C Reference

#c #paper-note

Functions

```
int main() {
    return 0
}

char * seals(char * clubs) {
    printf("bonk seals with %s", clubs);

    return clubs;
}
```

Variables

```
int x = 3;
char str[] = "string literal";
```

Pointer Manipulation

Pointers hold the address of a variable, not the value of the variable.

```
int x = 3;
int * xPtr = &x; //address of x
// dereferencing
int y = *xPtr; // value at xPtr
```

Pointer Arithmetic

Pointer arithmetic is performing operations on pointers. Only allowed operations are:

- adding or subtracting integers to or from pointers
  - `xPtr += 3;`
  - `xPtr -= 3;`
- subtract pointers from each other
  - `long zPtr = xPtr - yPtr;`
- compare pointers to each other
  - `int isMore = xPtr > yPtr;`

Arrays

Arrays are pointers to their first element and have allocated memory for the values after that.

```
float fArr[5];
float multiDim[5][5];
multiDim[3][1] = 5.0;

int * iArr = malloc(sizeof(int) * 2);
iArr[1] = 3;

int litArr[] = {1, 2, 3};
```

POSIX Threads (`pthreads`)

Processes can have threads. Threads run code.

```
#include <pthread.h>

int pthread_create(
    // address of thread
   pthread_t * restrict thread,
    // options
   const pthread_attr_t * restrict attr,
    // function
   typeof(void *(void *)) * start_routine,
    // argument (void * as general type)
   void * restrict arg
   );

int pthread_join(
    // actual thread (not address)
    pthread_t thread,
    // pointer to return value, void ** because pthread_exit returns void *
    void ** retval
```

```
);


void pthread_exit(void * retval /*pointer to return*/);
```

Mutexes
Mutexes are used to protect critical sections by providing "Mutual Exclusion".

```
pthread_mutex_t lock;

pthread_mutex_init(&lock);

pthread_mutex_lock(&lock);

printf("critical section\n");

pthread_mutex_unlock(&lock);
```

Printing `printf()`
Prints strings and arguments with specifiers.
`%[flags][width][.precision][length]specifier`

- specifier
  - %d: for printing integers
  - %f: for printing floating-point numbers
  - %c: for printing characters
  - %s: for printing strings
- Length
  - h: With short int and unsigned short int
  - l: With long int and unsigned long int.
  - L: With long double

Macros

```
#include <stdio.h> // printf, scanf
#include <stdlib> // atoi, malloc
#include <pthread.h> // pthreads and mutexes
#include <unistd.h> // sleep
#include <stdbool.h> // bool
#include <string.h> // strlen, strcpy

#define NUM_THREADS 3
```

If Else Statements

```
int x = 3;
if (x < 2) {
    printf("cry\n");
} else if (x > 3) {
    printf("boohoo\n");
} else {
    printf("three!\n");
}
```

For Loop

```
int arrLen = 3;
int arr[] = {1,2,3};

for (int i = 0; i < arrLen; i++) {
    printf("%d\n", arr[i]);
}
```

While Loop

```
int counter = 4;

while (counter > 0) {
    printf("%i\n", counter);
    counter--;
}
```

String to Integer Conversion `atoi()`

```
char * str = "12345";
int num = atoi(str);
num--;
printf("%i\n", num); \\12344
```

Type Definitions `typedef`

```
typedef struct town {
    int population;
    pthread_mutex_t * reaper;
} town_t;

typedef struct _move {
    town_t * oldTown;
    town_t * newTown;
} move_t;

town_t * town = (town_t *) townArg;
```

Allocating and Deallocating Memory `malloc`, `free`

`malloc` is used to allocate space in the heap. It returns a `void *` pointing to the starting address of the allocation. `free` takes a pointer and frees memory allocated at it.

```
struct node * new_node = malloc(sizeof(struct node)); // address of a node * in heap

int * intArr = malloc(5 * sizeof(int));
intArr[3] = 2;
printf("%i\n", intArr[3]); // 2

int * intPtr = malloc(sizeof(int));
*intPtr = 6;
printf("%i\n", *intPtr); \\ 6

free(intArr);
free(intPtr);
free(new_node);
```

`calloc`

`calloc` is the same as malloc but it initializes the space with 0.

Reallocating Memory `realloc`

`realloc` changes the size of an existing allocating, in the case of an array, this means adding or removing elements while preserving content.

```
int * intArr = malloc(5 * sizeof(int));
intArr[3] = 2;
printf("%i\n", intArr[3]); // 2

intArr = realloc(intArr, 8 * sizeof(int));
intArr[7] = 5;
printf("%i, %i\n", intArr[3], intArr[7]); // 2, 5

free(intArr);
```

Processes

```
pid_t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed"); return 1;
} else if (pid == 0) { /* child process */
execlp("/bin/ls", "ls", NULL);
} else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
}
return 0;
```