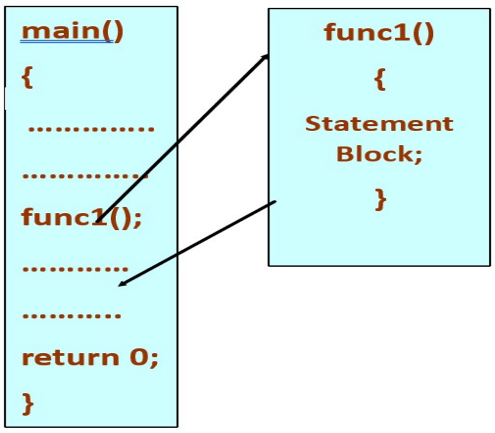
Course: Problem Solving Techniques using C

Course Code: 24BTPHY104

Module 4

4.FUNCTIONS AND POINTERS

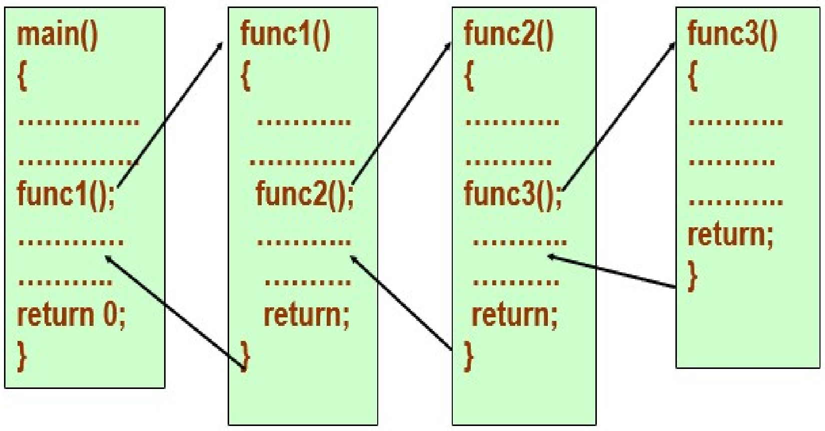
- A large C program is divided into basic building blocks called function. Function contains set of instructions enclosed by "{}" which performs specific operation in a program.
- Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from that of other functions.
- Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is specified by the function name.



- In the above figure main() calls another function, func1() to perform a well-defined task.
- main() is known as the calling function and func1() is known as the called function.
- When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.
- After the called function is executed, the control is returned back to the calling program.
- It is not necessary that the main() can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for

loop, while loop or do-while loop may call the same function multiple times until the condition holds true.

- It is not that only the main() can call another functions. Any function can call any other function. In the fig. one function calls another, and the other function in turn calls some other function.



Basically, there are two categories of function:

1. Predefined functions: available in C / C++ standard library such as stdio.h, math.h, string.h etc.

2. User-defined functions: functions that programmers create for specialized tasks such as graphic and multimedia libraries, implementation extensions or dependent etc.

Why Do We Need Functions?

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function (except main()), then there will be countless lines in the main().
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been prewritten and pre-tested, so the programmers use them without worrying about their code details. This speeds up program development.

Uses of Functions

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and 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 and to improve understandability of very large programs.

4.1 Function Definition

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.
- When a function defined, space is allocated for that function in the memory.
- The syntax of a function definition can be given as:

```
  return_data_type function_name(data_type var1, data_type var2,..)
{         ………….
                    statements
                    ………….
                    return(variable);
}
```

- The no. and the order of arguments in the function header must be same as that given in function declaration statement.

- The length of your program can be reduced.
- It becomes easy
- Functions can be called several times within your program.

All variables declared inside a function are local variables and are not accessible outside the function.

Syntax:

```
 return-value-type function-name( parameter-list)
 {
declarations and statements
```

Where,

- function-name: any valid identifier
- Return-value-type:
    - data type of the result (default int)
    - void – indicates that the function returns nothing
- Parameter-list:
    - Also called formal parameters.
    - A list of variables, comma separated list, declares parameters:
    - A type must be listed explicitly for each parameter unless, the parameter is of type int.
- Declarations and statements: function body (block)
    - Variables can be declared inside blocks (can be nested)
    - You cannot create functions within inside other functions.
- Returning control
    - If nothing returned
        - return;
        - or, until reaches right brace
    - If something returned
        - return expression;

4.2 Function Prototyping

Function prototype is a statement that tells the compiler about the function's name, its return type, numbers and data types of its parameters.

Function prototype works like a function declaration where it is necessary where the function reference or call is present before the function definition but optional if the function definition is present before the function call in the program.

A prototype functions is only used when its implementation comes after the main function.

Syntax

```
return_type function_name(parameter_list);
```

where,

- **return_type**: It is the data type of the value that the function returns. It can be any data type int, float, void, etc. If the function does not return anything, void is used as the return type.

- **function_name**: It is the identifier of the function. Use appropriate names for the functions that specify the purpose of the function.
- **parameter_list**: It is the list of parameters that a function expects in parentheses. A parameter consists of its data type and name. If we don't want to pass any parameter, we can leave the parentheses empty.

Program to illustrate the Function Prototype

```
#include <stdio.h>

// Function prototype
float calculateRectangleArea(float length, float width);

int main()
{
   float length = 5.0;
   float width = 3.0;

   // Function call
   float area = calculateRectangleArea(length, width);

   printf("The area of the rectangle is: %.2f\n", area);

   return 0;
}

// Function definition
float calculateRectangleArea(float length, float width)
{    return length * width;
}
```

Output

```
The area of the rectangle is: 15.00
```

4.3 Types of functions

A function may define with or without parameters, and it may or may not return a value. It entirely depends upon the user requirement. In Programming, as per our requirement, we can define the User-defined functions in multiple ways. The following are the types of Functions.

1. Function with no argument and no Return value.
2. With no argument and with a Return value.
3. With argument and No Return value.
4. With argument and Return value

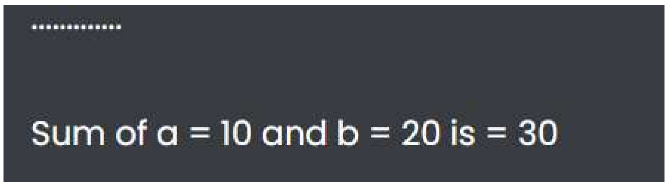
4.3.1 Function with no argument and no Return value

In this method, we won't pass any arguments to the function while defining, declaring, or calling it. This type of functions in C will not return any value when we call the function from main() or any sub-function. When we are not expecting any return value, we need some statements to print as output.

Example:

```
#include<stdio.h>
// Declaration
   void Addition();
void main()
 {
printf("\n ............ \n");
Addition();
}
void Addition()
{
 int Sum, a = 10, b = 20;
 Sum = a + b;
 printf("\n Sum of a = %d and b = %d is = %d", a, b, Sum);
}
```

Output:



```
..............

Sum of a = 10 and b = 20 is = 30
```

4.3.2 With no argument and with a Return value.

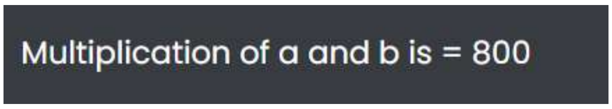
In this method, we won't pass any arguments while defining, declaring, or calling the function. This C type of function will return some value when we call it from the main() or any sub method.

The Data Type of the return value will depend upon the return type of function declaration. For instance, if the return type is int, then the return value will be int.

In these types of functions program, we are going to calculate the multiplication of 2 integers using the user defined functions without arguments and return keywords.

```
#include<stdio.h>

int Multiplication();

int main()
{
  int Multi;

  Multi = Multiplication();
  printf("\n Multiplication of a and b is = %d \n", Multi );

  return 0;
}


int Multiplication()

{

  int Multi, a = 20, b = 40;

  Multi = a * b;
  return Multi;
}
```

Output



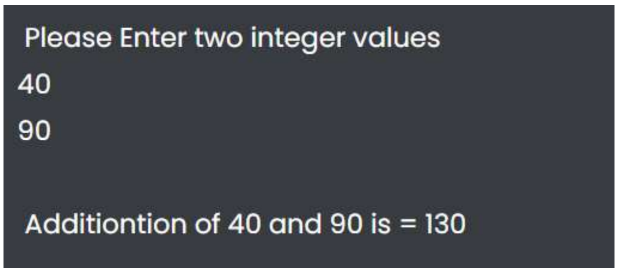
```
Multiplication of a and b is = 800
```

4.3.3 With argument and No Return value.

This method allows us to pass the arguments to the function while calling it. But, This C type of function will not return any value when we call it from main () or any sub method. If we want to allow the user to pass his data to the arguments, but we are not expecting any return value.

These Types of Functions allows the user to enter 2 integers. Next, we are going to pass those values to the user-defined function to calculate the sum.

```
#include<stdio.h>
void Addition(int, int);
void main()
{
  int a, b;
printf("\n Please Enter two integer values \n");
  scanf("%d %d",&a, &b);
  //Calling with dynamic values
  Addition(a, b);
}
void Addition(int a, int b)
{
  int Sum;
 Sum = a + b;
  printf("\n Addition of %d and %d is = %d \n", a, b, Sum);}
```

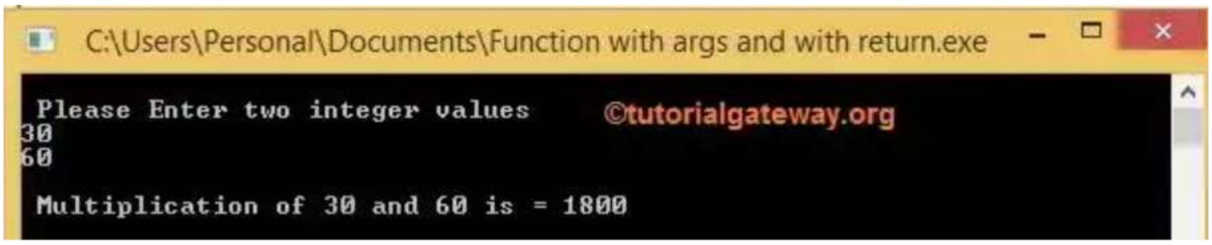


```
Please Enter two integer values
40
90

Additiontion of 40 and 90 is = 130
```

4.3.4 With argument and Return value

This method allows us to pass the arguments to the function while calling it. This type of function in C will return some value when we call it from the main () or any subfunction. The data Type of the return will depend upon the return type of function declaration. For instance, if the return type is int then the return value will be int. This type of user-defined function is called a fully dynamic one, and it provides maximum control to the end user. This Types of Functions allows the user to enter 2 integers. And then, we are going to pass those values to the UDFs to multiply them and return the output using the return keyword.

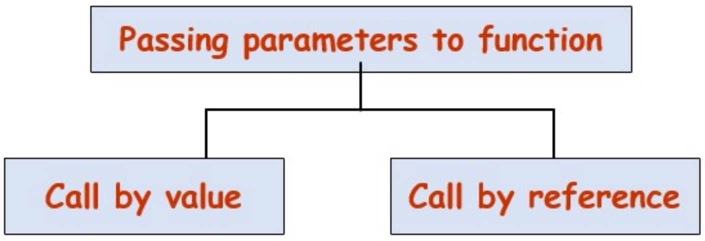
```
#include<stdio.h>
int Multiplication(int, int);
int main()
{
  int a, b, Multi;
  printf("\n Please Enter two integer values \n");
  scanf("%d %d",&a, &b);
  //Calling the with dynamic values
  Multi = Multiplication(a, b);
  printf("\n Multiplication of %d and %d is = %d \n", a, b, Multi);
  return 0;
}

     int Multiplication(int a, int b)
     {
       int Multi;
  Multi = a * b;
  return Multi;
}
```


C:\Users\Personal\Documents\Function with args and with return.exe
©tutorialgateway.org

```
Please Enter two integer values
30
60

Multiplication of 30 and 60 is = 1800
```

4.4 Passing arguments to functions

- There are two ways in which arguments or parameters can be passed to the called function.



- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.

4.5 Passing arrays to functions

In C, the whole array cannot be passed as an argument to a function. However, you can pass a pointer to an array without an index by specifying the array's name.

Syntax

Three ways to pass an array as a parameter to the function. In the function definition, use the following syntax:

return_type foo (**array_type array_name[size],** ...);

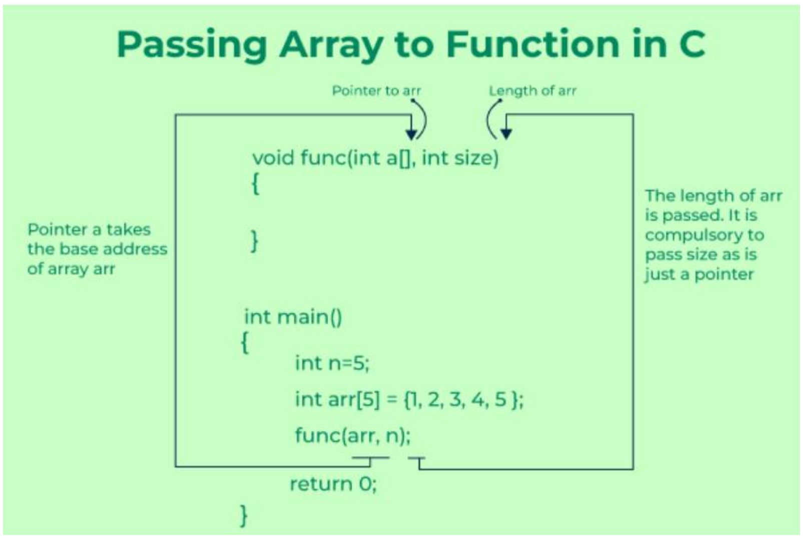
Mentioning the size of the array is optional. So the syntax can be written as:

return_type foo (**array_type array_name[]**, ...);

n both of the above syntax, even though we are defining the argument as array it will still be passed as a pointer. So we can also write the syntax as:

return_type foo (**array_type* array_name,** ...);

But passing an array to function results in array decay due to which the array loses information about its size.


Passing Array to Function in C
Pointer to arr
Length of arr
void func(int a[], int size)
{
}
int main()
{
int n=5;
int arr[5] = {1, 2, 3, 4, 5 };
func(arr, n);
return 0;
}
Pointer a takes the base address of array arr
The length of arr is passed. It is compulsory to pass size as is just a pointer

```
#include <stdio.h>

// Function prototype
void printArraySize(int arr[], int size);

int main() {
   // Define an array
   int myArray[] = {1, 2, 3, 4, 5};

   // Calculate the size of the array
    int size = sizeof(myArray) / sizeof(myArray[0]);
    // Pass the array and its size to the function
    printArraySize(myArray, size);

    return 0;
}

// Function to print the size of the array
void printArraySize(int arr[], int size) {
    printf("Size of the array passed to the function: %d\n", size);
}
```

Output

```
Size of the array passed to the function: 5
```

4.6 Passing strings to functions

String Function Arguments

In C, you can pass strings to functions in two ways:

- By passing an array of characters (i.e., a string)
- By passing a pointer to the first character in the string.

Passing an array of characters

Here's an example of passing an array of characters (string) to a function:

```
#include <stdio.h>
void print_string(char str[])
{
    printf("%s\n", str);
}
int main()
{
    char message[] = "Hello, world!";
    print_string(message);
    return 0;
}
```

Output:

```
Hello, world!
```

- The print_string function takes a single argument, str, which is an array of characters (i.e., a string).
- In the main function, a string message is defined and initialized with the value "Hello, world!". This string is then passed as an argument to the print_string function.
- The print_string function uses the printf function to print the string passed to it, along with a newline character to add a line break after the string.

Passing a pointer to the first character

And here's an example of passing a pointer to a string to a function:

```
#include <stdio.h>
void print_string(char *str)
{
    printf("%s\n", str);
}
int main()
{
    char message[] = "Hello, world!";
    print_string(message);
    return 0;
}
```

Output:

```
Hello, world!
```

Explanation

- The main function declares a string message with the value "Hello, world!", and passes it to the print_string function.
- The print_string function takes a pointer to a character as an argument and uses the printf function to print the string passed to it, along with a newline character.

4.7 Nested Functions

Some programmer thinks that defining a function inside another function is known as “nested function”. But the reality is that it is not a nested function, it is treated as lexical scoping. Lexical scoping is not valid in C because the compiler cannot reach/find the correct memory location of the inner function.

Nested function **is not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it’s not a nested function. Because nested functions definitions cannot access local variables of the surrounding blocks, they can access only global variables of the containing module. This is done so that lookup of global variables doesn’t have to go through the directory. As in C, there are two nested scopes: local and global (and beyond this, built-ins). Therefore, nested functions have only a limited use. If we try to approach nested function in C, then we will get compile time error.

// C program to illustrate the concept of Nested function.

```
#include <stdio.h>

void innerFunction1(void);
void innerFunction2(void);

void (*functionArray[])(void) = {innerFunction1, innerFunction2};

int main() {
   printf("Inside main function\n");

   // Simulate calling different nested functions
   int choice = 1; // Choose between 0 (innerFunction1) and 1 (innerFunction2)
   functionArray[choice](); // Call the chosen inner function
```

```
    return 0;
}

void innerFunction1(void) {
    printf("Inside innerFunction1\n");
}

void innerFunction2(void) {
    printf("Inside innerFunction2\n");
}
```

Output:

```
Inside main function

Inside innerFunction2
```

4.8 Call by value

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it.
- Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function.
  In the calling function no change will be made to the value of the variables.

```
#include<stdio.h>
void add( int n);
int main()
{
  int num = 2;
  printf("\n The value of num before calling the function = %d", num);
  add(num);
  printf("\n The value of num after calling the function = %d", num);
  return 0;
}
void add(int n)
```

```
{
  n = n + 10;
  printf("\n The value of num in the called function = %d", n);
}
```

Output:

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

The following points are to be noted while passing arguments to a function using the call-by-value method.

- When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.
- The values of the arguments passed by the function are copied into the newly created variables.
- Arguments are called by value when the called function does not need to modify the values of the original variables in the calling function.
- Values of the variables in the calling function remain unaffected when the arguments are passed by using call-by-value technique.

- Therefore, call-by-value method of passing arguments to a function must be used only in two cases:
    - When the called function does not need to modify the value of the actual parameter. It simply uses the value of the parameter to perform its task.
    - When we want the called function should only temporarily modify the value of the variables and not permanently. So, although the called function may modify the value of the variables, these variables remain unchanged in the calling function.

4.9 Call by reference

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.

- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.

- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

```
#include<stdio.h>
void add( int *n);
int main()
{
  int num = 2;
  printf("\n The value of num before calling the function = %d", num);
  add(&num);
  printf("\n The value of num after calling the function = %d", num);
  return 0;
}
void add(int *n)
{
  *n = *n + 10;
  printf("\n The value of num in the called function = %d", *n);
}
```

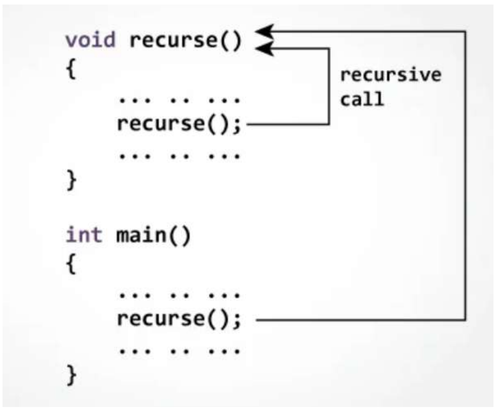
Output:

```
The value of num before calling the function = 2
The value of num in the called function = 12
```

4.10 Recursive functions

A recursive function is a function that calls itself either directly or indirectly through another function.

We cannot define function within function, but we can call the same function within that function.



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Classic example for recursive function is sum of Natural number using Recursion

```
#include <stdio.h>
int sum(int n);
int main() {
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum = %d", result);
    return 0;
}
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

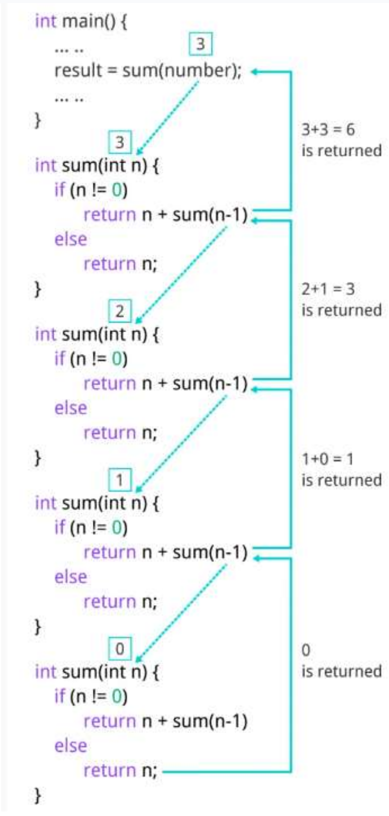
Output:

```
Enter a positive integer:3
sum = 6
```

Initially, the sum() is called from the main() function with *number* passed as an argument.

Suppose, the value of *n* inside sum() is 3 initially. During the next function call, 2 is passed to the sum() function. This process continues until *n* is equal to 0.

When *n* is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the main() function.



```
int main() {
  ... ..
  result = sum(number);
  ... ..
}
int sum(int n) {
  if (n != 0)
     return n + sum(n-1)
  else
     return n;
}
int sum(int n) {
  if (n != 0)
     return n + sum(n-1)
  else
     return n;
}
int sum(int n) {
  if (n != 0)
     return n + sum(n-1)
  else
     return n;
}
int sum(int n) {
  if (n != 0)
     return n + sum(n-1)
  else
     return n;
}
```

4.11 Pointers

Pointer is a variable that stores the address of another variable. A pointer in c is used to allocate memory dynamically at run time. The Pointer variable might be belonging to any of the data type such as int, float, chat, double,short etc.

Syntax:

Data_type*var_name;

Example:

Int*p;

Char*p;

4.11.1 Declaration of Pointer variable

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

4.11.2 Pointer arithmetic

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn't store any value. Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type

5. Comparison of pointers

Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

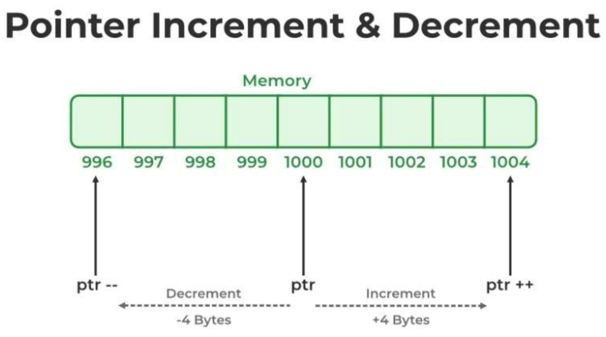
For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004.**

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.



Example of Pointer Increment and Decrement

```
#include <stdio.h>
// pointer increment and decrement
//pointers are incremented and decremented by the size of the data type they point to
int main()
{
int a = 22;
int *p = &a;
printf("p = %u\n", p); // p = 6422288
p++;
printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes
p--;
printf("p-- = %u\n", p); //p-- = 6422288          -4 // restored to original value

float b = 22.22;
float *q = &b;
printf("q = %u\n", q); //q = 6422284
q++;
printf("q++ = %u\n", q); //q++ = 6422288        +4 // 4 bytes
q--;
printf("q-- = %u\n", q); //q-- = 6422284          -4 // restored to original value

char c = 'a';
char *r = &c;
printf("r = %u\n", r); //r = 6422283
r++;
printf("r++ = %u\n", r); //r++ = 6422284          +1 // 1 byte
r--;
printf("r-- = %u\n", r); //r-- = 6422283          -1 // restored to original value

return 0;
}
```

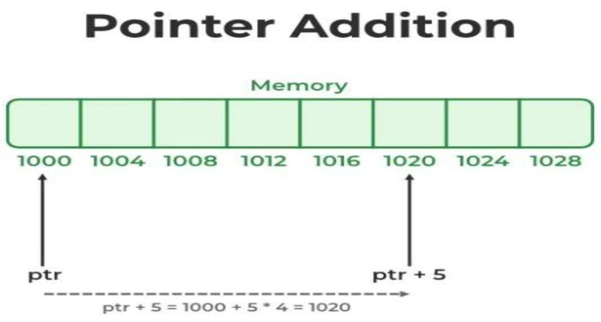
Output

```
p = 1441900792

p++ = 1441900796

p-- = 1441900792
```

```
q = 1441900796

q++ = 1441900800

q-- = 1441900796

r = 1441900791

r++ = 1441900792

r-- = 1441900791
```

Addition of Integer to Pointer
When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

For Example:
Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5,** then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020.**



Example of Addition of Integer to Pointer

```
// C program to illustrate pointer Addition
#include <stdio.h>
// Driver Code
int main()
{
        // Integer variable
        int N = 4;
        // Pointer to an integer
        int *ptr1, *ptr2;
        // Pointer stores the address of N
        ptr1 = &N;
        ptr2 = &N;
        printf("Pointer ptr2 before Addition: ");
        printf("%p \n", ptr2);
        // Addition of 3 to ptr2
        ptr2 = ptr2 + 3;
        printf("Pointer ptr2 after Addition: ");
        printf("%p \n", ptr2);
        return 0;
}
```

Output:

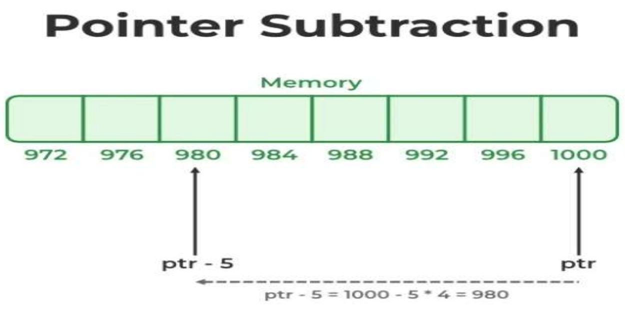
```
Pointer ptr2 before Addition: 0x7ffca373da9c

Pointer ptr2 after Addition: 0x7ffca373daa8
```

Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr – 5,** then, the final address stored in the ptr will be **ptr = 1000 – sizeof(int) * 5 = 980.**



Example of Subtraction of Integer from Pointer, below is the program to illustrate pointer Subtraction:

```
// Program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
    int x = 6; // Integer variable declaration
    int N = 4;
    // Pointer declaration
    int *ptr1, *ptr2;
    ptr1 = &N; // stores address of N
    ptr2 = &x; // stores address of x
    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
    // %p gives an hexa-decimal value,
    // We convert it into an unsigned int value by using %u

    // Subtraction of ptr2 and ptr1
    x = ptr1 - ptr2;

    // Print x to get the Increment
    // between ptr1 and ptr2
    printf("Subtraction of ptr1 "
           "& ptr2 is %d\n",
           x);
    return 0;
}
```

Output:

```
ptr1 = 2143484872, ptr2 = 2143484876
Subtraction of ptr1 & ptr2 is -1
```

Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1**.

Example:2

```
#include <stdio.h>
#include <stddef.h>
int main() {
    // Define an array of integers
    int arr[] = {10, 20, 30, 40, 50};

    // Define two pointers to elements within the array
    int *ptr1 = &arr[1]; // Points to the second element (20)
    int *ptr2 = &arr[4]; // Points to the fifth element (50)

    // Calculate the difference between the two pointers
    ptrdiff_t diff = ptr2 - ptr1;

    // Output the result
    printf("Pointer 1 points to value: %d\n", *ptr1);
    printf("Pointer 2 points to value: %d\n", *ptr2);
    printf("Difference between pointers: %td\n", diff);

    return 0;
}
```

OUTPUT

```
Pointer 1 points to value: 20
Pointer 2 points to value: 50
Difference between pointers: 3
```

Examples:C Program to swap two variables.

```
#include <stdio.h>

// Function prototypes
void swap(int *a, int *b);
void printValues(int a, int b);

int main() {
    // Define two integers
    int x = 5;
    int y = 10;

    // Print original values
    printf("Original values:\n");
    printValues(x, y);

    // Swap the values
    swap(&x, &y);

    // Print swapped values
    printf("Values after swap:\n");
    printValues(x, y);

    return 0;
}

// Function to swap two integers using pointers
void swap(int *a, int *b) {
    int temp;
    temp = *a; // Store value of *a in temp
    *a = *b;   // Assign value of *b to *a
    *b = temp; // Assign value of temp to *b
}

// Function to print the values of two integers
void printValues(int a, int b) {
    printf("a = %d, b = %d\n", a, b);
}
```

OUTPUT:

```
Original values:
a = 5, b = 10
Values after swap:
a = 10, b = 5
```

Comparison of Pointers
We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C **>, >=, <, <=, ==, !=.** It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

Example of Pointer Comparison

```
// Program to illustrate pointer comparison
#include <stdio.h>
int main()
{
   // declaring array
   int arr[5];
   // declaring pointer to array name
   int* ptr1 = &arr;
   // declaring pointer to first element
   int* ptr2 = &arr[0];
   if (ptr1 == ptr2) {
      printf("Pointer to Array Name and First Element "
           "are Equal.");
   }
   else {
      printf("Pointer to Array Name and First Element "
           "are not Equal.");
   }
   return 0;
}
```

Output

```
Pointer to Array Name and First Element are Equal.
```

4.11.3 Pointers and Functions

- Pointer as a function parameter is used to hold addresses of arguments passed during function call.
- This is also known as call by reference.
- When a function is called by reference any change made to the reference variable will effect the original variable.

Example

```
#include <stdio.h>
 void exchange(int *a, int *b);
 int main()
 {
int m = 10, n = 20;
 printf("m = %d\n", m);
 printf("n = %d\n\n", n);
swap(&m, &n);
printf("After Swapping:\n\n");
 printf("m = %d\n", m);
printf("n = %d", n); return 0;
}
void exchange (int *a, int *b)
 {
 int temp;
temp = *a;
*a = *b;
*b = temp;
 }
```

Output

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

4.11.4 Call by value

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Example

```
#include<stdio.h>
void change(int num) {
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
 int a = 10;
 int b = 20;
 printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b
in main
swap(a,b);
printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do
not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b =
10
}
```

Output

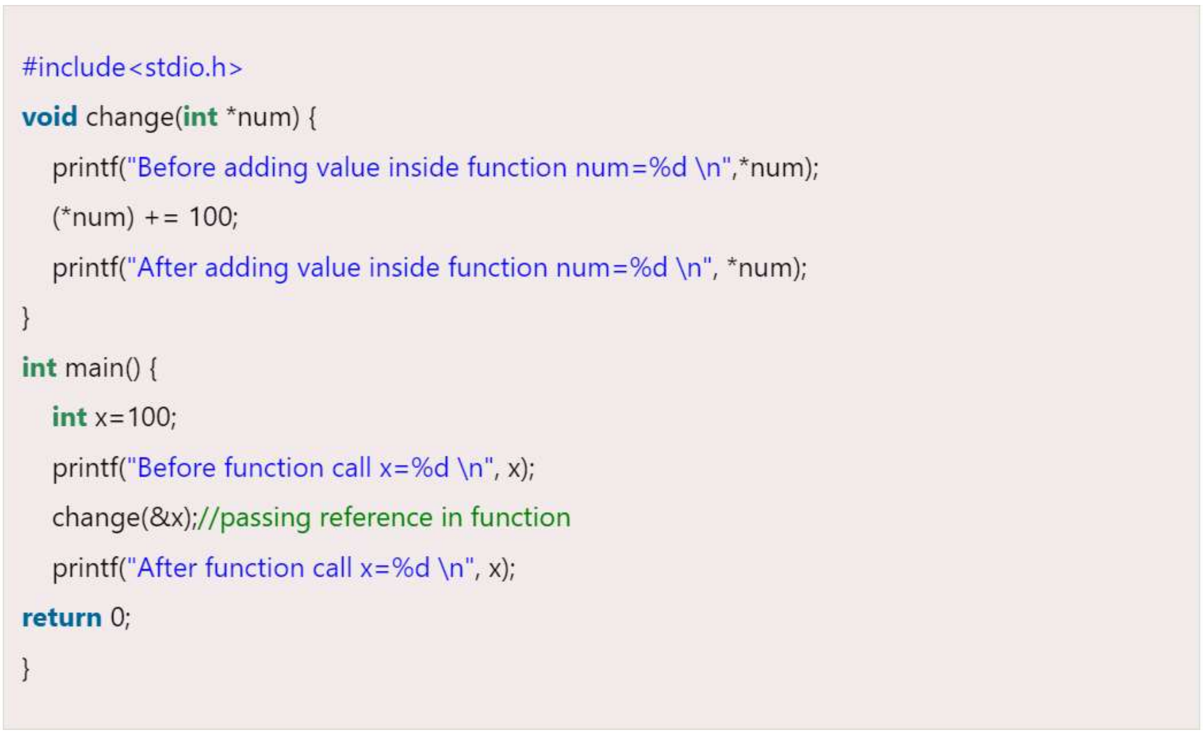
```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

4.11.5 Call by reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.



```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
    void swap(int *, int *); //prototype of the function
    int main()
    {
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a an
    d b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameter
    s do change in call by reference, a = 10, b = 20
    }
    void swap (int *a, int *b)
    {
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 2
    0, b = 10
    }
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

4.12 Pointers and Arrays

- The address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.
- &x[0] is equivalent to x. And, x[0] is equivalent to *x.
- Similarly, &x[1] is equivalent to x+1 and x[1] is equivalent to *(x+1).
- &x[2] is equivalent to x+2 and x[2] is equivalent to *(x+2).
- Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).

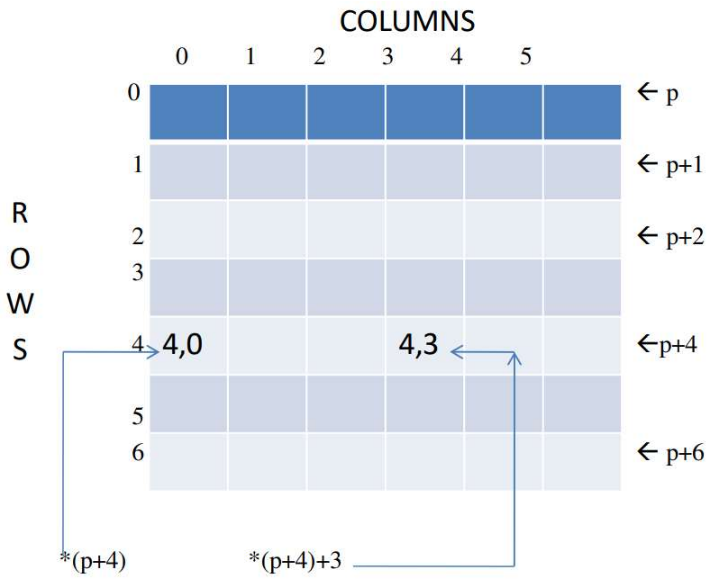
Example:

```
#include<stdio.h>
 int main()
{
 int i, x[20], sum = 0,n;
printf("Enter the value of n: ");
 scanf("%d",&n);
printf("Enter number one by one\n");
 for(i = 0; i < n; ++i)
 {
/* Equivalent to scanf("%d", &x[i]); */ scanf("%d", x+i);
// Equivalent to sum += x[i]
sum += *(x+i);
}
printf("Sum = %d", sum);
 return 0;
 }
```

Output

```
Enter the value of n: 5
 Enter number one by one
 5
10
15
20
25
Sum = 75
```

- Pointers can be used to manipulate two-dimensional arrays also.
- A two-dimensional array can be represented by the pointer expression as follows
- *(*(a+i)+j) or *(*(p+1)+j)



p ->pointer to first row

p+i ->pointer to ith row

*(p+i) ->pointer to first element in the ith row

*(p+i) +j ->pointer to jth element in the ith row

((p+i)+j) ->valu stored in the ith row and jth columns.

Example

```
#include
int main()
 {
 int arr[3][4] = { {11,22,33,44}, {55,66,77,88},{11,66,77,44}};
 int i, j;
for(i = 0; i < 3; i++)
 {
printf("Address of %d th array %u \n",i , *(arr + i));
for(j = 0; j < 4; j++)
{
printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
}
 printf("\n\n");
}
// signal to operating system program ran fine
return 0;
}
```

Output

```
arr[2][3]=44Address of 0 th array 2692284448
arr[0][0]=11
 arr[0][1]=22
 arr[0][2]=33
arr[0][3]=44
Address of 1 th array 2692284464
```

```
 arr[1][0]=55
 arr[1][1]=66
arr[1][3]=88
Address of 2 th array 2692284480
arr[2][0]=11
 arr[2][1]=66
 arr[2][2]=77
arr[2][3]=44
```

4.13 Arrays of Pointers

A pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in Programming when we want to point at multiple memory locations of a similar data type in program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Example

char name[4][25];

- The name is a table containing four names, each with maximum of 25 characters.
- The total storage requirements are 75 bytes.
- The individual strings will of equal lengths.

Example

```
#include <stdio.h>

int main() {
    // Define an array of pointers to strings
    const char *arrayOfStrings[] = {
        "Hello",
        "World",
        "This",
        "Is",
        "An",
        "Array",
        "Of",
        "Pointers"
    };

    // Calculate the number of elements in the array
    int numStrings = sizeof(arrayOfStrings) / sizeof(arrayOfStrings[0]);

    // Print each string using the array of pointers
    for (int i = 0; i < numStrings; i++) {
        printf("%s\n", arrayOfStrings[i]);
    }

    return 0;
}
```

Output:

```
Hello
World
This
```

Is

An

Array

Of

Pointers

4.14 Pointers and Structures

- We know that the name of an array stands for the address of its zero-th element.
- Also true for the names of arrays of structure variables.

Example

```
struct inventory
{
 int no;
 char name[30];
float price;
}
product[5], *ptr ;
```

- The name product represents the address of the zero-th element of the structure array.
- ptr is a pointer to data objects of the type struct inventory.
- The assignment

  ptr = product ;

  will assign the address of product [0] to ptr.
- Its member can be access

  ```
   ptr –>name ;
  ptr –> no ;
  ptr –> price;
  ```

  The symbol "–>" is called the arrow operator or member selection operator.
- When the pointer ptr is incremented by one (ptr++) :The value of ptr is actually increased by sizeof(inventory).
- It is made to point to the next record.

- We can also use the notation (*ptr).no;
- When using structure pointers, we should take care of operator precedence.
- Member operator "." has higher precedence than "*".
- ptr –> no and (*ptr).no mean the same thing.
- ptr.no will lead to error.
- The operator "–>" enjoys the highest priority among operators
- ++ptr –> no will increment roll, not ptr.
- (++ptr)→ –> no will do the intended thing.

Example

```
void main ()
{
struct book
 {
char name[25];
 char author[25];
 int edn;
 };
struct book b1 = { "Programming in C", "E Balagurusamy", 2 } ;
 struct book *ptr ; ptr = &b1 ;
 printf ( "\n%s %s edition %d ", b1.name, b1.author, b1.edn ) ;
printf ( "\n%s %s edition %d", ptr- >name, ptr->author, ptr->edn ) ;
 }
```

Output

```
 Programming in C E Balagurusamy edition 2
 Programming in C E Balagurusamy edition 2
```

4.15 Meaning of static and dynamic memory allocation

Static Memory Allocation

When the allocation of memory performs at the compile time, then it is known as static memory. In this, the memory is allocated for variables by the compiler.

OR

In static memory allocation whenever the program executes it fixes the size that the program is going to take, and it can't be changed further. So, the exact memory requirements must be known before. Allocation and deallocation of memory will be done by the compiler automatically. When everything is done at compile time (or) before run time, it is called static memory allocation.

Dynamic Memory Allocation

When the memory allocation is done at the execution or run time, then it is called dynamic memory allocation

OR

In Dynamic memory allocation size initialization and allocation are done by the programmer. It is managed and served with pointers that point to the newly allocated memory space in an area which we call the heap. Heap memory is unorganized and it is treated as a resource when you require the use of it if not release it. When everything is done during run time or execution time it is known as Dynamic memory allocation.

4.16 Memory Allocation Functions

Dynamic Memory allocation is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. free()
4. realloc()

1. malloc()

The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

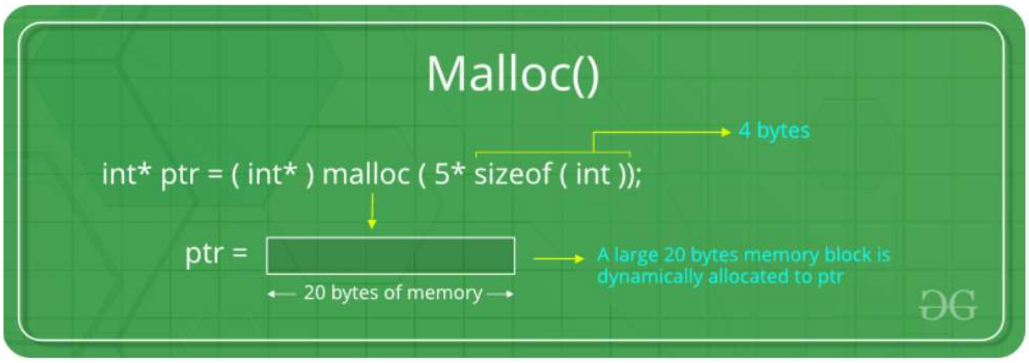
Syntax of malloc() in C

```
ptr = (cast-type*) malloc(byte-size)
```

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

If space is insufficient, allocation fails and returns a NULL pointer.

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

int main()
{

// This pointer will hold the
// base address of the block created
int* ptr;
int n, i;
```

```
// Get the number of elements for the array
printf("Enter number of elements:");
scanf("%d",&n);
printf("Entered number of elements: %d\n", n);

// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
}
else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
                ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
                printf("%d, ", ptr[i]);
        }
}

return 0;
```

```
}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

2. Calloc() method

1. **"calloc"** or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value '0'.
3. It has two parameters or arguments as compare to malloc().

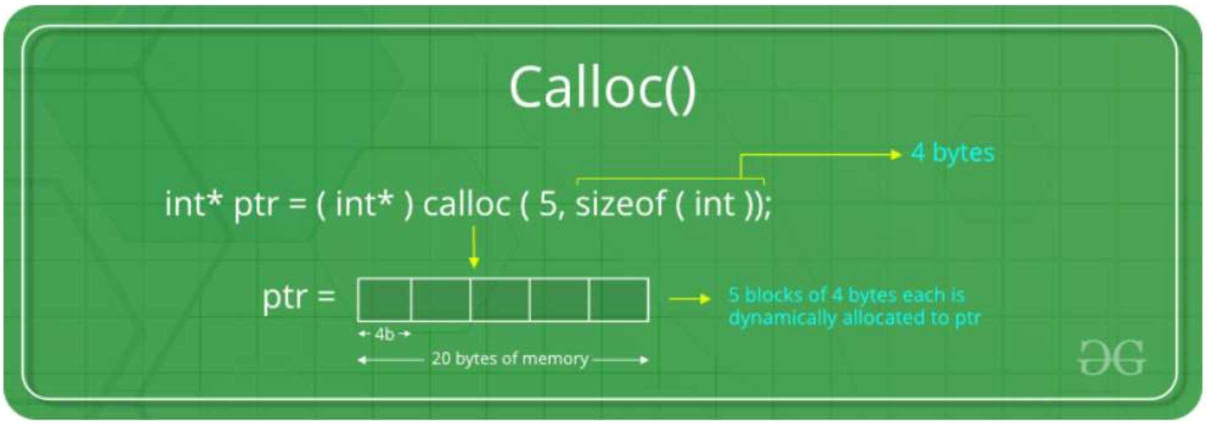
Syntax of calloc() in C

```
ptr = (cast-type*)calloc(n, element-size);
here, n is the no. of elements and element-size is the size of each element.
```

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.


Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory

If space is insufficient, allocation fails and returns a NULL pointer.

Example of calloc()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output

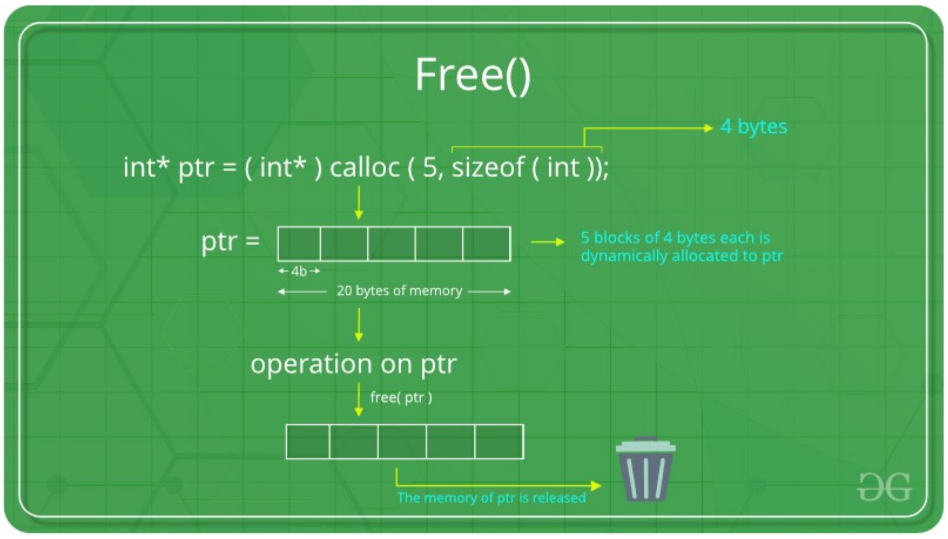
```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

3. free() method

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr
operation on ptr
free(ptr)
The memory of ptr is released

Example of free()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.
```

4.realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

```
ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.
```

If space is insufficient, allocation fails and returns a NULL pointer.

Example of realloc()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
```

```
int n, i;

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
```

```
        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using calloc.
```

```
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Macros: Definition—types of Macros—Creating and implementing user defined header files.

4.17 Macros

The macro in C language is known as the piece of code which can be replaced by the macro value. The macro is defined with the help of #define **preprocessor directive** and the macro doesn’t end with a semicolon (;). Macro is just a name given to certain values or expressions it doesn't point to any memory location.

Whenever the compiler encounters the macro, it replaces the macro name with the macro value. Two macros could not have the same name.

In C, we must use the #define directive, then the macro’s name and value, to define a macro. The following is the syntax to define a macro.

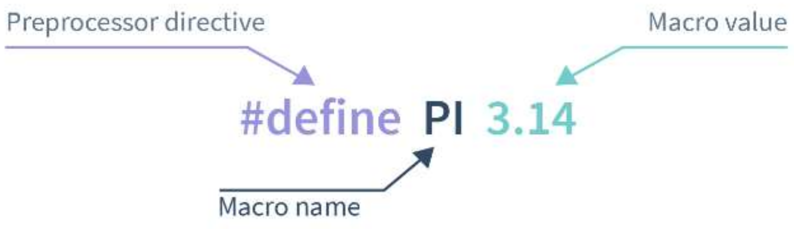
#define MACRO_NAME macro_value

For example, if you want to define a macro for the value of pi, you can write:

#define PI 3.14159

Here, we will have the three components:

1. #define - Preprocessor Directive
2. PI - Macro Name
3. 3.14 - Macro Value



Example:

```
#include<stdio.h>
// This is macro definition
#define PI 3.14
void main()
{
    // declaration and initialization of radius
    int radius = 5;
    // declaration and calculating the area
    float area = PI * (radius*radius);
    // Printing the area of circle
    printf("Area of circle is %f", area);
}
```

Output:

```
Area of circle is 78.500000
```

In the above code example, we wrote a program to calculate the area of a circle with the help of a given radius. We defined the PI constant as a macro with the value 3.14

4.17.1 Types of Macros

The macro in C language is classified based on the different types of values that it replaces.

The different types of macros are

Object like Macros

An object like macros in C programming is simply the macros that get replaced by certain values or segments of code.

In the above example, in the introduction, we saw the macro with the name PI and value 3.14 it is an example of an object like macros.

```
// Examples of object like macros in C language

#define MAX 100

#define MIN 1

#define GRAVITY 9.8

#define NAME "Scaler"

#define TRUE 1

#define FALSE 0
```

Function Like Macros

In the function like macros are very similar to the actual function in C programming.

We can pass the arguments with the macro name and perform the actions in the code segment.

In macros, there is no type checking of arguments so we can use it as an advantage to pass different datatypes in same macros in C language.

Let's consider the following code example which is the modification of the previous code -

```
#include <stdio.h>



//object like macro
```

```
#define PI 3.14



// function like macro

#define Area(r) (PI*(r*r))

void main()

{

    // declaration and initialization of radius

    float radius = 2.5;

    // declaring and assigning the value to area

    float area = Area(radius);

    // Printing the area of circle

    printf("Area of circle is %f\n", area);

    // Using radius as int data type

    int radiusInt = 5;

    printf("Area of circle is %f", Area(radiusInt));

}
```

Output:

```
Area of circle is 19.625000
Area of circle is 78.500000
```

See in the above code, we added the function-like macro at line no. 7.

The macro name is Area which takes the argument r as a radius that we have passed at line no. 15 and line no. 22.

At the time of preprocessing the value Area(radius) gets replaced with the processed macro value and it is assigned to the area variable.

At line no. 15, we passed the value of radius as a float and at line no. 22 we passed the value of radius as a float of type integer. So, macros gave us the advantage to use the same macro for different datatypes because there is no type checking of arguments.

Chain Like Macros

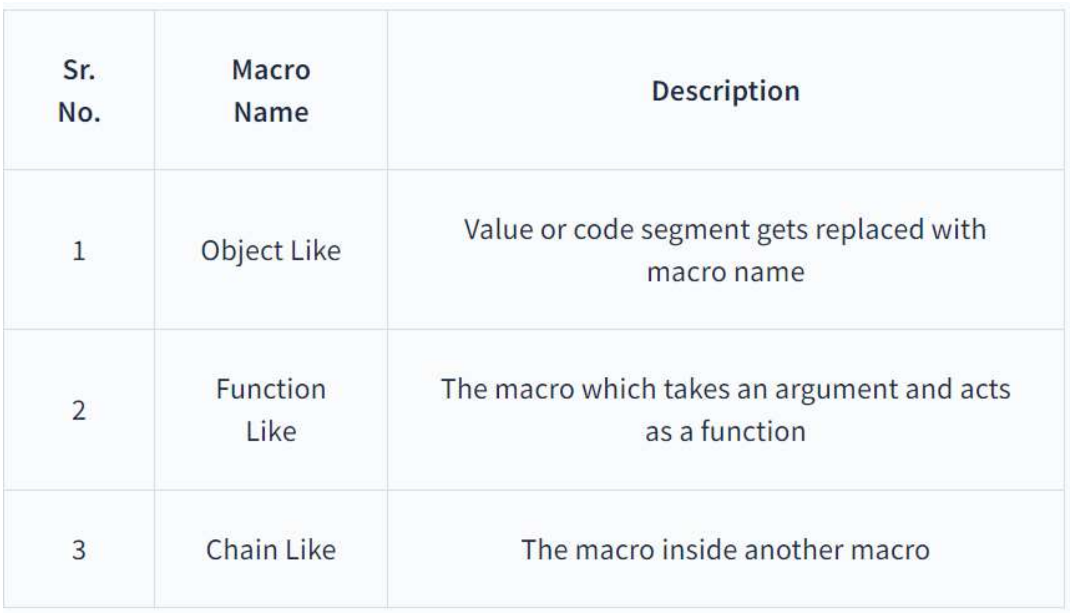
When we use one macro inside another macro, then it is known as the chain-like macro.

We already saw the example of a chain-like macro in the above code example where we used the PI in Area. Let's discuss it a little more -

```
#define PI 3.14
#define Area(r) (PI*(r*r))
```

In the above code snippet, we can see we used the object like macro PI inside the function like macro Area.

Here first parent macro gets expanded i.e. the functional macro Area and then the child macro gets expanded i.e. PI. This way when we use one macro inside another macro, it is called a chain macro.



Sr. No.	Macro Name	Description
1	Object Like	Value or code segment gets replaced with macro name
2	Function Like	The macro which takes an argument and acts as a function
3	Chain Like	The macro inside another macro

4.17.2 Creating and Implementing User Defined Header Files

Creating and implementing user-defined header files in C using macros is a fundamental aspect of modular programming. Header files typically contain function prototypes, macro definitions, type declarations, and other declarations required for the implementation of a module. Here's a step-by-step guide on how to create and implement user-defined header files in C using macros:

1. Create the Header File:

Start by creating a header file with a `.h` extension. This file will contain declarations and macro definitions that you want to share across multiple source files.

~~~
// myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H
// Macro definition
#define SQUARE(x) ((x) * (x))
// Function prototype
int add(int a, int b);
#endif
```
~~~

In this example, `myheader.h` declares a macro `SQUARE(x)` and a function prototype `add`.

2. Implement the Functions:

Implement the functions declared in the header file in a separate source file (`.c` file).

```
// myfunctions.c
#include "myheader.h"
int add(int a, int b) {
    return a + b;
}
```

Ensure that you include the header file `myheader.h` in the source file that contains the implementations.

3. Include the Header File:

In any source file where you want to use the declarations and macros defined in `myheader.h`, include the header file at the beginning.

```
// main.c
#include <stdio.h>
#include "myheader.h"
int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    int squareResult = SQUARE(5);
    printf("Square Result: %d\n", squareResult);
    return 0;
}
```

4. Compile:

When compiling your program, ensure that you include all relevant source files.

```
gcc main.c myfunctions.c -o myprogram
```

This command compiles `main.c` and `myfunctions.c` into an executable named `myprogram`.

5. Execute:

Run the compiled program.

```
./myprogram
```

The output is printed in `main.c`, which uses the functions and macros defined in `myheader.h`. By following these steps, you can create and implement user-defined header files in C using macros effectively. This modular approach enhances code organization, readability, and reusability.