COMP 206 Midterm 2 Crib Sheet

C's Standard IO library <stdio.h>

printf(char *format), **fprintf**(FILE *f, char* format): print formatted otpt
scanf(char *format), **fscanf**(FILE *f, char* format): read formatted inpt
putchar(int c), **fputc**(int c, FILE *f): print a single character
getchar(), **fgetc(**FILE *F): read a single character (and returns it)
puts(char *string), **fputs**(char*string, FILE *f): print a string
gets(char *string) **fgets**(char * string, int bytes, FILE *f), read a string

String Functions from <string.h>:

char * **strcpy** (char * destination, const char * source);
char * **strncpy**(char * destination, const char * source, size_t num);
char * **strcat** (char * destination, const char * source);
char * **strncat** (char * destination, const char * source, size_t num);
int **strcmp** (const char * str1, const char * str2); //0 if same
size_t **strlen** (const char * str);
char * **strstr** (char * str, const char * toFind); //NULL ptr if not found

Dynamic Memory and Process Organization

Memory organization

`sizeof(var_type)` or `sizeof(var)` returns the number of bytes associated with its argument, sizeof(array) will return total number of bytes (# of bytes of each element * array length), but will return the size of the pointer if passed into a function

Asking for Heap Memory: common error is trying to return a pointer to stack memory.

Requesting for N bytes of heap memory (all values 'NULL'):

```
void *malloc (int numberOfBytes);
Example: int *a = (int *) malloc(sizeof(int)*40);
```

Returns a void pointer which **must be cast before it is used**

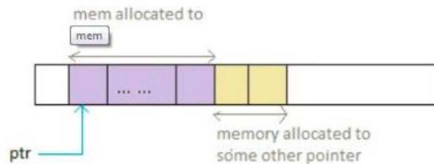
Requesting for an array of N elements with size bytes, initializes all values to zero:

```
void *calloc (int N, int size)
Example: int *a = (int *) calloc(40, sizeof(int));
```

Expanding an existing memory space

```
void *realloc(void *ptr, size_t newTotalSize)
ptr = (int*) realloc(ptr, 10*sizeOf(int));
```

When re-allocating: heap might not contain enough space, data from old space in heap might be copied over to some other place in memory (watch your pointers)



Remember to 'free' dynamically allocated memory when you're done using it:
free(void *ptr); ptr =NULL;

Malloc rules:

`TYPE1 var = (TYPE2)malloc(sizeof(TYPE3)*number);`

1. TYPE1 matches TYPE2, both pointers
2. TYPE3 is de-reference form of TYPE2

Debugging and Multi-file C programs

3 options:

1. **`printf everything`** (%s - string, %c - char %f-float, %d - integer)
2. **Using preprocessor** (interacts w/ # lines only),

```
    #ifdef DEBUG
      #define debug(x,y) printf(x,y)
    #else
      #define debug(x,y)
    #endif
 In main:
        int a = 3;
        debug("The value of a is %d.\n", a );
    //Will do nothing if DEBUG is not defined
```

- `#include <FILE>` or `"FILE"` imports libraries or other .c files, and dumps all that code into your code
- `#define` allows definition of a SYMBOL to a VALUE:
    - Ex: `#define TRUE 1; //Everywhere in the program there is a TRUE, it will be replaced by a 1`
- `#ifdef` SYMBOL returns true if SYMBOL has already been defined (`#ifndef` returns opposite)
- `#undef` SYMBOL removes or 'undefines' the SYMBOL
- To define a symbol at the command line, i.e., interact with the pre-compiler use -D(Symbol)
    - Ex: **`$gcc -DDEBUG program.c`**

C pipeline

- Source code (.c) to preprocessed code (.i) via the preprocessor, pre-processed code to Assembly code (.s) via the C Compiler, Assembly code to Object code (.o) via the Assembler, Object code to Executable code (.exe) via the Linker

gcc output control (Note: -o renames default output file**)**

- `$ gcc -E code.c -o pp_code.i`
    - Only pre-processes your code (i.e. looks only at # lines) to a ".i" file
- `$ gcc -S code.c -o assem_code.s`
    - To see assembly code (.s file)
- `$ gcc -c code.c -o object_code.o`
    - Compiles and/or assembles but does not link, but the .o file can than be linked to other files

3. **GDB Debugger,** exists everywhere the gcc does

Allows you to 'step' through your code through the command line, capture terminal output, etc, and effectively takes the place of your OS. To run:

```
$ gcc -g -o helloworld helloworld.c
$ gdb helloworld
```

Will launch program, show prompt (gdb) instead of '$'

Some GDB commands: `run <arguments>` → runs the program normally, with arguments, **`break <line # or function name>`** → stops the program at that point, **list** → lists the source code, **`backtrace`** → shows stack of frames, starting from current frame, **`print <var>`** → prints the value stored in var, **`disp<var>`** → will display var after every step, **`next`** → go to next line, doesn't dive into functions, **`nexti`** → same as next, but skips to after function stops, **`step`** → **s**ame as next but goes into functions

Multiple-file projects and Libraries: example

fcns.h

```
#define FIRST 5
int fn1(int a);
int fn2(int b);
```

fcns.c

```
#include "fcns.h"
int global_a = FIRST;
int fn1(int a) {return 5;}
int fn2(int b) {return global_a + b;}
```

main.c

```
#include "fcns.h"
#include <stdio.h>
extern int global_a;
int main() {
        int c = FIRST;
        global_a += fn1(c);
        printf("%d", fn2(c));
        return 0
}
```

`#include "fcns.h"` `(in main.c)` tells C before it compiles that the swap function actually exists. Note that the swap.h file contains just the header, which includes name and types, but no implementation

Object files

Compile a program ahead of time into a ".o" file, to be used by another program downstream

One-to-one correlation: one .c file (generally) produces one .o file

- **To use, use -c flag:**
- **`$ gcc -c swap.c`**
- **`$ gcc main.c swap.o`**
- Original swap.c can be deleted now if no longer needed, BUT need to recompile if swap.c if edited for updated swap.o file

Libraries → linking different functionalities together**: #include <library>:** ctype.h, **stldib.h**, **stio.h,** math.h, string.h, time.h

Two types of libraries:

Static (.a) : Copied by the linker into an executable, becomes part of the code. Links during compilation
`$ gcc -c swap.c /`
`$ ar rcs libswap.a swap.o`
`$ gcc main.c libswap.a`
	Shared/Dynamic (.so): remains a separate file from the program that called it, features are loaded dynamically as needed. Links during run-time
	`$ gcc -shared -fpic -o libswap.so swap.c`
	`$ gcc main.c -lswap -L`
	`$ export LD_LIBRARY_PATH= ${LD_LIBRARY_PATH}:.`

Note: the export line sets LD_LIBRARY_PATH, which is an environment variable that tells the program where the '.so' file is, for the rest of the terminal session,

- **`ldd`** - prints shared object dependencies
- **`nm`** `<file.o>` lists symbols of file

C Structs, Advanced Allocation

```
struct OPTIONAL_NAME {
        FIELDS;
} OPTIONAL_VAR_NAME;
```

COURSE Example: note that cs206 is **not** a pointer. Be careful of the 'padding' between fields of the same struct

cs206

int	numberOfStudents
100 chars	nameOfProfessor
100 chars	buildingName
int	roomNumber

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct COURSE { //optional if a var has been defined after '}'
        unsigned int numberOfStudent;
        char nameProfessor[100];
        char buildingName[100];
        unsigned int roomNumber;
} cs250; //optional; you can declare a variable immediately...

int main () {
        struct COURSE cs206; //Or declare it in main
        cs206.numberOfStudent = 60;
        strcpy(cs250.nameProfessor, "Alex");
        //You can also initialize values as you declare them
        struct COURSE cs189 = {100, "Ruths", "Stewart", 114};
}
```

typedef just simplifies your code, so you don't have to type out 'struct' every time

```c
typedef struct COURSE {
        int numberOfStudents; ...
} MYCOURSE; //This is a new type now, not a variable
```

Declaring in main: `MYCOURSE cs206;`

An array of structs: you can declare it after the struct or in your main

```c
struct STUDENT {  FIELDS;
        } s_arr[50];
In main:
s_arr[2].age = 20; //OR...
struct STUDENT s_arr2[50];
```

Pointers to structs, mallocing for structs

```c
struct STUDENT *p;
p = (struct STUDENT *) malloc (sizeof(struct STUDENT))
p -> age = 18;//Equivalent to (*p).age = 18; i.e., it de-references and accesses the field all at once
```

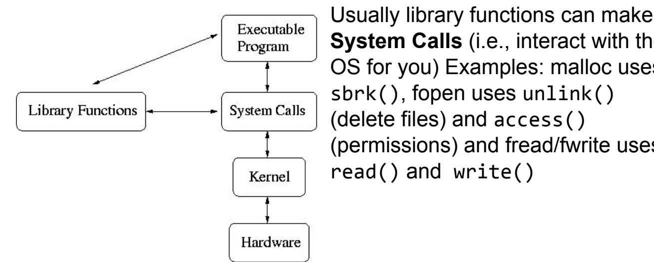
Linked list, insertion sort example from sample midterm

```c
LIST_NODE*insert_one_sorted(LIST_NODE * list_so_far, int new_val_to_sort) {

        LIST_NODE *new = (LIST_NODE*)malloc( sizeof(LIST_NODE));
        new->val = new_val_to_sort;

        //When list is empty or the value to sort happens to be the smallest
        if( list_so_far == NULL || list_so_far->val >= new_val_to_sort ){
                new->next = list_so_far;
                return new;
        }

        LIST_NODE* pos=list_so_far;
        while( pos->val < new_val_to_sort ){
                //When we get to the end, or if the next value is larger
                //than the value to be sorted
                if( pos->next == NULL || pos->next->val >= new_val_to_sort ){
                        new->next = pos->next;
                        pos->next = new;
                        return list_so_far;
                }
        pos = pos->next;
        }
}
```

Pointers to Functions, System Calls, Buffering

Pointers to functions: Note: int *fn() is a function that returns an int pointer, int(*fn) () is a pointer to a function

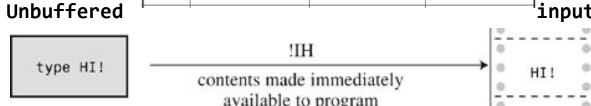
```c
void add(int a, int b) { printf("%d\n", (a + b));}
int main () {
        void (*p) (int, int) = &add;
        int a = (*p)(3, 4);
        int b = p(3, 4); } //Both acceptable syntax
```



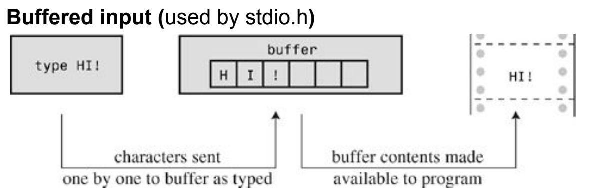
Usually library functions can make **System Calls** (i.e., interact with the OS for you) Examples: malloc uses `sbrk()`, fopen uses `unlink()` (delete files) and `access()` (permissions) and fread/fwrite uses `read()` and `write()`

	Buffered-Formatted	Buffered-Unformatted	Unbuffered
C	fprintf() fscanf()	fread()/fgets() fwrite()/fputs()	read()/getc() write()/putc()

Unbuffered input:



Buffered input (used by stdio.h)



Make, CMake

`make` allows us to specify dependencies, so that only the files that have been changed and files that depend on it get rebuilt instead of rebuilding every file in the project. To use `make`, we must put all our specifications in a file named "`Makefile`"

The syntax:

```
{target}: {dependencies}
        {gcc line, etc.}
```

Macros: `name=text_string`

- Example: `SRCS=foo.c bar.c`
- Use `${name}` to access info in variables
- `OBJS=${SRCS:.c=.o}` converts all file names to .o files, i.e, same as `OBJS=foo.o bar.o`

Example Makefiles:

```make
SRCS=foo.c bar.c
OBJS=foo.o bar.o barbar.o
CFLAGS=-Wall -ansi
LDFLAGS=-lm -lmylib
INCDIR=-I/home/ericb/include
LIBDIR=-L/home/ericb/lib

all: ${OBJS}
  ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
foo: ${OBJS}
  ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
clean:
    /bin/rm -f ${OBJS}
install:  foo
    /bin/cp -f foo /usr/local/bin

.c.o:
    ${CC} ${CFLAGS} ${INCDIR} -c $<
```

- Last 2 lines tells `make` how to compile .c files into .o files
- $< is a variable set to the current dependency

```make
tutorial: f1.o f2.o
    gcc -o tutorial f1.o f2.o
f1.o: f1.c f2.h
    gcc -c f1.c
f2.o: f2.c f1.h
    gcc -c f2.c
```

Example of a Makefile of a program named tutorial that contains f1.c, f2.c, f1.h, and f2.h, where f1.c includes f2.h and f2.c includes f1.h

Auto-configuration tools automatically generates Makefiles for more repetitive compilation tasks

Popular tool is **CMake,** where instructions are contained in "CMakeLists.txt"

- Where your code is stored and where your code is compiled to are separate folders--called an "out of source" build
- Instead of typing make in your source folder, the general procedure is the following, typed into the terminal:

`$ mkdir build` makes a folder that should only contain compilation outputs/files, **`$ cd build`** moves you to that folder, **`$cmake ..`** runs `cmake` on the parent folder (1 level outside of build folder), which should contain the CMakeLists.txt, this step auto-generates everything needed for `make` to compile files into the build folder, **but this step itself doesn't actually compile your code**

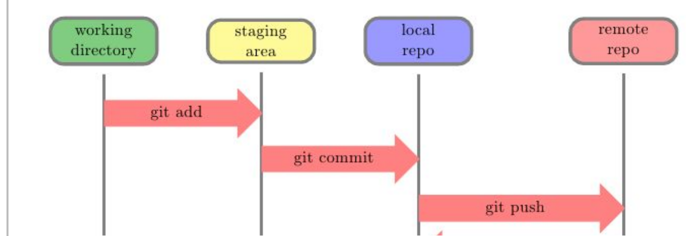
Typing **make** inside the build folder then compiles your code based on how your cmake set it up

Version Control

Version control lets you revert to previous versions and maintain branches

- **Local** version control works with your own, local directory (an example: revisions control system (RCS)).
- **Client-Server vc**, used a lot by a companies, for example, shares code via a central system, such as a company server. An example: SVN (subversion)
- **Distributed** vc, like git, allows for more flexible workflow; developers have their personal repositories and control how it's shared

Git diagram:



git clone	copy a repository into local directory
git add	add files to queue for next commit (to be done on any local changes)
git commit	commit queued files
git push	push commit(s) to remote repository (default: same one you cloned from)
git pull	fetch changes from remote repository (default: same one you cloned from)
git init	start a new repository from local files
.gitignore	ignore specific files by adding them here