Programming in C

- Passing Array to Function
- Recursion

Contents

Array

- Array is homogeneous collection of elements stored at contiguous location.

e.g. int a[5];
int b[10][20];
float marks[10];

Passing Array to Function

- Way-1
  - Formal parameters as a pointer –

```
void myFunction(int *param)
{ . . . }
```

- Way-2
  - Formal parameters as a sized array –

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

- Way-3
  - Formal parameters as an unsized array –

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

EXAMPLE

```
#include <stdio.h>
void Average(int arr[], int size);
void main ( )
{
int  n[5] = {1000, 2, 3, 17, 50};
Average( n, 5) ;
}
```

```
void Average(int arr[], int size)
{
 int i;
float avg;
float sum = 0;
 for (i = 0; i < size; i++)
 {
sum= sum+ arr[i];
}
avg = sum / size;
printf( "Average value is: %f ",
    avg );
}
```

RECURSION

- The process in which a function calls itself directly or indirectly is called recursion.
- Types of recursion:
  - 1. Direct
  - 2. Indirect

DIRECT RECURSION

A function is said to be direct recursive if it calls itself directly.

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

INDIRECT RECURSION

A function is said to be indirect recursive if it calls another function and this new function calls the first calling function again.

Example:

```
int func1(int n)
{
 if (n<=1)
return 1;
 else
return func2(n);
}
 int func2(int n)
{  return func1(n);
}
```

- In this program, *func1()* calls *func2()*, which is a new function. But this new function *func2()* calls the first calling function, *func1()*, again. This makes the above function an indirect recursive function.

Advantages and Disadvantages of Recursion

- **Advantages:**
  - Recursion makes program elegant and cleaner.
  - Better understanding of algorithm.
  - Less code and small programs.
- **Disadvantages:**
  - Slow speed.
  - More memory space is required to store intermediate results.

Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int addNumbers(int n);
void main()
{ int num;
printf("Enter a positive integer: ");
 scanf("%d", &num);
printf("Sum = %d", addNumbers(num));
}
 int addNumbers(int n)
{ if(n != 0)
return n + addNumbers(n-1);
else
 return n;
 }
```

Output

Enter a positive integer: 20
Sum = 210

Factorial program in C using recursion

- As factorial is *n!= (n-1)! * n*,
- *factorial* function calculates the factorial by recursively multiplying *n* with factorial of *(n-1)*.

return 5 * factorial(4) = 120
└ return 4 * factorial(3) = 24
└ return 3 * factorial(2

Factorial program in C using recursion

```
#include<stdio.h>
int factorial(int);
void main()
{
 int n;
 int f;
 printf("Enter an integer to find its
   factorial\n");
 scanf("%d", &n);
 if (n < 0)
 printf("Factorial of negative integers isn't
   defined.\n");
 else
 {
  f = factorial(n);
  printf("\nFactorial = %d", f);
 }
}
```

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

Explanation:
•The number whose factorial is to be found is stored in the variable *n*.
•A recursive function ***factorial(n)*** calculates the factorial of the number.
•As factorial is *n!= (n-1)! * n*,
•*factorial* function calculates the factorial by recursively multiplying *n* with factorial of *(n-1)*.
•Finally, when *n = 0*, it returns 1 because *0! = 1*.

Program for Fibonacci Series

```
#include <stdio.h>
 int fibonacci(int i)
{
if(i == 0)
{ return 0;
 }
if(i == 1)
{ return 1; }
 return fibonacci(i-1) + fibonacci(i-2); }
void main()
{ int i;
 for (i = 0; i < 10; i++)
{ printf("%d\t\n", fibonacci(i)); }
}
```

Ackermann function

- In computability theory, the **Ackermann function**, named after Wilhelm Ackermann, is one of the earliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.
- One common version, the two-argument **Ackermann–Péter function**, is defined as follows for nonnegative integers m and n:

$$A(m,n) = \begin{cases} n+1 \\ A(m-1,1) \end{cases}$$

- Its value grows rapidly, even for small inputs. For example, $A(4, 2)$ is an integer of 19,729 decimal digits.

C Program to implement Ackermann function using recursion

```
#include<stdio.h>
int A(int m, int n);
main()
{
int m,n;
printf("Enter two numbers :: \n");
scanf("%d%d",&m,&n);
printf("\nOUTPUT :: %d\n",A(m,n));
}
int A(int m, int n)
{
if(m==0)
return n+1;
else if(n==0)
return A(m-1,1);
else
return A(m-1,A(m,n-1));
}
```

References

1. https://www.tutorialspoint.com/cprogramming/c_recursion.htm
2. https://www.geeksforgeeks.org/recursion/
3. https://beginnersbook.com/2014/01/c-passing-array-to-function-example/
4. https://www.tutorialspoint.com/cprogramming/c_passing_arrays_to_functions.htm
5. Let Us C , Yashwant Kanetkar
6. Programming in C, 2011, by J.B. Dixit
7. Basics of C Programming, 2011, by J.B. Dixit