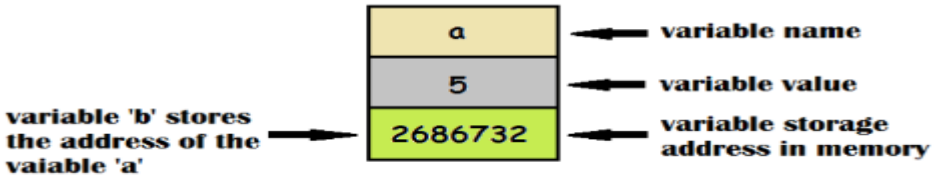
Module 4

Pointers and Files in C

Pointers

Pointers are variables that contain the memory address of another variable. It enables the programmer to directly access the memory location and hence the value stored at it.

For example a variable 'a' stores the value '**5**' and another variable '**b**' stores the address value of variable '**a**' which is '**2686732**'. The address allocated to the variable '**a**' can be any random memory address



C language makes the use of certain operators to make it possible for the programmer to access the memory locations of a variable.

The & operator

The scanf () contains the '**ampersand**' operator or '**&**' which refers to the '***address of***" the variable. To directly access the memory location of the variable we can use this operator.

```
#include<stdio.h>
int main()
{
        int a=5;
        printf("The address of %d is %u ", a, &a);
        return 0;
}
```

The above code will return the output as **The address of 5 is 2686732**

% u is a format specifier for fetching values from the address of a variable having an unsigned decimal integer stored in memory.

The * operator(Indirection)

The * operator or value at address operator returns the value stored at the particular address. It is also known as the '**indirection**' operator.

```
#include <stdio.h>

int main()

{

        int a=5;

        printf("The value of a is %d\n", *(&a));
```

```
        printf("The address of a is %u", &a);
        return 0;
}
```

The output of the above code is

The value of a is 5
The address of a is 2644516420

(&a)** yields the same result as '**a**' because the '**' operator returns the value at the address given by '**&a**'.

Declaration of Pointers

As we have already seen that **pointers** are variables so like other variables pointers too need to be declared. The '*****' operator is used while declaring a pointer.
The syntax of the declaration is given below:

Datatype *pointer_variable ;

Examples:-

```
char *j;
int *i;
double *d;
float *f;
```

Here we see that the variable ***j** has been declared as *char*. The reason behind this is that ***j** refers to the value at the address given by variable '**j**'. So here the value at address '**j**' is a character value, hence, it is declared as a *char*.

Assigning Value to a Pointer Variable

When we declare a pointer, it contains garbage value, which means it could be pointing anywhere in the memory. **Pointer Initialization** is the process of assigning the address of a variable to a pointer. In C language, the address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

Eg:

```
#include <stdio.h>
int main()
{
        int a = 5, *b;
        b = &a;
   printf("The value of a is: %d\n",a);
   printf("The value of a is: %d\n",*b);
   printf("The address of a is :%u\n",&a);
   printf("The value of b is: %u\n", b);
   return 0;
}
```

The output of the above code is

```
The value of a is: 5
The value of a is: 5
The address of a is :2738236268
The value of b is: 2738236268
```

Pointer operations

Pointers are variables that contain the memory address of another variable. Since an address in a memory is a numeric value we can perform arithmetic operations on the pointer values.

1. Incrementing/decrementing a pointer
2. Addition or subtraction
3. Subtraction of one pointer from other
4. Comparison of two pointers.

Incrementing or decrementing

Any pointer variable when incremented points to the *next memory location of its type*.
Eg:-

```
#include <stdio.h>
int main()
{
  int a = 5, *x;
  char b = 'z', *y;
  x = &a;
  printf("x= %d\n", x);
  x++;
  printf("x++= %d\n", x);
  x--;
  printf("x--= %d\n", x);
  y = &b;
  printf("y= %d\n", y);

  printf("y++= %d\n", y);
  y--;
  printf("y--= %d\n", y);
  return 0;
}
```

Output

```
x= -1136672652
x++= -1136672648
x--= -1136672652
y= -1136672653
y++= -1136672652
y--= -1136672653
```

Addition or Subtraction

Addition or **subtraction** of a constant number to a pointer is allowed. The result is similar to the increment or decrement operator with the only difference being the increase or decrease in the memory location by the constant number given
Eg:

```
#include <stdio.h>
int main()
{
  int a = 5, *x;
  char b = 'z', *y;
  x = &a;
  printf("x= %d\n", x);
  printf("x+3= %d\n", x + 3);
  printf("x-2= %d\n", x - 2);
  y = &b;
  printf("y= %d\n", y);
  printf("y+3= %d\n", y + 3);
  printf("y-2= %d\n", y - 2);
  return 0;
}
```

Output

```
x= -1413962812
x+3= -1413962800
x-2= -1413962820
y= -1413962813
y+3= -1413962810
y-2= -1413962815
```

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

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1016)** are subtracted. The difference between address is 16 bytes. Since the size of int is 2 bytes, therefore the **increment between ptr1 and ptr2** is given by **(16/2) = 8**.

```
int main()
{
   int x;
   int N = 4;
   int *ptr1, *ptr2;
   ptr1 = &N;
   ptr2 = &N;
   ptr2 = ptr2 + 3;
   x = ptr2 - ptr1;
   printf("Subtraction of ptr1 & ptr2 is %d\n", x);
  return 0;
}
```

Output

```
Subtraction of ptr1 & ptr2 is 3
```

Comparison of pointers of the same type

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

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

Example program

```
#include <stdio.h>
 int main()
{
            int num1=5,num2=6,num3=5;
            int *p1,*p2,*p3;
          p1=&num1;
            p2=&num2;
            p3=&num3;
            if(*p1<*p2)
            {
                        printf("\n%d less than %d",*p1,*p2);
            }
            if(*p2>*p1)
            {
                        printf("\n%d greater than %d",*p2,*p1);
            }
             if(*p3==*p1)
             {
                         printf("\nBoth the values are equal");
             }
             if(*p3!=*p2)
            {
                         printf("\nBoth the values are not equal");
            }
            return 0;
}
```

Output

```
5 less than 6
6 greater than 5
Both the values are equal
Both the values are not equal
```

Program - To add two numbers suing pointers

```
#include <stdio.h>
int main()
{
  int first, second, *p, *q, sum;
  printf("Enter two integers to add\n");
  scanf("%d%d", &first, &second);
  p = &first;
  q = &second;
```

```
  sum = *p + *q;
  printf("Sum of the numbers = %d\n", sum);
  return 0;
}
```

PASSING POINTERS TO FUNCTION

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Example Program

```
#include <stdio.h>
void salaryhike(int  *var, int b)
{
   *var = *var+b;
}
int main()
{
   int salary=0, bonus=0;
   printf("Enter the employee current salary:");
   scanf("%d", &salary);
   printf("Enter bonus:");
   scanf("%d", &bonus);
   salaryhike(&salary, bonus);
   printf("Final salary: %d", salary);
   return 0;
}
```

Output

```
Enter the employee current salary:10000
Enter bonus:2000
Final salary: 12000
```

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

Swapping two numbers using pointers

```
#include <stdio.h>
void swapnum(int *num1, int *num2)
{
  int tempnum;
  tempnum = *num1;
  *num1 = *num2;
  *num2 = tempnum;
}
int main( )
```

```
{
  int v1,v2 ;
  printf(“Enter two numbers:”);
  scanf (“%d%d”, &v1,&v2);
  printf("Before swapping:");
  printf("\nValue of v1 is: %d", v1);
  printf("\nValue of v2 is: %d", v2);
  swapnum( &v1, &v2 );
  printf("\nAfter swapping:");
  printf("\nValue of v1 is: %d", v1);
  printf("\nValue of v2 is: %d", v2);
}
```

Program -store n elements in an array and print the elements using pointer.

```
#include <stdio.h>
int main()
{
   int arr[20];
   int N, i;
   int * ptr = arr;
   printf("Enter size of array: ");
   scanf("%d", &N);
   printf("Enter elements in array:\n");
   for (i = 0; i < N; i++)
   {
      scanf("%d", ptr);
      ptr++;
   }
   ptr = arr;
   printf("Array elements: ");
   for (i = 0; i < N; i++)
   {
      printf("%d, ", *ptr);
      ptr++;
   }
   return 0;
}
```

Output

```
Enter size of array: 5
Enter elements in array:
2
8
4
6
9
Array elements: 2, 8, 4, 6, 9,
```

Programming in c

What is a Pointer to a Pointer?

A pointer to a pointer is a variable that stores the address of another pointer. This can be useful for dynamic memory allocation, passing pointers to functions, and managing arrays of pointers.

Declaration

You declare a pointer to a pointer by using two asterisks (**). For example:

```
int **ptr;
```

Example Usage

1. Dynamic Memory Allocation

Pointers to pointers are often used in dynamic memory allocation, especially for creating multi-dimensional arrays.

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

int main() {
    int rows = 3, cols = 4;
    int **array;

    // Allocate memory for rows
    array = (int **)malloc(rows * sizeof(int *));

    // Allocate memory for columns in each row
    for (int i = 0; i < rows; i++) {
        array[i] = (int *)malloc(cols * sizeof(int));
    }

    // Assign values and print the array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            array[i][j] = i + j;
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory
    for (int i = 0; i < rows; i++) {
        free(array[i]);
    }
    free(array);

    return 0;
}
```

Programming in c

Key Points

- **Declaration**: int **ptr;
- **Usage**: Dynamic memory allocation, passing pointers to functions.
- **Memory Management**: Always free dynamically allocated memory to avoid memory leaks.

Array of pointer

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

Syntax:

pointer_type *array_name [array_size];
Here,

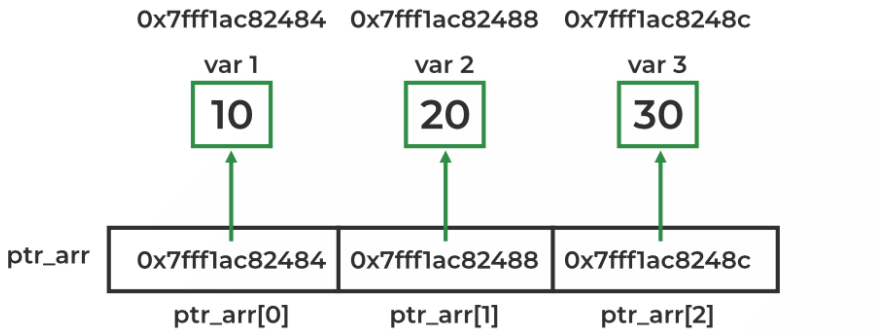
pointer_type: Type of data the pointer is pointing to.
array_name: Name of the array of pointers.
array_size: Size of the array of pointers.

// C program to demonstrate the use of array of pointers

```
#include <stdio.h>

int main()
{
   // declaring some temp variables
   int var1 = 10;
   int var2 = 20;
   int var3 = 30;

   // array of pointers to integers
   int* ptr_arr[3] = { &var1, &var2, &var3 };

   // traversing using loop
   for (int i = 0; i < 3; i++) {
      printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
   }

   return 0;
}
```

Output

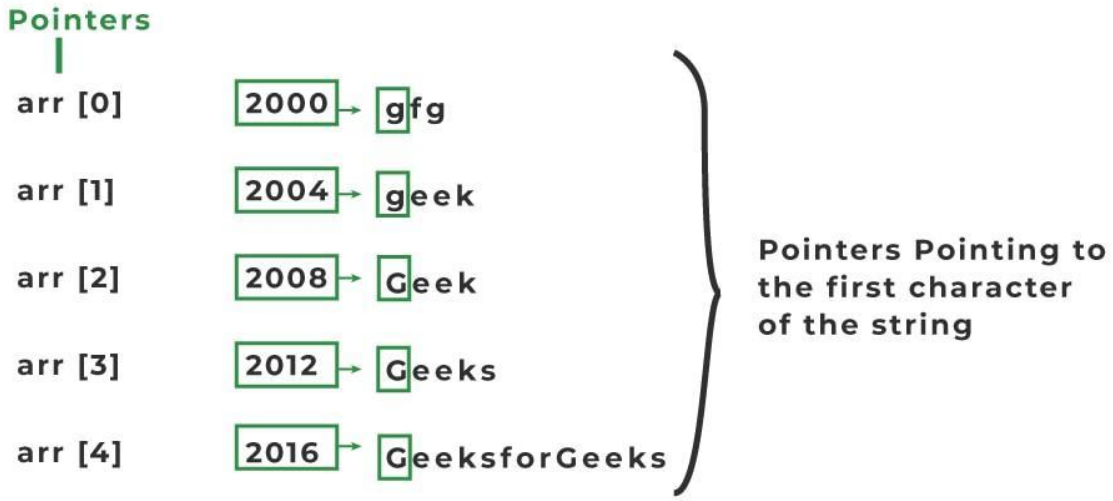
Value of var1: 10 Address: 0x7fff1ac82484
Value of var2: 20 Address: 0x7fff1ac82488
Value of var3: 30 Address: 0x7fff1ac8248c



Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.
Syntax:
char *array_name [array_size];
After that, we can assign a string of any length to these pointers.

```
char* arr[5]
    = { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }
```



Dynamic memory allocation

When we declare a variable or an array of any data type the space occupied by them in the system's memory remains constant throughout the execution of the program. Sometimes the constant space allocated at the compile-time may fall short, and to increase the space during run-time we came through the concept of **Dynamic Memory Allocation**. The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

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

There are two types of memory in our machine, one is Static Memory and another one is Dynamic Memory; both the memory are managed by our Operating System. Our operating system helps us in the allocation and deallocation of memory blocks either during compile-time or during the run-time of our program. When the memory is allocated during compile-time it is stored in the Static Memory and it is known as Static Memory Allocation, and when the memory is allocated during run-time it is stored in the Dynamic Memory and it is known as Dynamic Memory Allocation.

4 components in system memory

1. Stack Segment (Static Memory)
2. Global Variables Segment (Static Memory)
3. Instructions / Text Segment (Static Memory)
4. Heap Segment (Dynamic Memory)

The amount of memory allocated for *Stack, Global Variables, and Instructions / Text* during compile-time is invariable and cannot be reused until the program execution finishes. However, the *Heap* segment of the memory can be used at run-time and can be expanded until the system's memory exhausts.

malloc()

malloc() is a method in C which is used to allocate a memory block in the heap section of the memory of some specified size (in bytes) during the run-time of a C program. It is a library function present in the <stdlib.h> header file.

Syntax:-

(Cast-datatype *)malloc(size in bytes)

Eg:- int *ptr;

ptr=(int *)mallac(sizeof(int));

Example Program

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{

   int* ptr;
   int n, i;
   printf("Enter number of elements:");
   scanf("%d",&n);
   printf("Entered number of elements: %d\n", n);
   ptr = (int*)malloc(n * sizeof(int));
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else
         {
      printf("Memory successfully allocated using malloc.\n");
      for (i = 0; i < n; ++i)
      {
         ptr[i] = i + 1;
      }
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i)
    {
         printf("%d, ", ptr[i]);
      }
   }

   return 0;
}
```

calloc() method

“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:

It initializes each block with a default value '0'.
It has two parameters or arguments as compare to malloc().

Syntax: -ptr = (cast-type*)calloc(n, element-size);

Eg:- **ptr = (float*) calloc(25, sizeof(float));** This statement allocates contiguous space in memory for 25 elements each with the size of the float.

Example Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int* ptr;
   int n, i;
   n = 5;
   printf("Enter number of elements: %d\n", n);
   ptr = (int*)calloc(n, sizeof(int));
   if (ptr == NULL)
   {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else
   {
      printf("Memory successfully allocated using calloc.\n");
      for (i = 0; i < n; ++i)
      {
         ptr[i] = i + 1;
      }
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i)
      {
         printf("%d, ", ptr[i]);
      }
   }
    return 0;
}
```

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:- free(ptr)

Example Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, *ptr1;
   int n, i;
   n = 5;
   printf("Enter number of elements: %d\n", n);
   ptr = (int*)malloc(n * sizeof(int));
   ptr1 = (int*)calloc(n, sizeof(int));
   if (ptr == NULL || ptr1 == NULL)
   {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else
   {
      printf("Memory successfully allocated using malloc.\n");
      free(ptr);
      printf("Malloc Memory successfully freed.\n");
      printf("\nMemory successfully allocated using calloc.\n");
      free(ptr1);
      printf("Calloc Memory successfully freed.\n");
   }
    return 0;
}
```

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:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.

Example Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int* ptr;
   int n, i;
   n = 5;
```

```
    printf("Enter number of elements: %d\n", n);
    ptr = (int*)calloc(n, sizeof(int));
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
         printf("Memory successfully allocated using calloc.\n");
         for (i = 0; i < n; ++i)
         {
            ptr[i] = i + 1;
         }
         printf("The elements of the array are: ");
        for (i = 0; i < n; ++i)
        {
            printf("%d, ", ptr[i]);
        }
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);
        ptr = realloc(ptr, n * sizeof(int));
        printf("Memory successfully re-allocated using realloc.\n");
        for (i = 5; i < n; ++i)
        {
            ptr[i] = i + 1;
        }
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i)
        {
            printf("%d, ", ptr[i]);
        }
        free(ptr);
    }
     return 0;
}
```

Relationship of arrays and pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include<stdio.h>
int main()
{
   int x[4];
   int i;
   for(i = 0; i < 4; ++i)
   {
      printf("&x[%d] = %p\n", i, &x[i]);
```

```
  }
  printf("Address of array x: %p", x);
  return 0;
}
```

Output of the above program is

```
&x[0] = 0x7ffe5286b450
&x[1] = 0x7ffe5286b454
&x[2] = 0x7ffe5286b458
&x[3] = 0x7ffe5286b45c
Address of array x: 0x7ffe5286b450
```

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, 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. from the above example, it is clear that &x[0] is equivalent to x and x[0] is equivalent to *x.

Structure and pointers

Pointer to structure holds the add of the entire structure.

It is used to create complex data structures such as linked lists, trees, graphs and so on.

The members of the structure can be accessed using a special operator called as an arrow operator (->).

Declaration :- To declare a pointer of structure use the following syntax

Syntax:- struct structurename *ptr;

To access the pointer to a structure as **ptr->membername;**

Eg:- Program to use pointer to store a structure variable

```
#include<stdio.h>
struct student{
  int sno;
  char sname[30];
  float marks;
};
int main ( ){
  struct student s;
  struct student *st;
  printf("enter sno, sname, marks:");
  scanf ("%d%s%f", & s.sno, s.sname, &s. marks);
  st = &s;
  printf ("details of the student are\n");
  printf ("Number = %d\n", st ->sno);
  printf ("name = %s\n", st->sname);
  printf ("marks =%f\n", st ->marks);
```

```
  return 0;
}
```

Output

```
enter sno, sname, marks:1
Anu
34
details of the student are
Number = 1
name = Anu
marks =34.000000
```

FILE HANDLING IN C

So far the operations using C program are done on a prompt / terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store the fetched information in a file. Different operations that can be performed on a file are:

1. Creation of a new file (**fopen with attributes as "a" or "a+" or "w" or "w+")**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgets**)
4. Writing to a file (**fprintf or fputs**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

The text in the brackets denotes the functions used for performing those operation.

Opening or creating file

For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

- **"r" –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
- **"w" –** Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"a" –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"r+" –** Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
- **"w+" –** Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.
- **"a+" –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

```
FILE *filePointer;
```

So, the file can be opened as

```
filePointer = fopen("fileName.txt", "w")
```

Reading from a file

The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.
Eg:-

```
FILE * filePointer;

filePointer = fopen(“fileName.txt”, “r”);

fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

Writing a file –:

The file write operations can be performed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :
Eg:-

```
FILE *filePointer ;

filePointer = fopen(“fileName.txt”, “w”);

fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);
```

Closing a file –:

After every successful file operations, you must always close a file. For closing a file, you have to use fclose function. The snippet for closing a file is given as

Eg:-

```
FILE *filePointer ;

filePointer= fopen(“fileName.txt”, “w”);

---------- Some file Operations -------

fclose(filePointer);
```

Write content into a file

```
#include <stdio.h>

#include <stdlib.h>


int main()

{

  int num;

  FILE *fptr;
```

```
  fptr = fopen("C:\\program.txt","w");

  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }

  printf("Enter num: ");
  scanf("%d",&num);

  fprintf(fptr,"%d",num);
  fclose(fptr);
}
```

Read a content from the file

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

int main()
{
  int num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.txt","r")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
      exit(1);
  }
```

```
   fscanf(fptr,"%d", &num);

   printf("Value of n=%d", num);
   fclose(fptr);

   return 0;
}

   return 0;
}
```

fseek() in file

As the name suggests, fseek() seeks the cursor to the given record in the file.

Syntax of fseek()

fseek(FILE * stream, long int offset, int whence);

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different whence in fseek()

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.

Different whence in fseek()

Whence	Meaning
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

ftell Function

The ftell function is used to obtain the current value of the file position indicator for the stream pointed to by file. This is useful for determining the current position in a file.

Syntax: long ftell(FILE *stream);

Example:

```
#include <stdio.h>


int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return -1;
    }

    // Move the file pointer to the end
    fseek(file, 0, SEEK_END);
    long position = ftell(file);
    printf("The file size is %ld bytes.\n", position);

    fclose(file);
    return 0;
}
```

fread Function

The fread function reads data from the given stream into the array pointed to by ptr.

Syntax:

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

Example:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "rb");
    if (file == NULL) {
        perror("Error opening file");
        return -1;
    }

    char buffer[100];
    size_t bytesRead = fread(buffer, sizeof(char), sizeof(buffer) - 1, file);
    buffer[bytesRead] = '\0'; // Null-terminate the string

    printf("Read %zu bytes: %s\n", bytesRead, buffer);

    fclose(file);
    return 0;
}
```

fwrite Function

The fwrite function writes data from the array pointed to by ptr to the given stream.

Syntax:

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

Example:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "wb");
    if (file == NULL) {
```

```
        perror("Error opening file");
        return -1;
    }

    const char *text = "Hello, World!";
    size_t bytesWritten = fwrite(text, sizeof(char), strlen(text), file);

    printf("Wrote %zu bytes.\n", bytesWritten);

    fclose(file);
    return 0;
}
```

These functions are essential for file manipulation in C, allowing you to read from, write to, and determine the position within files.