C Basics

```
void* malloc( int num_bytes );
void* calloc( size_t num, size_t size );
void *realloc( void *ptr, size_t new_size );
void free( void* p );
int atoi( const char *s ); /* Convert string to int */
int printf( const char * fmt, ... ); /* %d int, %lu unsigned long, %f double,  %s string, \n newline, ...=args */
void* memcpy( void *destination, const void *source, size_t num_bytes ); /* Returns destination */
void *memset( void *mem, int value, size_t num_bytes ); /* Set memory to 0 */
size_t strlen( const char * string ); /* returns length of null-terminated string, not counting the terminator */
int strncmp ( const char * str1, const char * str2, size_t max_size ); /* return 0 if strings equal */
```

Files

```
int open( const char *filename, int flags );  /* Returns a file descriptor if successful, -1 on error */
ssize_t read( int file_descriptor, void *buffer, size_t count ); /* Returns number of bytes read */
ssize_t write( int file_descriptor, const void *buffer, size_t count ); /* Returns number of bytes written */
int rename( const char *old_filename, const char *new_filename ); /* Returns 0 on success */
int close( int file_descriptor );
FILE* fopen( const char *filename, const char *mode );
int fclose( FILE* f );
size_t fread(void *restrict ptr, size_t size, size_t nmemb, FILE *restrict stream);
size_t fwrite(const void *restrict ptr, size_t size, size_t nmemb, FILE *restrict stream);
int fprintf( FILE* f, const char* format, ... ); /* Arguments go in the ...  */
int fscanf( FILE* f, const char* format, ... ); /* Arguments go in the ...  */
int fileno( FILE* f ); /* Convert FILE* to file descriptor */
int fseek ( FILE * f, long int offset, int origin ); /*SEEK_SET from file start. SEEK_CUR from current loc.*/
int remove( const char* filename );

void* mmap( void* address, size_t length, int protection, int flag, int fd, off_t offset ); /* address = NULL */
int mprotect( void* address, size_t length, int prot ); /* PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXECUTE */
int msync( void* address, size_t length, int flags ); /* Use MS_SYNC for flags */
int munmap( void* address, size_t length );
```

When opening a file with `fopen()`, the options are:

Mode	Meaning
`r`	Open the file read-only.
`w`	Open the file for writing (create if needed)
`a`	Open the file for appending (create if needed)
`r+`	Open the file for reading, from the start
`w+`	Open the file for writing (overwrite)
`a+`	Open the file for reading and writing, append if exists

For `open()` the following flags may be used for the `flags` parameter, and can be combined with `|` (bitwise OR):

Flag	Meaning
`O_RDONLY`	Open the file read-only
`O_WRONLY`	Open the file write-only
`O_RDWR`	Open the file for both reading and writing
`O_APPEND`	Append information to the end of the file
`O_TRUNC`	Initially clear all data from the file
`O_CREAT`	Create the file
`O_EXCL`	If used with `O_CREAT`, the caller MUST create the file; if the file exists it will fail

Processes, Threads, Concurrency

```
pid_t fork( );
pid_t wait( int* status );
pid_t waitpid( pid_t pid, int *status, int options ); /* 0 for options fine */
void exit( int status );

pthread_create( pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)( void * ), void *arg );
pthread_join( pthread_t thread, void **returnValue );
pthread_detach( pthread_t thread );
pthread_cancel( pthread_t thread );
pthread_testcancel( ); /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_setcanceltype( int type, int *oldtype ); /* PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS */
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */
pthread_exit( void *value );

pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes );
pthread_mutex_lock( pthread_mutex_t *mutex );
pthread_mutex_trylock( pthread_mutex_t *mutex ); /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex );
pthread_mutex_destroy( pthread_mutex_t *mutex );
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr );
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock );
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock );
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock );
pthread_rwlock_trywrlock( pthread_rwlock_t * rwlock );
pthread_rwlock_unlock( pthread_rwlock_t * rwlock );
pthread_rwlock_destroy( pthread_rwlock_t * rwlock );

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore );
sem_wait( sem_t* semaphore );
sem_trywait( semt_t* semaphore );
sem_post( sem_t* semaphore );
```