PROGRAMMING IN C

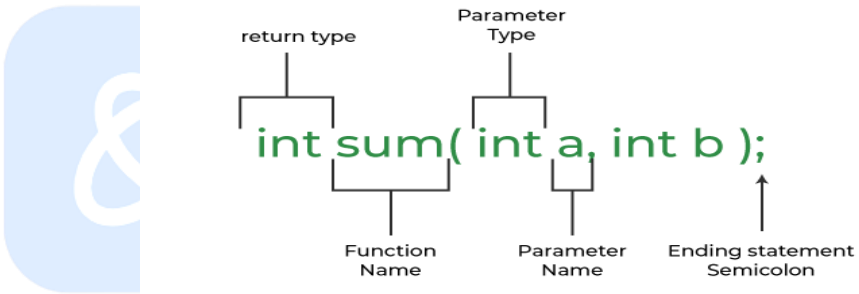
UNIT-3

Functions:

Functions are sets of statements that take inputs, perform some operations, and produce results. The operation of a function occurs only when it is called. Rather than writing the same code for different inputs repeatedly, we can call the function instead of writing the same code over and over again. Functions accept parameters, which are data. A function performs a certain action, and it is important for reusing code. Within a function, there are a number of programming statements enclosed by {}.

Example:

```
int sum (int a, int b);
```



Elements of Functions/User Defined Functions

1. Function Declarations:

Function declarations tell the compiler how many parameters a function takes, what kinds of parameters it returns, and what types of data it takes. Function declarations do not need to include parameter names, but definitions must.

Syntax:

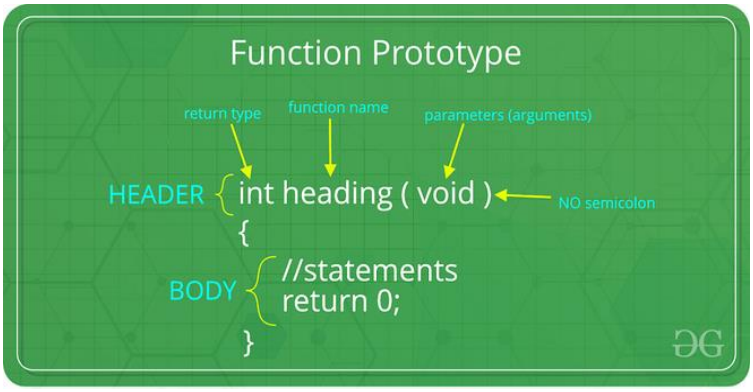
```
return_type name_of_the_function (parameters);
```

Function declaration for the sum of two numbers is shown below

```
int sum(int var1, int var2);
```

The parameter name is not mandatory while declaring functions. We can also declare the above function without using the name of the data variables.

```
int sum(int, int);
```



2. Function Definition:

A function definition consists function header and a function body.

```
return_type function_name (parameters)
{
    //body of the function
}
```

- **Return_type:** The function always starts with a return type of the function. But if there is no return value then the void keyword is used as the return type of the function.
- **Function_Name:** Name of the function which should be unique.
- **Parameters:** Values that are passed during the function call.

3. Function Call:

To Call a function parameter are passed along the function name. In the below example, the first sum function is called and 10,30 are passed to the sum function. After the function call sum of a and b is returned and control is also returned back to the main function of the program.

Example:

```
// C program to show function
// call and definition
#include <stdio.h>
// Function that takes two parameters
// a and b as inputs and returns
// their sum
int sum(int a, int b)
{
```

```
  return a + b;
}
// Driver code
int main()
{
  // Calling sum function and
  // storing its value in add variable
  int add = sum(10, 30);
  printf("Sum is: %d", add);
  return 0;
}
```

Output: `Sum is: 40`

Advantage of functions:

- There are the following advantages of C functions.
- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Types of Functions:

1. User Defined Function:

Functions that are created by the programmer are known as User-Defined functions or **"tailor-made functions"**. User-defined functions can be improved and modified according to the need of the programmer. Whenever we write a function that is case-specific and is not defined in any header file, we need to declare and define our own functions according to the syntax.

Advantages:

- Changeable functions can be modified as per need.
- The Code of these functions is reusable in other programs.
- These functions are easy to understand, debug and maintain.

2. Library Function:

A library function is also referred to as a **"built-in function"**. There is a compiler package that already exists that contains these functions, each of which has a specific meaning and is included

in the package. Built-in functions have the advantage of being directly usable without being defined, whereas user-defined functions must be declared and defined before being used.

Example:

`pow(), sqrt(), strcmp(), strcpy() etc.`

Advantages:

- C Library functions are easy to use and optimized for better performance.
- C library functions save a lot of time i.e., function development time.
- C library functions are convenient as they always work.

Categories of Functions:

A function is one that is defined by the user when writing any program, as we do not have library functions that have predefined definitions. To meet the specific requirements of the user, the user has to develop his or her own functions. Such functions must be defined properly by the user. There is no such kind of requirement to add any particular library to the program.

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and with return value

1. Function with No Arguments and No Return Value

Functions that have no arguments and no return values. Such functions can either be used to display information or to perform any task on global variables.

Syntax:

```
void function_name()
{
   return;
}
```

2. Function with No Arguments and With Return Value

Functions that have no arguments but have some return values. Such functions are used to perform specific operations and return their value.

Syntax:

```
return_type function_name()
{
  return value;
}
```

3. Function with Arguments and No Return Value

Functions that have arguments but no return values. Such functions are used to display or perform some operations on given arguments.

Syntax:

```
void function_name(type1 argument1, type2 argument2,...typeN argumentN)
{
   return;
}
```

4. Function with Arguments and With Return Value

Functions that have arguments and some return value. These functions are used to perform specific operations on the given arguments and return their values to the user.

Syntax:

```
return_type function_name(type1 argument1, type2 argument2,...typeN argumentN)
{
   return value;
}
```


Passing Parameters to Functions:

The value of the function which is passed when the function is being invoked is known as the **Actual parameters**. In the below program 10 and 30 are known as actual parameters.

Formal Parameters are the variable and the data type as mentioned in the function declaration. In the below program, a and b are known as formal parameters.

1. Pass by Value:

Parameter passing in this method copies values from actual parameters into function formal parameters. As a result, any changes made inside the functions do not reflect in the caller's parameters.

Below is the C program to show pass-by-value:

```
// C program to show use
// of call by value
#include <stdio.h>

void swap(int var1, int var2)
{
```

```
  int temp = var1;
  var1 = var2;
  var2 = temp;
}
// Driver code
int main()
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
        var1, var2);
  swap(var1, var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
        var1, var2);
  return 0;
}
```


Output:

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 3, 2
```

2. Pass by Reference:

The caller's actual parameters and the function's actual parameters refer to the same locations, so any changes made inside the function are reflected in the caller's actual parameters.

Below is the C program to show pass-by-reference:

```
// C program to show use of
// call by Reference
#include <stdio.h>

void swap(int *var1, int *var2)
{
  int temp = *var1;
```

```
  *var1 = *var2;
  *var2 = temp;
}
// Driver code
int main()
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
        var1, var2);
  swap(&var1, &var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
        var1, var2);
  return 0;
}
```

Output:

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 2, 3
```


Recursion:

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls.

Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Recursive Function:

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Memory allocation of Recursive method:

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore, we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.


```
int display (int n)
{
   if(n == 0)
      return 0; // terminating condition
   else
   {
      printf("%d",n);
      return display(n-1); // recursive call
   }
}
```

Passing Array to Function:

There are various general problems which requires passing more than one variable of the same type to a function. As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Syntax:

functionname(arrayname);

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

return_type function(type arrayname[]);

Declaring blank subscript notation [] is the widely used technique.

Second way:

return_type function(type arrayname[SIZE]);

Optionally, we can define size in subscript notation [].

Third way:

return_type function(type *arrayname);

C language passing an array to function example:


```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
}//end of for
return min;
}//end of function

int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9};//declaration of array
```

```
min=minarray(numbers,6);//passing array with size

printf("minimum number is %d \n",min);

return 0;

}
```

Output: minimum number is 3

Returning array from the function:

As we know that, a function cannot return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a,b,c), the function will return the last-mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function.

Syntax:

```
int * Function_name() {

return array_type;

}
```


Passing Strings to Function:

A string is a collection of characters enclosed inside double quotes (i.e., “hello everyone”).

We will create two functions, one is to send a string as a parameter and the other should take the string as input through the parameter.

```
#include <stdio.h>
void str (char *str)
{
  printf ("String is: %s", str);
}

int main ()
{
  char string[100];
  printf ("Enter a String:  ");
  scanf ("%s", &string);
  str (string);
```

```
  return 0;
}
```

Input:
Enter a String: Hello everyone, this is aniket malik.

Output:
String is: Hello everyone, this is aniket malik.

Explanation:

```
void str (char *str)
{
  printf ("String is: %s", str);
}
```

This is the function that takes the string which we have passed. It has one parameter *str, where * represents a pointer. In the next step, we used a simple print statement to print the string we passed.

```
int main() {
  char string[100];
  printf("Enter a String:  ");
  scanf("%s",&string);
  str(string);
  return 0;
}
```


This is the main function where we have declared a variable ‘string’ to store a string obtained from the user; by using a simple scanf statement. In the next step, we called the function str and ‘string’ was given as a parameter to the function str.
Immediately after calling the str() function the control goes to the str function and the code inside it gets executed (i.e., the string is printed).
Since the main method return type is int so we have returned 0.

Storage Classes:

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

1. Automatic
2. External
3. Static
4. Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

1. Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

2. Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

3. Register

1. The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.

2. We cannot dereference the register variables, i.e., we can not use &operator for the register variable.
3. The access time of the register variables is faster than the automatic variables.
4. The initial default value of the register local variables is 0.
5. The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
6. We can store pointers into the register, i.e., a register can store the address of a variable.
7. Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

4. External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Pointers:

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 bytes.

Consider the following example to define a pointer which stores the address of an integer.

int n = 10;

int* p = &n;

Pointer variables declaration and Initialization:

Pointer variables declaration:

Pointer variables are used to store the address of the variable. And a pointer variable can store only the address of the variable which has the same data-type as the pointer variable.

Syntax: data-type *pointer-variable name;

Example: int *ptr;

Initialization of pointer variables:

The initialization of the pointer variable is simple like other variable but in the pointer variable we assign the address instead of value.

Example:

int *ptr,var1;

ptr=&var1;

If we want to access the pointer variable then we have to understand the most important thing which is as follows:

If the '*' asterisk sign is before the pointer variable this means the pointer variable is now pointing to the value at the location instead of the pointing to location.

NULL Pointer:

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

int *p=NULL;

Pointer to array:

```
int arr[10];

int *p[10]=&arr;
```

Pointer to a function:

```
void show (int);

void(*p)(int) = &display;
```

Pointer to structure:

```
struct st {
   int i;
   float f;
}ref;

struct st *p = &ref;
```

Usage of pointer:

There are many applications of pointers in c language.

1. Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2. Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Advantages of Pointers:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduces length of the program and its execution time as well.

Disadvantages of pointers:

- Memory corruption can occur if an incorrect value is provided to pointers
- Pointers are Complex to understand.
- Pointers are majorly responsible for memory leaks.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause segmentation fault.

Address Of (&) Operator:

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>

int main(){

int number=50;

printf("value of number is %d, address of number is %u",number,&number);

return 0;

}
```

Output: value of number is 50, address of number is fff4

Accessing the value of a variable using pointer in C:

As we know that a pointer is a special type of variable that is used to store the memory address of another variable. A normal variable contains the value of any type like int, char, float etc, while a pointer variable contains the memory address of another variable.

Steps:

1. Declare a normal variable, assign the value
2. Declare a pointer variable with the same type as the normal variable
3. Initialize the pointer variable with the address of normal variable
4. Access the value of the variable by using asterisk (*) - it is known as **dereference operator**

Example:

Here, we have declared a normal integer variable num and pointer variable ptr, ptr is being initialized with the address of num and finally getting the value of num using pointer variable ptr.

```
#include <stdio.h>
int main(void)
{
    //normal variable
    int num = 100;
    //pointer variable
    int *ptr;
    //pointer initialization
    ptr = &num;
    //pritning the value
    printf("value of num = %d\n", *ptr);
    return 0;
}
```


Output: value of num = 100

Array through pointers:

An array name is a constant pointer to the first element of the array. Therefore, in the declaration –

double balance[50];

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance** –

```
double *p;
double balance[10];
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on. Given below is the example to show all the concepts discussed above −

```
#include <stdio.h>
int main () {
  /* an array with 5 elements */
  double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
  double *p;
  int i;
  p = balance;
  /* output each array element's value */
  printf( "Array values using pointer\n");
  for ( i = 0; i < 5; i++ ) {
    printf("*(p + %d) : %f\n",  i, *(p + i) );
  }
  printf( "Array values using balance as address\n");
  for ( i = 0; i < 5; i++ ) {
    printf("*(balance + %d) : %f\n",  i, *(balance + i) );
  }
  return 0;
}
```


When the above code is compiled and executed, it produces the following result –

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```