Page No.
Date: 06 / 11 / 24

Unit-3

C function :- It is a block of code that perform some specific task.

In detail - A large C program is divided into basic building blocks called C function. It contains set of instructions enclosed by "{}" which performs specific operations in a C program. Actually, the collection of these functions creates a C program.

↳ A large program in C can be divided to many subprogram.

↳ The subprogram posses a self contain components & have well define purpose.

Types of C function :-

1. User define function
   ↳ main()

2. Library function
   ↳ printf(), scanf(), sqrt(), getchar().

Standard Library functions.

1. Library function in C language are inbuilt function which are grouped together & placed in a common place called library. Each library function in C performs specific operations.

2. The Standard library functions are built in function in C programming to handle tasks such as mathematical operations, I/O processing, string handling etc.

3. These functions are declared in the header file when you include the header file, these functions are available for use. ~~Ex~~ →

Ex ↓

The printf() is a standard library function to send output to the screen (display output on the screen). The function is declared in "stdio.h" header file. Once you include "stdio.h" in your program, all these functions are available for use like printf(), scanf().

User-defined function – C allows programmers to define function. Such function created by user are called user-defined function.

Uses of C function –

- C function are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C function to make use of same functionality wherever required.
- We can call functions any number of times in a program & from any place in a program.
- A large C program can easily be tracked when it is divided into functions.

The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality & to improve understandability of very large C programs.

- Function declaration/Prototype: It is used to give specific information to the compiler about the function so that it can check the function calls.
  The calling function needs information about the called function.
  If definition of the called function is placed before the calling function, then declaration is not needed.

  Syntax: return_type function_name (argument list);

Function Call: This calls the actual function.

Syntax: function_name (arguments list);
(arguments list → optional)

- Function definition: This contains all the statements to be executed.

  Syntax: Return_type function_name (arguments list)
  {Body of function;}

- Function should be declared & defined before calling in C program.

```
int main () {
  Compare (5, 8)
}
Compare (int x, int y)
{
  if (x > y)
    printf ("x is greatest");
    else
      printf ("y is greatest");
  }
```

Example —

In below program, function "Square" is called main function.
The value of "n" is passed as arguments to the function "square". This value is multiplied by itself in this function & multiplied value "f" is returned to main function "Square".

```
#include <stdio.h>
// function prototype, also called function declaration.
float square (float x);
// main function, program starts from here.
int main ()
{
float m, n;
printf ("\n Enter some number for finding Square\n");
scanf ("%f", &m);
// function call
n = Square(m); → actual fun
printf ("\n Square of given number %f is %f", m, n); }
```

```
float Square (float x) // function definition
                       ↘ formal argument
{
  float p;
  p = x * x ;
  return (p);
}
```

This will go to `n = Square (m);`

`*(&m)`

These Two ways that a C-function can be called from a program as follows:

`*` • Pointer to function. `&m`

1. Call by value
2. Call by reference (address)

- When a func^n is called by value, a copy of the actual argument's value is passed to the formal parameter.
- Any chances made to the formal parameter within the func^n do not affect the original value of the actual argument.

- In call by reference, the address of the actual argument is passed to the formal parameter.
- Any changes made to the formal parameter within the func^n directly affect the original value of actual argument.

feature	Call by Value	Call by reference
↳ argument passed	Copy of value	address of variable
↳ changes inside func^n	do not affect original value	affect original value
↳ Efficiency	less efficient	more efficient
↳ Complexity	Simpler to understand	more complex to understand

```
#include <stdio.h>
  float square (ff /* function declaration */
int max (int num1, int num2);
  int main () {
  /* local variable definition */
   int a = 200;
   int b = 800;
   int ret;
  /* calling a function to get max value */
   ret = max (a, b);
   printf (" Max value is %d \n", ret);
   return 0;
  }
  /* function returning the max b/w 2 no. */
    int max (int num1, int num2)
   {
  int result;
   if (num1 > num2)
   {
    result = num1;
   }
   else {
     result = num2;
    }
  return result;
}
```

```
#include <stdio.h>
long cube (long x);  /* func^n prototype */
long input, answer;
int main (void)
{
printf ("Enter an integer value:");
scanf ("%d", &input);
answer = cube (input);  /* Calling function */
printf ("\n The cube of %ld is %ld\n", input, answer);
return 0;
}
long cube (long x) /* function definition */
{
long x_cubed;
x_cubed = x*x*x;
return x_cubed;
}
```

argument formal parameter

return data type

actual parameter

↳ The variable to be passed on is x. (has single arguments) - value can be passed to function so it can perform the specific task. It is called arguments.

* Function with arguments & return value

These types of func^n have arguments, so the Calling func^n can send data to the Called func^n. The Called func^n can return any value to the Calling func^n using return statement.

* Function with arguments but no return value

* Function with no arguments but a return value.
* Function with no arguments and no return value.

↳ "Address of" & "value at" code

```
#include <stdio.h>
Void swapvalues (int *a, int *b);
int main ()
{
  int m, n;
  printf ("Enter 2 numbers:");
  Scanf ("%d %d", &m, &n);
  Swap_values (&m, &n);
  return 0;
}
void swap_values (int *a, int *b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
printf ("Values of a = %d, b = %d after swap", *a, *b);
}
```

Calling by reference

```
# include <stdio.h>
 void swapvalues (int a, int b);
 int main () {
   int a, b;
   printf ("Enter 2 numbers:");
   Scanf ("%d %d", &a, &b);
   Swapvalues (a, b);
   return 0;
 }
```

```
void SwapValues (int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
    printf ("Values of a = %d, b = %d after Swap", a, b);
}
```

↳ The function which is c/d at last will finish its function first.

Ex - c → 1st
b
a
main → last

c must finish than only b can complete then a & in last main function.

while writing the progm start with main but in Execution just opposite way. This is c/d stacking of function.

Ex → main {
    ≡
    a();
    ≡
}
a()
{ ≡
    b();
{ ≡

Q: Using function (Call by value) find wheather no. is Even or odd.

```
#include <stdio.h>
Void Even_or_Odd (int num); // function declaration
int main ()
{
```

```
int a;
printf ("Enter a number:");
Scanf ("%d", &a);
Even_or_Odd (a); // function Call
return 0;
}
Void Even_or Odd (int num) // function definition
{
if (num % 2 == 0)
{ printf ("Number is Even");
return 1;
}
else {
    printf ("Number is odd");
return 0;
} }
```

Q: Compare greater among 2 nos. using ternary operators.

```
#include <stdio.h>
void check_greater (int num1, int num2); // declaration
int main () {
int x, y;
printf ("Enter 2 numbers");
Scanf ("%d %d", &x, &y);
find_greater (x, y); // Calling
return 0; }
Void check_greater (int num1, int num2) { //definition
int result;
result = (num1 > num2) ? num1 : num2;
printf ("%d is greater among 2 numbers", result);
}
```

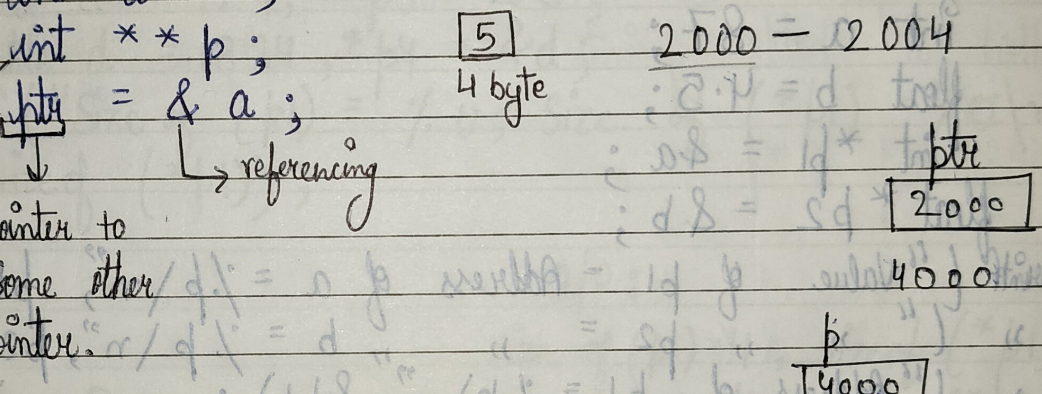
```
#include <stdio.h>
#include <math.h>
int isArmstrongNumber (int main) {
int originalNumber, remainder, n = 0, result = 0;
originalNum = num ;
// Count the number of digits
while (originalNum != 0) {
originalNum /= 10 ;
++n ; }
originalNum = num ;
// Calculate the sum of powers of digits
while (originalNum != 0) {
remainder = originalNum % 10;
result += pow (remainder, n);
originalNum /= 10 ;
}
// Check if the no. is Armstrong
if (result == num) {
return 1 ; // it's an armstrong number
} else {
return 0; // it's not an armstrong number.
}
}
int main () {
int num;
printf ("Enter an integer: ");
scanf ("%d", &num);
if (isArmstrongNumber (num)) {
printf ("%d is an armstrongNumber\n", num); }
else {
printf ("%d is not an armstrongNumber\n", num); }
return 0; }
```

Page No.: 1
Date: 13/11/24

★ Pointers :- It is a variable that holds / stores the address of some other variable.
It is called pointers because it points to a particular location in memory by storing the address of the location.
NULL pointer holds the value nothing.

```
int *ptr;
int a = 5;
int **p;
ptr = &a;
```

pointer to pointer variable.
pointer to some other pointer.
referencing



* requires some address.
& requires some variable.

```
a = 5
&a = 2000;        {ptr is storing address of a}
*(&a) = 5
*ptr = 5
&ptr = 4000
```

dereferencing

ptr = 2000
*(&2000) = 5.

* This address is the location of another obj in the memory.

Declaration of pointer variables.
It should be declared before being used.

The general syntax of declaration is :

data_type *pname ;

Eg., int *iptr ; float *fptr ; char *cptr, ch1, ch2 ;

/* Dereferencing pointer variables */

```
# include <stdio.h>
  int main (void)
{
  int a = 87;
  float b = 4.5;
  int *p1 = &a;
 float *p2 = &b;
printf ("Value of p1 = Address of a = %p\n", p1);
  "   "  "  p2 =  "  "  b = %p\n", p2);
  "Address of p1 = %p\n", &p1);
  "   "  "  p2 = %p\n", &p2);
  "value of a = %d %d %d\n", a, *p1, *(&a));
  "Value of b = %f %f %f\n", b, *p2, *(&b));
}
```

O/P :-

```
Value of p1 = Address of a = 0012 FED4
  "   "  p2 =  "  "  b = 0012 FEC8
Address of p1 = 0012 FEBC
  "   "  p2 = 0012 FEB0
Value of a = 87 87 87
  "   "  b = 4.5 4.5 4.5
```

If a pointer that contains Null is dereferenced, the results are implementation dependent.

/* Prgm to print Size of pointer variable & Size of values dereferenced by them */

```
#include <stdio.h>
int main(Void)
{
char a = 'x', *p1 = &a;
int b = 12, *p2 = &b;
float c = 12.4, *p3 = &c;
double d = 18.34, *p4 = &d;
printf("Size of (p1) = %u, Size of (*p1) = %u\n", sizeof(p1)
, sizeof(*p1));
.
.
.
return 0;
}
```

O/P :-

Size of (p1) = 4, Sizeof (*p1) = 1
" (p2) = 4, " (*p2) = 4
" (p3) = 4, " (*p3) = 4
" (p4) = 4, " (*p4) = 8

Pointer can have 3 kinds of content in it.

- Address of an object, which can be dereferenced.
- A null pointer
- Invalid content, which doesn't point to an object.

If p is a pointer to integer, then Int *p.

Declaring pointer

Data-type * name.
Variable

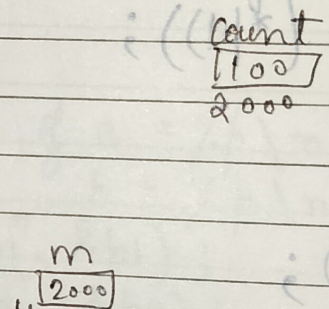
↳ * is a unary operator, also called as indirection operator.

↳ Data-type is the type of obj which the pointer is pointing.

↳ * is used to declare a pointer.

↳ When you write int *, Compiler assumes that any address that it holds points to an integer type.

pointer int **p;
int *m;
int count = 100;
normal m = &count;
p = &m;



* Functions performed in pointers are limited.
Increment, Decrement, A integer may be added to pointer (+ or +=), A integer may be subtracted to pointer (- or -=)

1. Increment a Pointer

↳ If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
↳ The pointer will get increased by the size of the data type to which the pointer is pointing.
↳ The Rule to increment the pointer is -
new_address = Current_address + i * sizeof(data type).

Ex, int a = 10
int *p;
p = &a;
p = p++

Suppose address of a is 2010.

p = p + 1 * 4 = 2010 + 4
= 2014

2. Decrementing pointer in C.

p = p - 1 * 4 = 2010 - 4 = 2006.

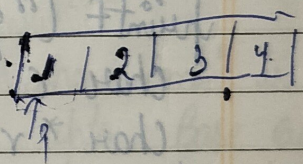
3. Pointer Addition.

float a = 10;
float *p;
p = &a;
p = p + 3;

Suppose address of a is 2410 which is stored in pointer variable p;
p = p + (3 * 4) = 2410 + 12
= 2422.

4. Pointer Subtraction.

p = p + 2;



p = p - (2 * 4) = 2410 - 8 = 2402

Illegal arithmetic with pointers.

pointer + pointer = Illegal
" * " = "
" % " = "
" / " = "
" & " = "
" ^ " = "
" | " = "
~ pointer = Illegal

```
#include <stdio.h>
 int main () {
int a = 10;
 int *p = &a;
 printf ("p = %u\n", p);
  p++;
 printf ("p++ = %u\n", p);
  p--;
 printf ("p-- = %u\n", p);
  float b = 10.0;
  float *q = &b;
printf ("q = %u\n", q);
  q++;
printf ("q++ = %u\n", q);
  q--;
printf ("q-- = %u\n", q);
  char c = 'x';
  char *r = &c;
printf ("r = %u\n", r);
  r++;
printf ("r++ = %u\n", r);
  r--;
printf ("r-- = %u\n", r);
 return 0;
}
```

When both pointers data type are same then only we can subtract otherwise not.

```
int a, b
int *p1, *p2;
p1 = &a; //2014
p2 = &b; //2010
```

$p_1 - p_2 = 1$

* No. of bytes / Size of element

Page No.:
Date: 18/11/24

Recursion : When a called function in turn calls another function a process of chaining occurs. It is a special case of this process, where a function calls itself again and again.

"Main" is a user-defined function.

↳ Recursion when a function call itself again & again & Every time it becomes closer to its base case.

↳ Recursive call

```
main ()
{
  ——————
  printf ("Hello");
  main ();
}
```

↳ Recursive function

$$n! = fact(n) = \begin{cases} 1 & \text{if } n == 0 \quad \text{(Base cond°)} \\ n * (n-1)! & \text{if } n >= 1 \quad \text{(Recursive call)} \end{cases}$$

Iteration : Terminate when cond° is false.
Recution : Terminate upto Base case.

$5! = 5 \times 4!$

$4! = 4 \times 3!$ (4×6)

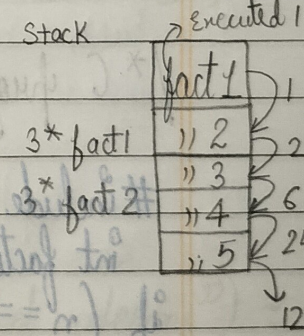
$3! = 3 \times 2!$ (3×2)

$2! = 2 \times 1!$ (2×1)

$1! = 1 \times 0!$

LIFO
↓
Last In First Out

Stack



★ The expansion is forward direction.

★ The collection is from downward.

↳ Base Case, when a function stops calling itself.

Sum (n) = n + Sum (n-1)
↳ (n-1) + Sum (n-2)
(n-n)
0

Page No.:
Date: 20 / 11 / 22

Types of Recursion

- Direct Recursion :- When a function calls itself directly
- Indirect :- When 2 or more functions call each other directly or indirectly.
- Tail Recursion :- A special type of recursion where the recursive call is the last operation performed by the function.

Ex :-
```
void fun (int n) {
    if (n == 0) return ;
    else printf ("%d", n);
    return fun (n-1) ; }
int main () {
    fun (3) ;
    return 0 ; }
```

func° definition me last me call hoga

/* C program to find the factorial using tail recursive */

```
#include <stdio.h>
int factorialTail (int n) {
if (n==1 || n==0){
    return 1;
}
    else {
        return n* factorialTail (n-1);
    }
}
```

func° call

```
int main (){
    int n = 5 ;
    int fact1 = factorialTail(n);
printf ("factorial of %d: %d", n, fact1);
    return 0;
}
```

O/P :- Factorial of 5: 120

- Non-Tail Recursion : It refers to a recursive function where the recursive call is not the last operation performed. This means that after the recursive call returns, there are still more operations to be done.

- Tree Recursion : It's a type of recursion where a function calls itself multiple times (more than 1 time) within its execution. This creates a tree-like structure of func° calls, with each call branching out into further calls.

Ex -

```
#include <stdio.h>
int fibonacci (int n) {
if (n==0) {
  return 0;
} else if (n==1) {
  return 1;
} else {
    return fibonacci (n-1) + fibonacci (n-2);
                              func° call
  }
}
int main () {
  int terms;
  printf ("Enter the number of terms");
  scanf ("%d", &terms);
printf ("fibonacci Series:");
for (int i=0; i<terms; i++) {
  printf ("%d", fibonacci (i));
  }
return 0;
}
```