), include file
↳ Output intermediate file
gcc -S : Compiling Stage (.s)
↳ Translate code to assembly language (file)
gcc -c : Assembling Stage (.o)
↳ Convert assembly into machine code
gcc : Linking stage
↳ Link .o files with other predefined object files
↳ Create EOF (executable obj files)
gcc -g : Enable debugging

Vim
vim file : Open file in editor.
If file doesn't exist then create and open

ESC : Normal mode
:q! : Quit file
:w : Save file
i : Text edit mode / Insert text
x : Delete 1 character
dd : Delete the whole line
/<char> : Find <char> in file

C Programming
1) if (c = 0) ⇔ c = 0; if (c)
0, NULL, '\0' return false

2) Variables:
Endianness: Bytes ordering
- Little endian : ↑ high / low
- Big endian : ↓ low / high

Explicit casting : (type name) expression

Implicit casting :
1) Arguments are forced castings
2) Casting hierarchy : long double > double > float > int
char and short are convert to int

i++ ⟶ do something ; i=i+1
++i ⟶ i=i+1 ; do something

3) Pointers

Indirect access to memory and functions

int *p; Declare a pointer p1 to int

int *p1, *p2; Declare 2 pointers p1 and p2 to int
int *p1, p2; Declare a pointer p1 and an int p2

&i : Address of i
*p : Dereference p, access pointee of p

Arrays

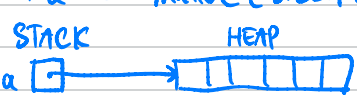
Arrays (Stack Allocated Array)

int a[]; // Stack allocated array
a is not a variable, but an identifier

If used as source operand, provide the address (at a[0])

Arrays (Heap Allocated)

int *a = malloc(sizeof(int) * N);



based on type: int * a ⇒ +4 bytes
char * a ⇒ +1 bytes

Address Arithmetic: $a[i] = *(a+i)$

int *y = x+n // where x = a and a has size > n

Then: → *y - *x = size_of_type * n

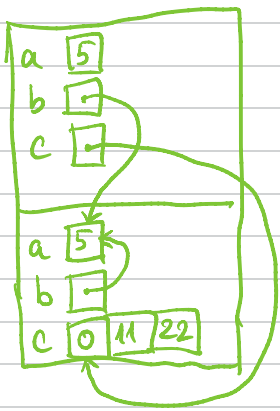
→ y - x = n

Passing Address

Scalar : Copies value of its scalar args

Pointers : Copies address variable

Arrays : Copies array address variable



malloc (size_of_bytes)

↳ Reserve blocks of heap in memory

↳ Returns a generic pointer to the memory

free (void * ptr)

↳ Free the memory in heap

sizeof () → Return # bytes of a type

↳ double : 8

↳ char : 1

↳ int : 4

Pointers Caveat:

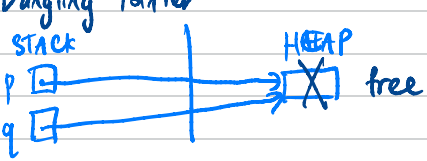
1) Uninitialized

int *p; *p = 11; → Intermittent error

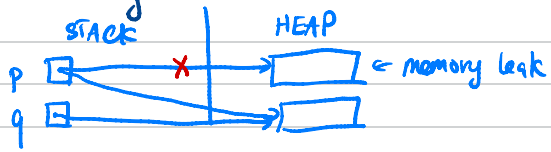
2) Null pointer

int *q; *q = 12; → Segmentation fault

3) Dangling Pointer



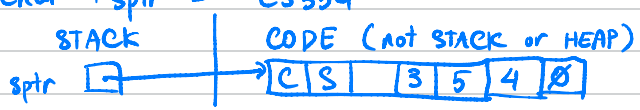
4) Memory leak



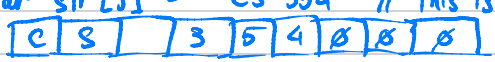
C String / string.h

String literal : A constant source code string allocated prior to execution in the CODE segment

```
char *sptr = "CS354"
```




```
char str[9] = "CS 354" // This is SAA, and the chars are copied
```



↳ strlen = 7 ; sizeof = 9

Assignment :

✓ `sptr = "abc";` // Assigning pointers
✓ `strcpy (str, "abc");`

✗ `str = "abc";` // C doesn't allow this
✗ `strcpy (str, "very very long string");` // Buffer overflow
✗ `strcpy (sptr, "...");` // Cannot change in CODE segment
// Seg fault

string.h

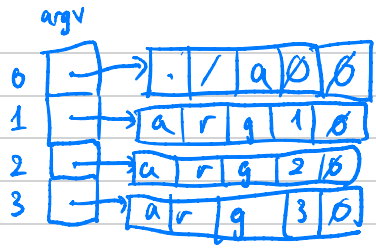
`int strlen(const char *)` : Return the length (from first index to first '\0')
`int strcomp(const char *, const char *)` : Compare 2 strings
`char * strcpy (char *, const char *)` : Copies
`char * strcat (char *, const char *)` : Concatenate

Command Line Arguments

CLAs

./a arg1 arg2 arg3 arg4

program args



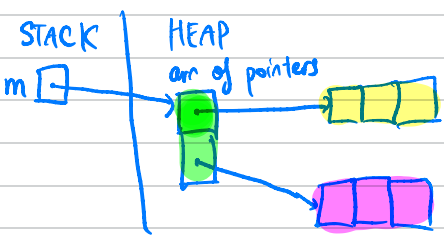
int argc : # CLAs

char ** argv : Argument vector, array of CLAs

2D Arrays

1) 2D Arrays On Heap

```
int ** m;
m = malloc(sizeof(int*) * n);
for i = 0 -> n-1 :
    *(m+i) = malloc(sizeof(int) * n)
```



Free the component in reverse order of malloc

Address Arithmetic :

$m[i][j] \equiv (*(m+i))[j] \equiv *(m[i]+j) \equiv *(*(m+i)+j)$

2) Stack :

```
int m[][];
```

2D Arrays allocated in memory are laid out in row-major order, as a contiguous block of memory with 1 row after another

Address Arithmetic :

$m[i][j] \equiv *(*(m+i)+j)$

$\equiv *(m + cols*i + j)$

3) Caveats

Arrays have no bound checking // Buffer Flow
Arrays cannot be return type, use pointer instead
Not all 2D arrs are alike (Test: check if m[i][j] is valid)
An array argument must match it types
↳ SAA requires all but the first dim matches

Structures

Define	struct name { } ;	typedef struct { } name ;
Declare	struct name Var ;	name Var ;

A structure identifier ⌐ read the entire struct (as source op)
write the entire struct (as dest op)

Structure can contain other structs or arrays as deep as we can
Arrays can have struct for elements

Structures are passed-by-value to function, which copies the entire struct (slow)
Solution: Pointers (enable linked structure)

(*m).data ≡ m→data
*m.data ≡ *(m.data)

I/O

Format Specifiers

%d : Signed int %i : Unsigned integer
%f : Float %.*f : Precision
%s : String %.*s : Print at most characters
%c : Char
%p : Pointer (hex) %%: A normal %

1) Standard I/O : Keyboard + Terminal

getchar : Get 1 char
gets : Get 1 string (Not formatted)

int scanf (const char * format, &v1, &v2)
↳ Read the FORMATTED string
↳ Return # inputs stored, or EOF if error / end of file occurs before inputs

putchar : Print 1 char
puts : Print 1 string (Not formatted)

int printf (const char * format, v1, v2)
↳ Write formatted output
↳ Return # char written, or negative if error
* Use \n to flush buffer

2) String I/O

int sscanf (const char * str, const char * format, &v1, &v2, ...)
↳ Read formatted string from str
↳ Return # chars read, or negative if error

int sprintf (char * str, const char * format, v1, v2, ...)
↳ Write formatted string to str
↳ Return # chars written, or negative if error

3) File I/O

fgetc : Read 1 char at a time

fgets : Read 1 string, terminate with a newline char or EOF

int fscanf (FILE * stream, const char * format, &v1, &v2, ...)

↳ Read formatted string from stream

↳ Return # inputs stored, or EOF if error / end of file

fputc : Write 1 char

fputs : Write 1 string

int fprintf (FILE * stream, const char * format, v1, v2, ...)

↳ Write formatted output to stream

↳ Return # chars written

stdin - Console keyboard

stdout - Console terminal window

stderr - Console terminal window, second stream for errors.

FILE * fopen (const char * filename, const char * mode)

↳ Open the specified filename with specified mode

↳ Return FILE pointer, or NULL if there's an access problem

int fclose (FILE * stream)

↳ FLUSH the output buffer and then close the stream

↳ Return 0, or EOF if error

Memory

- Abstraction: Manage complexity by focusing on relevant detail

Three Faces of Memory:

1) Process View = Virtual Memory
   Goal: Provide simple view
   Virtual Address Space: Illusion that each process has its own contiguous blocks of memory

2) System View = Illusionist
   Goal: Make memory SHAREABLE & SECURE
   Pages: 4KB of storage fixed-size blocks
   Page table: An OS Data structure that virtual page to physical page to keep process from interfering each other

3) Hardware View = Physical Memory
   Goal: Keep CPU busy
   Physical Address Space: multi-level addr hierarchy access data in memory closest to CPU

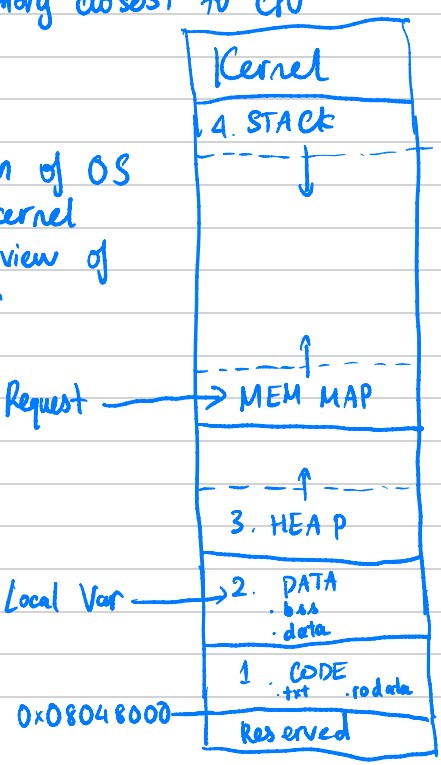
Virtual Address Space

Kernel: Memory resident portion of OS
User Process: Process that are not kernel
Every user process has a simple view of memory and make coding easier

DLL, File, I/O, Main Memory, Request → MEM MAP

User has 2GB, OS has 2GB

Global & Static Local Var → 2. DATA .bss .data



Abstract Memory Model

1) CODE

Contains : Program code in binary-machine code
- .text : machine code
- .rodata : string literal

Lifetime : Entire program's execution
Initialization : By loader from exec obj file BEFORE execution
Access : Read only

2) DATA :

Contains : Global & Static Local Variable:
- .bss : Uninitialized or initialized to 0
- .data : initialized to non-zero values

Lifetime : Entire program's execution
Initialization : By loader from exec obj file
Access : Read + Write

3) HEAP (Free Store)

Contains : Memory allocated and freed during execution
Lifetime : Managed by programmer through malloc/calloc/alloc and free
Initialization : None by default
Access : Read + Write

4) STACK (Auto Store)

Contains : Memory in stack frame, auto-allocate and free
Stack frame : Non-static local var, func params, temp var of loop, etc.
Lifetime : From declaration until end of scope
Initialization : None by default
Access : Read + Write

* Note :
- Memory allocations ALWAYS initialized : CODE, DATA
- Can write : DATA, HEAP, STACK
- malloc : HEAP
- Not user process : Kernel
- Global / Static Local : DATA
- String literal : CODE

Shadowing
When local variable block access to global variable

C stdlib.h
- Contains a collection of ~25 common C functions that do
  - + Conversion
  - + Search
  - + Execution Flow
  - + Sorting
  - + Math
  - + Randomize

Heap Allocator
void * malloc (size_t size):
Allocate and return generic pointer to block of heap memory of size, or return NULL if error

void *calloc (size_t n, size_t size)
Allocate, clear to 0, and return a block of heap of size * n, or return NULL if error

void * realloc (void * ptr, size_t size)
Reallocate to size bytes a previously allocated block of heap memory pointed to by ptr, or return NULL if error

void free (void * ptr)
Free the heap memory. If ptr NULL then do nothing.
// Double free cause undefined behavior