GROOT ACADEMY

Functions In C

Function Basics

⊙ What are functions?

↳ A function is a block of code that performs a specific task and can be reused throughout a program.

↳ think of it as a mini program with a name, inputs (optional) and an output (optional).

↳ C program always starts with main(), which itself is a f(n).

↳ Syntax:

```
return_type function_name (parameter_list) {
    // Body of function
    return value;    // based on return_type
}
```

⊙ Return Type - what function gives back (e.g. int, void - for no return)

⊙ Function Name - a unique identifier (e.g. add, printHello)

⊙ Parameters - inputs the function needs (eg. int a, int b)

⊙ Body - the code that does the work.

↳ f(n) Declaration - telling compiler about the f(n) (eg. int add(int, int);

↳ f(n) Definition - writing the actual code (eg. return a+b);

↳ Call - using the f(n) - (e.g. add(3, 4)).

↳ ex-

```
#include <stdio.h>
// declaration
int add (int a, int b);

int main() {
    int result = add(5,3);        // call
    printf("sum: %d \n", result);
    return 0;
}
// definition
int add (int a, int b) {
    return a+b;
}
```

↳ do for Multiply, Divide

⇒ Advantages of using functions

• Why use functions ?

⇒ Modularity = breaking of bigger problem into smaller manageble pieces.

⇒ Reusability = write once, use anywhere

⇒ readibility, debugging

⇒

```
#include <stdio.h>

void printGreeting() {                    // reusable fun
    printf("Hello, welcome to C.\n");
}

int main() {
    printGreeting();                      // call1
    printf("learning C");
    printGreeting();                      // call2
    return 0;
}
```

GROOT ACADEMY

⊙ Recursion

↳ what is Recursion ?

⇒ a function calling itself to solve a problem by breaking it into smaller versions of the same problem.

⇒ Requires a base case (to stop) and a recursive case (to keep going).

⇒ ex- Factorial

```
#include <stdio.h>
int factorial (int n) {
    if (n==0 || n==1) {
        return 1;
    }
    return n * factorial(n-1);   // recursive call
}
int main() {
    int n=5;
    printf("factorial of %d is %d \n", n, factorial(n));
    return 0;
}
```

// Output: factorial of 5 is 120. (5×4×3×2×1 = 120)

⇒ factorial (5)

→ 5 * factorial (4)

→ 4 * factorial (3)

→ 3 * factorial (2)

→ 2 * factorial (1)

→ 1 (base case)

<- 2 * 1 = 2

<- 3 * 2 = 6

<- 4 * 6 = 24

+91-8233266276
info@grootacademy.com
122/166, 2nd Floor, vijay path, Mansarovar, 302020
www.Grootacademy.com

⊙ No base call = Infinite recursion = stack overflow crash!

⊙ Variable Storage Classes:

↳ what are storage classes?

⊙ define how variables are stored, their scope (visibility) and lifetime (how long they exist).

⊙ four main types in C: auto, register, static, extern.

↳ Details:

(a) Auto (default)

→ local to a block (e.g. { })
→ lifetime: exists only while the block runs
→ example: int x; in a function

(b) register:

→ suggest storing in CPU registers for speed
→ scope: local, lifetime: block
→ example - register int count;

(c) static:

→ retains value b/w function calls
→ scope: local, lifetime: entire program
→ ex: counter in a function

(d) extern:

→ declares a variable defined elsewhere (global).
→ scope: global, lifetime: entire program.

↳ ex - #include <stdio.h>

```
void counter () {
    static int count = 0;      // retains value
    count++;
    printf ("count : %d \n", count);
}
int main() {
    counter();    // 1
    counter();    // 2
    counter();    // 3
    return 0;
}
```

[auto - temp. worker, static - permanent employee]

variable Arguments functions:

↳ what are they?

⇒ functions that accept a variable number of arguements (eg. Printf())

⇒ use <stdarg.h> library with macros: va_list, va_start, va_arg, va_end

⇒ working: first parameter is fixed, rest accessed by va_list

Ex. ⇒ Average of Numbers

// f(n) to calculate average of any number of integers.

```
#include <stdio.h>
#include <stdarg.h>

// function to find avg. of variable numbers
double average (int count, ...) {
    va_list args;          // declare argum. list
    va_list
```

```
    va_start(args, count);          // start after count

    int sum = 0;
    for (int i = 0; i < count; i++) {
        sum += va_arg(args, int);
    }
    va_end(args);
    return (double)sum / count;
}

int main() {
    printf("Avg. of 3 numbers: %.2f\n", average(3,10,20,30));
    printf("Avg. of 4 numbers: %.2f\n", average(4,5,10,15,20));
    return 0;
}
```

↳ Avg. of 3 numbers: 20.00
Avg. of 4 numbers: 12.50

→ Explanation:

a) count is fixed, ... → means any no. of arguements after this
b) va_list args → creates a variable to hold argument list
c) va_start(args, count) → sets args to point the first variable argument after count
d) va_arg(args, int) → fetches next integer from list, moving the pointer each time
e) loop runs (count) times to sum all numbers
f) va_end(args): frees resources used by va_list.

→ Memory Picture:

for average(3, 10, 20, 30)

Stack: [count = 3] [10] [20] [30]

va_start = points args to 10
va_arg = gets 10, then 20, then 30.

⇒ Function Examples

1.) Function with no parameter & return value
↳ print a simple message using a function

```
#include <stdio.h>
void greet() {
    printf("Hello, welcome to Cprogramming!\n");
}
int main() {
    greet();          // function call
    return 0;
}
```

2.) Function with parameters & NO return value
↳ take two numbers as inputs & return their sum

```
#include <stdio.h>
void add(int a, int b) {
    printf("sum: %d\n", a+b);
}
int main() {
    add(5, 10);       // function call
    return 0;
}
```

3.) Function with Return value : (takes input & returns output)
↳ calculate & return square of a number

```
#include <stdio.h>
int square(int num) {
    return num * num;
}
```

```
int main() {
    int result = square(6);
    printf("Square : %d \n", result);
    return 0;
}
```

4.) Function with Array as argument
↳ calculates & returns sum of array elements

```
#include <stdio.h>

int sumArray (int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum = sum + arr[i];
    return sum;
    }

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    printf("Sum of array : %d \n", sumArray(numbers, size));
    return 0;
}
```

20/4

5.) functions Returning Multiple values (using pointers)
↳ swap two numbers using a function

```
#include <stdio.h>

void swap (int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main() {
    int x = 5, y = 10;
    printf("Before swap: x= %d, y= %d \n", x, y);
    swap(&x, &y);
printf("after swap: x= %d, y= %d \n", x, y);
return 0;
}
```

6.) function returning a pointer:

↳ find the largest element in an array & return its address;

```
#include <stdio.h>
int* findMax(int arr[], int size) {
    int *max = &arr[0];
    for (int i=1; i<size; i++) {
        if (arr[i] > *max) {
            max = &arr[i];
        }
    return max;
}
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int *max = findMax(numbers, 5);
    printf("the largest element: %d \n", *max);
    return 0;
}
```

+91-8233266276
info@grootacademy.com
122/166, 2nd Floor, vijay path, Mansarovar, 302020
www.Grootacademy.com

① One more Recursion example,

↳ print numbers from n to 1, then say blast off!, when it's zero.

↳ #include <stdio.h>

```
void countdown(int n){
    // base case: when n reaches zero
    if (n == 0) {
        printf("Blast off !!\n");
        return;          // stop recursion
    }

    // recursive case: print n, then count down from n-1
    printf("%d\n", n);
    countdown(n-1);
}

int main() {
    countdown(5);        // starts from 5;
    return 0;
}
```

```
// 5
4
3
2
1
Blast off !!
```

② Base case - when do we stop, otherwise it'd count forever!

③ Recursive call - each step reduces the problem (n-1) until we hit the base."

④ Stack concept - each call waits for the next to finish, like stacking plates

⇒ recursion is a when function calls itself to solve a problem by breaking it into smaller simpler versions of the same problem. it like a loop, but instead of iterating, it dives deeper until hits a stopping point.

a) Base call - cond^n to stop recursion, b) Recursive call -