Functions in C

Introduction

- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

Defining a Function

```
return_type function_name( parameter list ) {
   body of the function
}
```

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

```
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declaration

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

```
return_type function_name( parameter list );
```

- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value

```
int max(int num1, int num2);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
}
```

```
Max value is : 200
```

Contd..

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Passing Arrays as Function Arguments

- Method – 1

```
void myFunction(int param[10]) {
    .
    .
    .
}
```

Passing Arrays as Function Arguments

- Method 2

```
void myFunction(int param[]) {
   .
   .
   .
}
```

Example

```
double getAverage(int arr[], int size) {
  int i;
  double avg;
  double sum = 0;
  for (i = 0; i < size; ++i) {`
    sum += arr[i];
  }
  avg = sum / size;
  return avg;
}
```

Function Arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

Arguments

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by reference

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Contd…

- By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Function Call by Value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function

Example

```
/* function definition to swap the values */
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

}
```

Contd...

```
#include <stdio.h>
/* function declaration */
void swap(int x, int y);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   /* calling a function to swap the values */
   swap(a,b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
```

Output

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Function call by reference

- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.
- To pass a value by reference, argument pointers are passed to the functions just like any other value.
- So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

Example

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

}
```

Contd..

```
#include <stdio.h>
/* function declaration */
void swap(int *x, int *y);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
```

Output

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

Contd…

```
#include <stdio.h>
/* function declaration */
double getAverage(int arr[], int size);
int main () {
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;
   /* output the returned value */
   printf( "Average value is: %f ", avg );
   return 0;
}
```

Output

- Average value is: 214.400000

Passing 2D Array to Functions

```
#include <stdio.h>
const int n = 3;

void print(int arr[3][3], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

Passing array directly to function

```
void printArray(int arr1[], int size) {
    int i;
    printf("Array elements are: ");
    for(i = 0; i < size; i++)
    {
        printf("%d, ", arr1[i]);
    }
}
int main() {
    int arr[5]={1,2,3,4,5};
    printArray(arr, 5);    // Pass array directly to function printArray
    return 0; }
```

Passing array using pointer

```
void printArray(int *a, int size)
{
    int i;
    printf("Array elements are: ");
    for(i = 0; i < size; i++)
    {
        printf("%d, ", *(a+i));     // a[i]  or *(a+i)
   }
}
int main()
{
    int arr[5]={1,2,3,4,5};
    printArray(arr, 5);    // Pass array directly to function printArray
    return 0;}
```

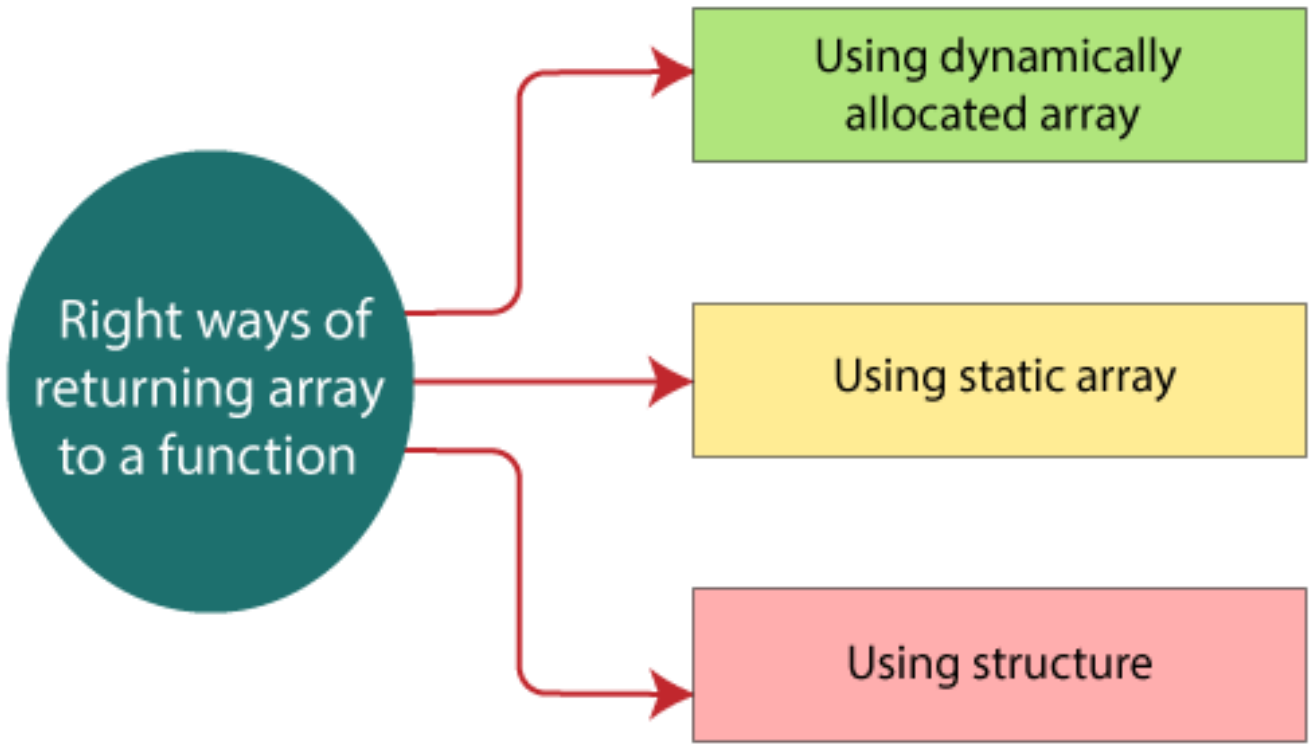
Return array from function

```
int* getarray()
{
   static int arr[5]={1,2,3,4,5};
    return arr;
}
int main()
{
  int *n;
  n=getarray();     *(n+0)  *(n+1)  *(n+2)
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
       printf("%d", *(n+i)); // *(n+i)
    }
    return 0;
}
```

- int* getarray() or int *getarray() – anything is correct
- **ERROR - address of local variable 'arr' returned**
- The problem is, we return address of a local variable which is not advised as local variables may not exist in memory after function call is over.
- So in simple words, Functions can’t return arrays in C.

Return array from function

- There are three alternate right ways of returning an array to a function:
  - Using dynamically allocated array
  - Using static array
  - Using structure



Return array from function

```
#include <stdio.h>

int* getarray(int *a)
{
   for(int i=0;i<5;i++)
   {
      *(a+i)=i+1;   // (or) a[i] = i+1;
   }

   return a;    a<- &arr[0] <- arr
}
```

```
int main()
{
  int *n;
  int arr[5];
  n=getarray(arr);
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
   {
      printf("\n%d", n[i]);
   }
   return 0;
}
```

Return array from function (static array)

```
#include <stdio.h>
int* getarray()
{
  static int p[5];

  for(int i=0;i<5;i++)
   {
      p[i] = i+1;
   }
   return p;
}
```

```
int main()
{
 int *n;
 n=getarray();
 printf("\nElements of array are :");

 for(int i=0;i<5;i++)
  {
     printf("\n%d", n[i]);
  }
  return 0;
}
```

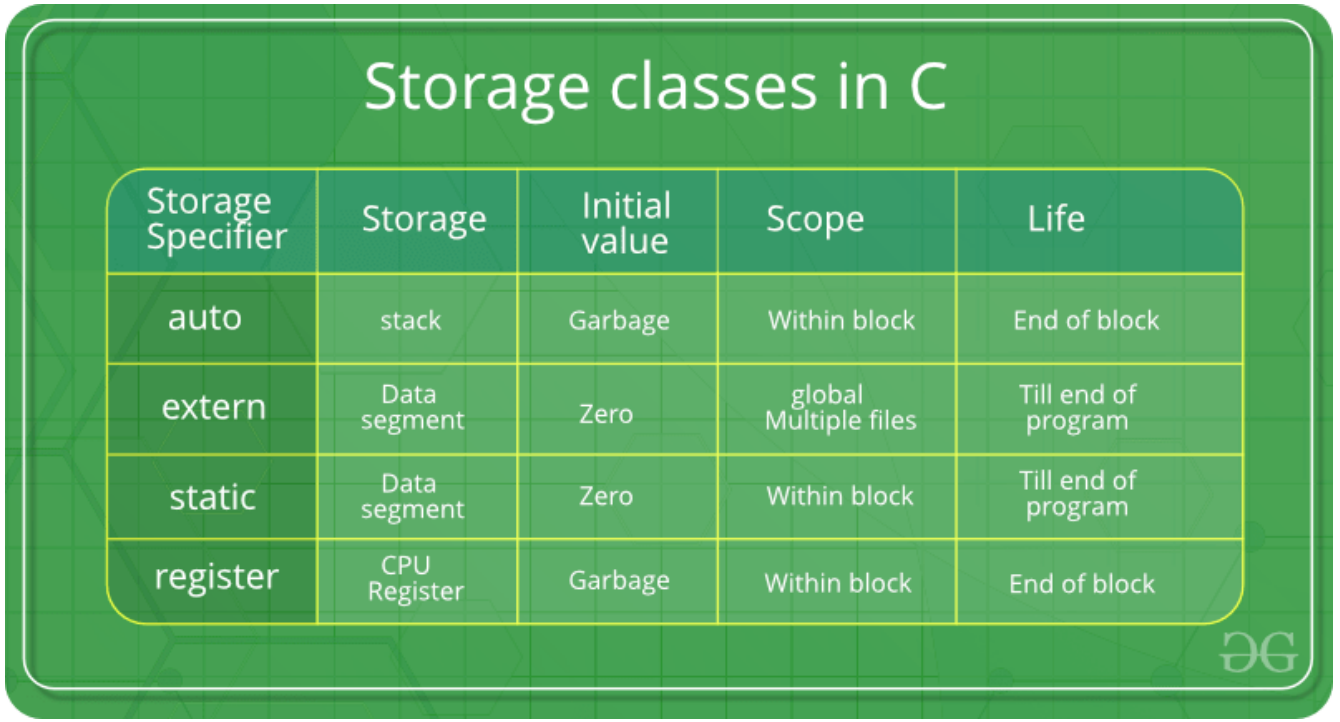
Return array from function (dynamically allocated array)

```
#include <stdio.h>
#include<malloc.h>
int* getarray()
{
  int *p = malloc(5*sizeof(int));

  for(int i=0;i<5;i++)
   {
      *(p+i) = i+1;
   }
   return p;
}
```

```
int main()
{
  int *n;
  n=getarray();
  printf("\nElements of array are :");

for(int i=0;i<5;i++)
    {
       printf("\n%d", n[i]);   // *(n+0)
    }
    return 0;
}
```

storage class

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.
- C language uses 4 storage classes,
  - auto
  - register
  - static
  - extern



The auto Storage Class (local variables)

- The auto storage class is the default storage class for all local variables.
- All variables in C that are declared inside the block, are automatic variables by default.
- We can explicitly declare an automatic variable using auto keyword.

```
void main(){
int x=10;//local variable (also automatic)
auto int y=20;//automatic variable
}
```

The extern Storage Class (Global Variable)

- Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
- Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.
- So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere.

```
#include <stdio.h>
extern int x=10;//external variable (also global)
void printValue(){
   printf("Global variable: %d", global_variable);
}
```

The register Storage Class

- This storage class declares register variables that have the same functionality as that of the auto variables.
- The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free registration is available.
- This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.
- If a free registration is not available, these are then stored in the memory only.

```
register int  miles;
```

The static Storage Class

- The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.
- Making local variables static allows them to maintain their values between function calls.
- For static applied to global variables, it causes that variable's scope to be restricted to the file in which it is declared.

```
void function1(){
int x=10;//local variable
static int y=10;//static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

Static variable

```
#include <stdio.h>
static int count = 5; /* global variable */

void func() {
  static int i = 5; /* local static variable */
   i++;
  printf("i is %d and count is %d\n", i, count);
}
main() {
  while(count--) {
      func();
   }
   return 0;
}
```

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

Consider a Big Classroom

Recursion

- Recursive functions
  - **Functions that call themselves**
  - Can only solve a base case

Recursion in Real Life

- John is seated in the last row of a very big classroom. Mike is sitting in last but one row of the same classroom. John wants to know how many rows are there in the classroom.
- John: Hi Mike! Can you please say me in which row you are seated?
- Mike: Yes… Of course John.
- Mike ask the same question to Patil who is seated in front of him and so on… till Alice who is seated first row is asked the question
- Alice: Alice can answer as one to Jack who is seated in the second row
- This is the base case
- Jack adds 1 to Alice's answer and tells it to Jill who is in the third row and so on… until the answer reaches John

Recursive functions

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

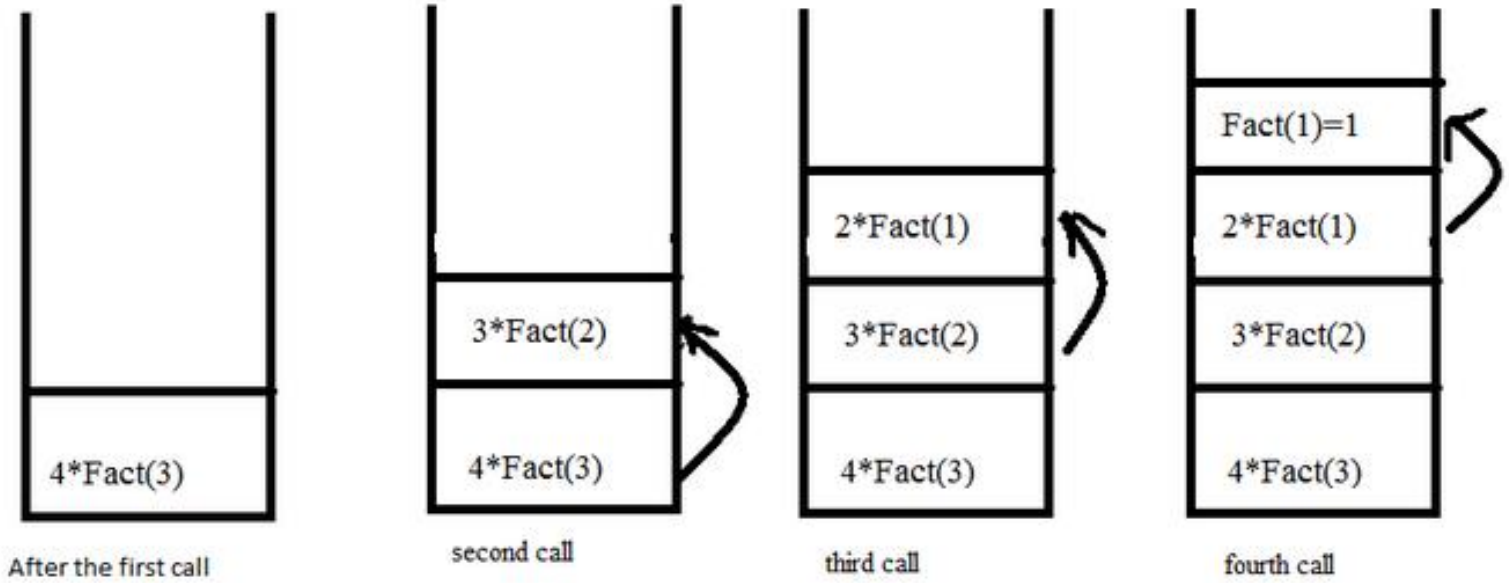
Factorial using Recursion

```c
#include<stdio.h>
int fact(int);
void main()
{
 int num;
 scanf("%d",&num);
 printf("%d",fact(num));
}

int fact(int n)
{
 if ((n==0)||(n==1))
        return 1;
 else
        return n*fact(n-1);
}
```

Base Case

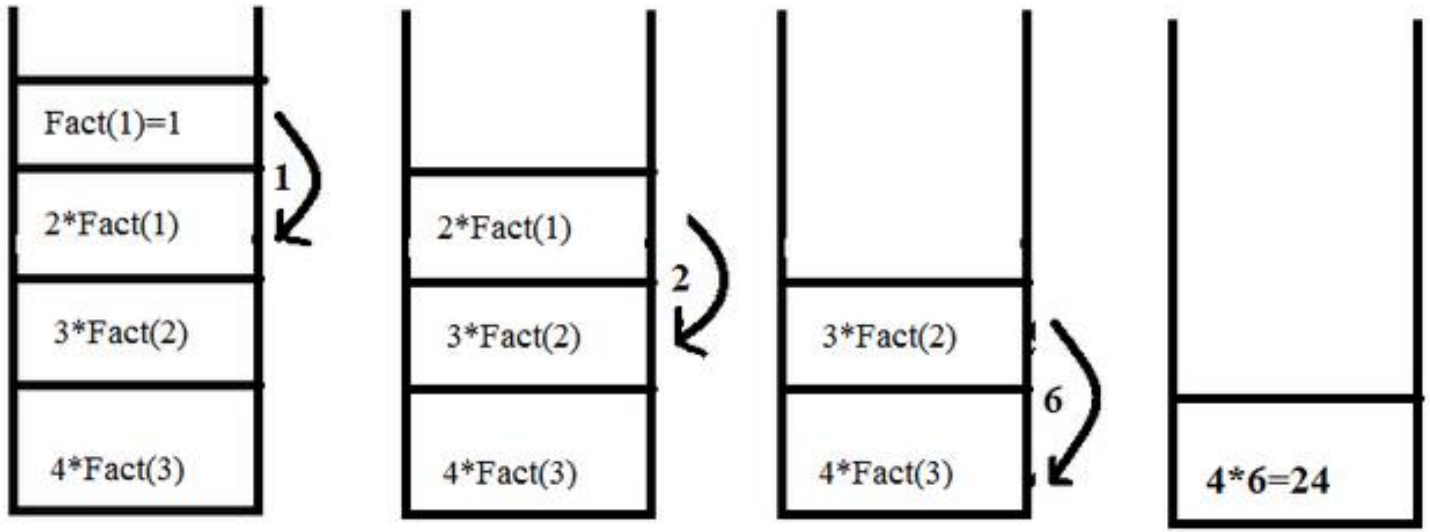
Recursive Call

When function call happens previous variables gets stored in stack



After the first call | second call | third call | fourth call

Returning values from base case to caller function



Fibonacci series

0, 1, 1, 2, 3, 5, 8…

```
int fibonacci( int n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci( n - 1) +
        fibonacci( n – 2 );
}
```

Function reverse_input_words

```
/*
 * Take n words as input and print them in reverse order on separate lines.
 * Pre: n > 0
 */
void
reverse_input_words(int n)
{
      char word[WORDSIZ]; /* local variable for storing one word          */

    if (n <= 1) { /* simple case: just one word to get and print          */

          scanf("%s", word);
          printf("%s\n", word);

    } else { /* get this word; get and print the rest of the words in
                reverse order; then print this word                       */

          scanf("%s", word);
          reverse_input_words(n - 1);
          printf("%s\n", word);
    }
}
```

```c
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char fromrod, char torod, char auxrod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", fromrod, torod);
        return;
    }
    towerOfHanoi(n-1, fromrod, auxrod, torod);
    printf("\n Move disk %d from rod %c to rod %c", n, fromrod, torod);
    towerOfHanoi(n-1, auxrod, torod, fromrod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```

Recursion vs. Iteration

- **Repetition**
  - Iteration: explicit loop
  - Recursion: repeated function calls
- **Termination**
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)