Lecture 04
C Functions

In this lecture
• C functions
• Command line arguments
• Function prototypes
• Recursive Functions
• Runtime Stack
• Reference versus Value arguments
• Passing and returning values to/from functions
• Exercises

Each unit in a C program is a function. The entry point to a C program is the main program. The main program typically looks as:

```c
int main(int argc, char* argv[]) {
    return EXIT_SUCCESS;
}
```

The command line arguments given to a C program are processed by argc (argument count) and argv (the array of arguments). For example, if a C program executable a.out runs as:

```
>./a.out  file1  file2
```


Function Prototypes
A C program can be compiled w/o completing all its definitions. Function prototypes (w/o their definitions) are declared in the beginning of a program. Function prototypes do not require any variables declarations, but rather the types as shown below.

```c
void foo(int, int);

int main(int argc, char* argv[]) {
    int x =10, y =10;
    foo(x,y);
    return EXIT_SUCCESS;
}
```
The actual definition of the function can be defined below the main program at a later time.

**Recursive Functions**

A recursive function is a function that calls itself. Recursive functions heavily use the runtime stack to evaluate its value. Despite the cost of recursion, it is used widely since recursion is a natural problem solving method. For example, consider the following recursive definition of the mod (%) operator. Recall that $x\%y$ returns the remainder when $x$ is divided by $y$

$$\text{Mod}(x,y) = \begin{cases} x & \text{if } x \leq y \\ \text{Mod}(x-y, y) & \text{otherwise} \end{cases}$$

Try to convince yourself that this definition gives the actual answer.

The mathematical definition can be converted into a recursive C function as follows.

```c
int MOD(int x, int y) {
    if ( x <= y ) return x;
    else
        return MOD(x-y, y);
}
```

Although recursive functions are expensive to evaluate, they are elegant and the correctness of the algorithm can be proven mathematically.

**Runtime Stack**

Each application is allocated a runtime stack for storing of static variables and evaluating of functions. A runtime stack is used to allocate memory needed to execute the functions. Also stack is a great data structure to remember where to return after functions are executed. Before

**Reference versus value arguments**

All arguments to a C function are passed as “value” parameters. That is, a copy of the variable is given to function stack to be used in its value calculations. For example consider the function

```c
int foo(int x ){
}
```

When foo is called, a copy of the calling variable is passed into x (a local variable whose value is removed at function exit). However, a reference to a variable can be
passed to a function as a “pointer” argument. That is, function is given direct access to the passed variable. For example consider the function

```c
int foo(int* ptr){
    *ptr = 10;
    ....
}
```

If the function foo is called as:

```c
int x = 0;
foo (&x);
```

then the value of x will be affected by the statement *ptr=10. That is x will be changed to 10.

Hence a value of a variable or “value of an address of a variable” can be passed to function. In the former case we say we pass the variable as a value parameter and in the latter we say value is passed as a reference.

**Passing/Returning arguments to/from functions**

Values can be passed into or return from a function. However, one must be careful about not passing back a stack variable. For example, consider the following function.

```c
int* foo( ) {
    int x = 10;
    return &x;
}
```

The function returns the address of a stack variable that no longer exists upon return from foo. Hence the following code will throw a segmentation fault at execution.

```c
int* x = foo();
printf("%d", *x);
```