



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV – PROBLEM SOLVING WITH C and C++ SCSA1102

UNIT 4

POINTERS AND FILE PROCESSING

POINTERS AND FILE PROCESSING: Pointers: Introduction, Arrays Using Pointers – Structures Using Pointers – Functions Using Pointer, Dynamic Memory Allocation, Storage Classes, File Handling in 'C' Algorithms: Swap elements using Call by Reference – Sorting Arrays using pointers- Finding sum of array elements using Dynamic Memory Allocation.

POINTERS

- A Pointer in C language is a variable which holds the address of another variable of same data type.
- Pointers are used to access memory and manipulate the address.
- Pointers are one of the most distinct and exciting features of C language.
- It provides power and flexibility to the language.

ADDRESS IN C

- Whenever a variable is defined in C language, a memory location is assigned for it, in which its value will be stored using the & symbol.
- If var is the name of the variable, then &var will give its address.

EXAMPLE

```
#include<stdio.h>

void main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
    printf("Memory address of the variable var is: %x\n", &var);
}
```

OUTPUT:

```
Value of the variable var is: 7
Memory address of the variable var is: bcc7a00
```

- You must have also seen in the function scanf(), we mention &var to take user input for any variable var.

```
scanf("%d", &var);
```

- This is used to store the user inputted value to the address of the variable var.

CONCEPT OF POINTERS

```
int a = 10;
```

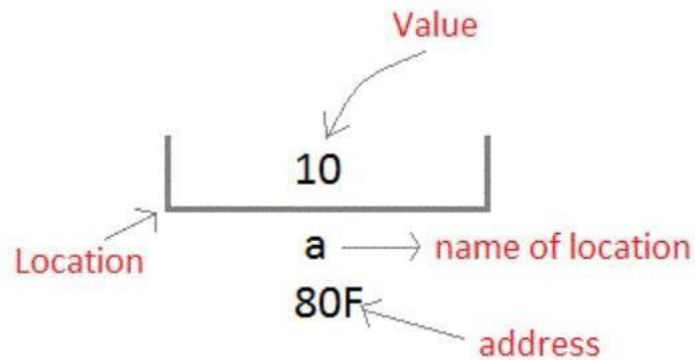


Fig.1. Concept of pointer

- Whenever a variable is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value.(Fig.1)
- This location has its own address number
- Let us assume that system has allocated memory location 80F for a variable a. (above mentioned)

POINTER VARIABLES

- The memory addresses are also just numbers, they can also be assigned to some other variable.
- The variables which are used to hold memory addresses are called Pointer variables.
- A pointer variable is therefore nothing but a variable which holds an address of some other variable.

BENEFITS OF USING POINTERS

- Pointers are more efficient in handling Arrays and Structures.
- Pointers allow references to function and thereby help in passing of function as arguments to other functions.
- It reduces length of the program and its execution time as well.
- It allows C language to support Dynamic Memory management.

DECLARING, INITIALIZING AND USING A POINTER VARIABLE IN C

DECLARING OF C POINTER

Syntax

```
datatype *pointer_name;
```

- Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing.
- void type pointer works with all data types, but is not often used.

Examples

```
int *ip      // pointer to integer variable
float *fp;    // pointer to float variable
double *dp;   // pointer to double variable
char *cp;     // pointer to char variable
```

INITIALIZATION OF C POINTER VARIABLE

- Pointer initialization is the process of assigning address of a variable to a pointer variable.
- Pointer variable can only contain address of a variable of the same data type.
- In C language 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.

```
#include<stdio.h>

void main()
{
    int a = 10;
    int *ptr;    //pointer declaration
    ptr = &a;    //pointer initialization
}
```

NULL Pointer

- The variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable.
- A pointer which is assigned a NULL value is called a NULL pointer.

```
#include <stdio.h>

int main()
{
    int *ptr = NULL;
    return 0;
}
```

EXAMPLE FOR POINTERS

```
#include <stdio.h>

int main()
{
    int i = 10;    // normal integer variable storing value 10
    int *a;        // since '*' is used, hence its a pointer variable
    a = &i;
    /*
        below, address of variable 'i', which is stored
        by a pointer variable 'a' is displayed
    */
    printf("Address of variable i is %u\n", a);
    /*
        below, '*a' is read as 'value at a'
        which is 10
    */
    printf("Value at the address, which is stored by pointer variable a is %d\n", *a);
    return 0;
}
```

Output

Address of variable i is 2686728 (The address may vary)

Value at an address, which is stored by pointer variable a is 10

POINTERS USING ARRAYS

Example 1

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = {10, 100, 200};
    int i;

    for (i = 0; i < MAX; i++) {
        printf("Value of var[%d] = %d\n", i, var[i] );
    }

    return 0;
}
```

Output

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Example 2

```
#include <stdio.h>

const int MAX = 4;

int main () {

    char *names[] = {
        "ABCD",
        "EFGH",
        "IJKL",
        "MNOP"
    };

    int i = 0;

    for ( i = 0; i < MAX; i++) {
        printf("Value of names[%d] = %s\n", i, names[i] );
    }

    return 0;
}
```

Output

Value of names[0] = ABCD Value
of names[1] = EFGH
Value of names[2] = IJKL
Value of names[3] = MNOP

POINTERS USING STRUCTURES

Example

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

struct student
{
    int id;
    char name[30];
    float percentage;
};

int main()
{
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;

    ptr = &record1;

    printf("Records of STUDENT1: \n");
    printf("  Id is: %d \n", ptr->id);
    printf("  Name is: %s \n", ptr->name);
    printf("  Percentage is: %f \n\n", ptr->percentage);

    return 0;
}
```

Output

```
Records of STUDENT1:
Id is: 1
Name is: Raju
Percentage is: 90.500000
```

POINTERS USING FUNCTIONS

Example 1

```
#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
```

Example 2

Swapping two numbers using Pointers

- This is one of the most popular example that shows how to swap numbers using call by reference.

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

    tempnum = *num1;
    *num1 = *num2;
    *num2 = tempnum;
}
int main( )
{
    int v1 = 11, v2 = 77 ;
    printf("Before swapping:");
    printf("\nValue of v1 is: %d", v1);
    printf("\nValue of v2 is: %d", v2);

    /*calling swap function*/
    swapnum( &v1, &v2 );

    printf("\nAfter swapping:");
    printf("\nValue of v1 is: %d", v1);
    printf("\nValue of v2 is: %d", v2);
}
```

Output

```
Before  swapping:
Value of v1 is: 11
Value of v2 is:  77

After  swapping:
Value of v1 is:  77
Value of v2 is:  11
```

DYNAMIC MEMORY ALLOCATION

- The process of allocating memory at runtime is known as dynamic memory allocation.
- Library routines known as memory management functions are used for allocating and freeing memory during execution of a program.
- These functions are defined in `stdlib.h` header file.
- Dynamic memory allocation in c language is possible by 4 functions
 - `malloc()`
 - `calloc()`
 - `realloc()`
 - `free()`

Function	Description
<code>malloc()</code>	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
<code>calloc()</code>	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
<code>free</code>	releases previously allocated memory
<code>realloc</code>	modify the size of previously allocated space

OR

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by <code>malloc()</code> or <code>calloc()</code> functions.
free()	frees the dynamically allocated memory.

MEMORY ALLOCATION PROCESS

- Global variables, static variables and program instructions get their memory in permanent storage area whereas local variables are stored in a memory area called Stack.(Fig.2)
- The memory space between these two regions is known as Heap area.
- This region is used for dynamic memory allocation during execution of the program.
- The size of heap keeps changing.

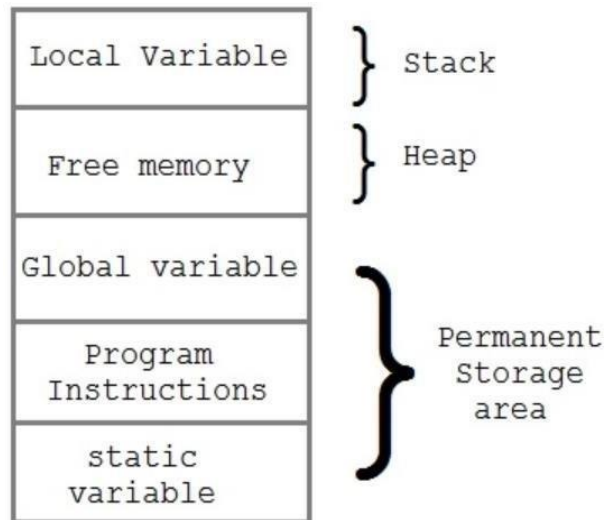


Fig.2.Memory Allocation Process

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

DIFFERENCE BETWEEN `calloc()` and `malloc()`

<code>calloc()</code>	<code>malloc()</code>
<code>calloc()</code> initializes the allocated memory with 0 value.	<code>malloc()</code> initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax : (cast_type *)calloc(blocks , size_of_block);	Syntax : (cast_type *)malloc(Size_in_bytes);

malloc() function in C

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.

Syntax of malloc()

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

Example

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.

Syntax

ptr = (cast-type*) calloc (number, byte-size)

Example

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```


Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Syntax

```
ptr = realloc (ptr, new-size)
```

Example

```
int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory: ");
    for(i = 0; i < n2; ++i)
        printf("%u\n", ptr + i);

    free(ptr);

    return 0;
}
```

Output

```
Enter size: 2
Addresses of previously allocated memory:26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:26855472
26855476
26855480
26855484
```

free() function in C

- The memory occupied by malloc() or calloc() functions must be released by calling free() function.
- Otherwise, it will consume memory until program exit.

Syntax

free (ptr)

SMART POINTER

- Smart Pointer is used to manage the lifetimes of dynamically allocated objects.
- They ensure proper destruction of dynamically allocated objects.
- Smart pointers are defined in the memory header file.
- Smart pointers are built-in pointers, we don't have to worry about deleting them, they are automatically deleted.
- By using the delete keyword we can delete the memory allocated

Example

```
S_ptr *ptr = new S_ptr();
ptr->action();
delete ptr;
```

STORAGE CLASSES IN C

- In C language, each variable has a storage class which decides the following things:
- Scope- where the value of the variable would be available inside a program.
- Default initial value- if we do not explicitly initialize that variable, what will be its default initial value.
- Lifetime of that variable - for how long will that variable exist.

Storage classes are most often used in C programming

- Automatic variables
- External variables
- Static variables
- Register variables

AUTOMATIC VARIABLES

Keyword : auto

- SCOPE: Variable defined with auto storage class are local to the function block inside which they are defined.
- Default initial Value: Any random value i.e garbage value.
- Lifetime: Till the end of the function/method block where the variable is defined.

AUTOMATIC VARIABLES

- A variable declared inside a function without any storage class specification, is by default an automatic variable.
- They are created when a function is called and are destroyed automatically when the function's execution is completed.
- Automatic variables can also be called local variables because they are local to a function.
- By default they are assigned garbage value by the compiler

Example

```
#include<stdio.h>
void main()
{
    int detail;
    // or
    auto int details;    //Both are same
}
```

EXTERNAL OR GLOBAL VARIABLE

Keyword : extern

- **Scope:** Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.
- **Default initial value:** 0(zero).
- **Lifetime:** Till the program doesn't finish its execution, you can access global variables.

GLOBAL VARIABLE

- A variable that is declared outside any function is a Global Variable.
- Global variables remain available throughout the program execution.
- By default, initial value of the Global variable is 0 (zero).

Example

```
#include<stdio.h>
int number;    // global variable
void main()
{
    number = 10;
    printf("I am in main function. My value is %d\n", number);
    fun1();    //function calling, discussed in next topic
    fun2();    //function calling, discussed in next topic
}
/* This is function 1 */
fun1()
{
    number = 20;
```

```

    printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */
fun2()
{
    printf("\nI am in function fun2. My value is %d", number);
}

```

Output

```

I am in function main. My value is 10
I am in function fun1. My value is 20
I am in function fun2. My value is 20

```

extern KEYWORD

- The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else.
- The extern declaration does not allocate storage for variables.(Fig.3)

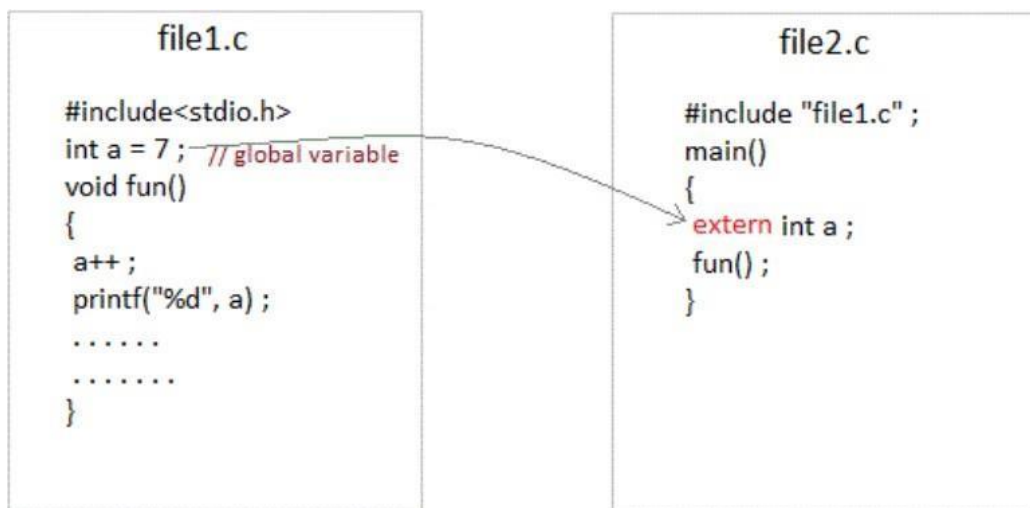


Fig.3.extern Keyword

Problem when extern is not used

```

int main()
{
    a = 10;    //Error: cannot find definition of variable 'a'
    printf("%d", a);
}

```

Example :

using extern in same file

```
int main()
{
    extern int x;    //informs the compiler that it is defined somewhere else
    x = 10;
    printf("%d", x);
}

int x;              //Global variable x
```

STATIC VARIABLES

Keyword : static

- **Scope:** Local to the block in which the variable is defined
- **Default initial value:** 0(Zero).
- **Lifetime:** Till the whole program doesn't finish its execution

STATIC VARIABLES

- A static variable tells the compiler to persist/save the variable until the end of program.
- Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program.
- A static variable can either be internal or external depending upon the place of declaration.
- Scope of internal static variable remains inside the function in which it is defined.
- External static variables remain restricted to scope of file in which they are declared

Example

```
#include<stdio.h>
void test();    //Function declaration (discussed in next topic)
int main()
{
    test();
    test();
    test();
}
```

```
void test()
{
    static int a = 0;        //a static variable
    a = a + 1;
    printf("%d\t",a);
}
```

Output

1 2 3

REGISTER VARIABLE

- **Scope:** Local to the function in which it is declared.
- **Default initial value:** Any random value i.e garbage value
- **Lifetime:** Till the end of function/method block, in which the variable is defined.

Keyword : register

Syntax

```
register int number;
```

REGISTER VARIABLE

- Register variables inform the compiler to store the variable in CPU register instead of memory.
- Register variables have faster accessibility than a normal variable.

FILE HANDLING IN C

- In any programming language it is vital to learn file handling techniques.
- Many applications will at some point involve accessing folders and files on the hard drive.
- In C, a stream is associated with a file.
- Special functions have been designed for handling file operations.
- The header file `stdio.h` is required for using these functions

The basic file operations are

- `fopen` - open a file- specify how its opened (read/write) and type (binary/text)
- `fclose` - close an opened file
- `fread` - read from a file
- `fwrite` - write to a file
- `fseek/fsetpos` - move a file pointer to somewhere in a file.
- `ftell/fgetpos` - tell you where the file pointer is located.

File Open Modes

Mode	Meaning
r	Open text file in read mode <ul style="list-style-type: none">• If file exists, the marker is positioned at beginning.• If file doesn't exist, error returned.
w	Open text file in write mode <ul style="list-style-type: none">• If file exists, it is erased.• If file doesn't exist, it is created.
a	Open text file in append mode <ul style="list-style-type: none">• If file exists, the marker is positioned at end.• If file doesn't exist, it is created.

Opening file

- The function `fopen()` for opening a file.
- Once this is done one can read or write to the file using the `fread()` or `fwrite()` functions
- The `fclose()` function is used to explicitly close any opened file.

SYNTAX

```
FILE*fopen("filename" , "mode");
```


EXAMPLE

```
#include <stdio.h>
main ()
{
FILE *fp;
fp = fopen("data.txt", "r");
if (fp == NULL) {
printf("File does not exist,
please check!\n");
}
fclose(fp);
}
```

fopen()

- This function accepts two arguments as strings.
- The first argument denotes the name of the file to be opened and the second signifies the mode in which the file is to be opened.
- The second argument can be any of the following:

File Mode	Description
r	Open a text file for reading
w	Create a text file for writing, if it exists, it is overwritten.
a	Open a text file and append text to the end of the file.
rb	Open a binary file for reading
wb	Create a binary file for writing, if it exists, it is overwritten.
ab	Open a binary file and append data to the end of the file.

fclose()

- The fclose() function is used for closing opened files.
- The only argument it accepts is the file pointer.
- If a program terminates, it automatically closes all opened files.
- fclose(p1);

fscanf() and fprintf()

- The functions fprintf() and fscanf() are similar to printf() and scanf() except that these functions operate on files and require one additional and first argument to be a file pointer

fprintf()

Example

```
fprintf()

Syntax:
    fprintf (fp,"string",variables);

Example:
    int i = 12;
    float x = 2.356;
    char ch = 's';
    FILE *fp;
    fp=fopen("out.txt","w");
    fprintf (fp, "%d %f %c", i, x, ch);
```

fscanf()

```
fscanf()

Syntax:
    fscanf (fp,"string",identifiers);

Example:
    FILE *fp;
    Fp=fopen("input.txt","r");
    int i;
    fscanf (fp,"%d",i);
```

EXAMPLE

```
#include <stdio.h>
main ()
{
    FILE *fp;
    float total;
    fp = fopen("data.txt", "w+");
```

```

if (fp == NULL) {
printf("data.txt does not exist, please check!\n");
exit (1);
}
fprintf(fp,          100);
fscanf(fp, "%f", &total);
fclose(fp);
printf("Value of total is %f\n", total);
}

```

getc() and putc()

- The functions getc() and putc() are equivalent to getchar() and putchar() functions
- Input and Output, except that these functions require an argument which is the file pointer.
- Function getc() reads a single character from the file which has previously been opened using a function like fopen().
- Function putc() does the opposite, it writes a character to the file identified by its second argument.
- The format of both functions is as follows

```

getc(in_file); putc(c,
out_file);

```

getc()

getc()

Syntax:

```
identifier = getc (file pointer);
```

Example:

```

FILE *fp;
fp=fopen("input.txt","r");
char ch;
ch = getc (fp);

```

putc()

putc()

write a single character to the output file, pointed to by fp.

Example:

```
FILE *fp;  
char ch;  
putc (ch,fp);
```

Example

Using getc() and putc()

```
#include <stdio.h>  
main ()  
{  
    char in_file[30], out_file[30];  
    FILE *fpin, *fpout;  
    int c;  
    printf("This program copies the source file to the destination file \n\n");  
    printf("Enter name of the source file :");  
    scanf("%30s", in_file);  
    printf("Enter name of the destination file :");  
    scanf("%30s", out_file);  
    if((fpin=fopen(in_file, "r")) == NULL)  
        printf("Error could not open source file for reading\n"); else  
    if ((fpout=fopen(out_file, "w")) == NULL) printf("Error  
could not open destination file for reading\n"); else  
    {  
        while((c =getc(fpin)) != EOF)  
            putc(c, fpout);  
        printf("Destination file has been copied\n");  
    }  
}
```

BINARY STREAMINPUT AND OUTPUT

- The functions `fread()` and `fwrite()` are a somewhat complex file handling functions used for reading or writing chunks of data containing NULL characters ('\0') terminating strings
- The function prototype of `fread()` and `fwrite()` is as below

```
size_t fread (void *ptr, size_t sz, size_t n, FILE *fp)
```

```
size_t fwrite (const void *ptr, size_t sz, size_t n, FILE *fp);
```

- The return type of `fread()` is `size_t` which is the number of items read.
- Function `fread()` reads it as a stream of bytes and advances the file pointer by the number of bytes read.
- Function `fwrite()` works similarly, it writes `n` objects of `sz` bytes long from a location pointed to by `ptr`, to a file pointed to by `fp`, and returns the number of items written to `fp`

Description for `fread()` and `fwrite()` declaration (above mentioned)

`fread()`

fread ()

Declaration:

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

Remarks:

`fread` reads a specified number of equal-sized data items from an input stream into a block.

`ptr` = Points to a block into which data is read
`size` = Length of each item read, in bytes
`n` = Number of items read
`stream` = file pointer

Example for fread()

Example

Example:

```
#include <stdio.h>
int main()
{
    FILE *f;
    char buffer[11];
    if (f = fopen("fred.txt", "r"))
    {
        fread(buffer, 1, 10, f);
        buffer[10] = 0;
        fclose(f);
        printf("first 10 characters of the file:\n%s\n", buffer);
    }
    return 0;
}
```

fwrite()

fwrite()

Declaration:

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

Remarks:

fwrite appends a specified number of equal-sized data items to an output file.

ptr = Pointer to any object; the data written begins at ptr
size = Length of each item of data
n = Number of data items to be appended
stream = file pointer

Example for fwrite()

Example

Example:

```
#include <stdio.h>
int main()
{
    char a[10]={'1','2','3','4','5','6','7','8','9','a'};
    FILE *fs;
    fs=fopen("Project.txt","w");
    fwrite(a,1,10,fs);
    fclose(fs);
    return 0;
}
```

Example:

Using fread() and fwrite() function:

```
#include <stdio.h>
#define MAX_SIZE 1024
main ()
{
    FILE *fp, *gp;
    char buf[MAX_SIZE];
    int i, total = 0;
    if ((fp = fopen("data1.txt", "r")) == NULL)
        printf("Error in data1.txt file \n");
    else if ((gp=fopen("data2.txt", "w")) == NULL)
        printf("Error in data2.txt file \n");
    else
    {
        while(i=fread(buf, 1, MAX_SIZE, fp))
        {
            fwrite(buf, 1, MAX_SIZE, gp);
            total +=i;
        }
        printf("Total is %d\n", total);
    }
    fclose(fp);
    fclose(gp);
}
```

ftell()

- Functions ftell() and fseek() are important in a program performing file manipulations.
- Function ftell() returns the current position of the file pointer in a stream.
- The return value is 0 or a positive integer indicating the byte offset from the beginning of an open file
- A return value of -1 indicates an error.
- Prototype of this function is as shown below :

```
long int ftell(FILE *fp);
```

ftell()

```
offset = ftell( file pointer );
```

"ftell" returns the current position for input or output on the file
#include <stdio.h>

```
int main(void)
{
    FILE *stream;
    stream = fopen("MYFILE.TXT", "w");
    fprintf(stream, "This is a test");
    printf("The file pointer is at byte %ld\n", ftell(stream));
    fclose(stream);
    return 0;
}
```

fseek()

- This function positions the next I/O operation on an open stream to a new position relative to the current position.

```
int fseek(FILE *fp, long int offset, int origin);
```

- Here fp is the file pointer of the stream on which I/O operations are carried on, offset is the number of bytes to skip over
- The offset can be either positive or negative, denoting forward or backward movement in the file.

fseek()

This function sets the file position indicator for the stream pointed to by stream or you can say it seeks a specified place within a file and modify it.

SEEK_SET	Seeks from beginning of file
SEEK_CUR	Seeks from current position
SEEK_END	Seeks from end of file

Example:

```
#include <stdio.h>
int main()
{
    FILE * f;
    f = fopen("myfile.txt", "w");
    fputs("Hello World", f);
    fseek(f, 6, SEEK_SET);    SEEK_CUR, SEEK_END
    fputs(" India", f);
    fclose(f);
    return o;
}
```

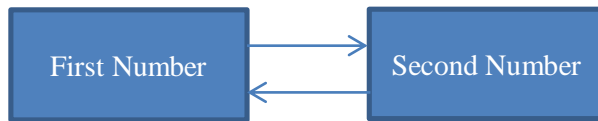
EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
char buffer[11];
int position;
main ()
{
    FILE *file_ptr;
    int num;
    if ((file_ptr = fopen("test_file", "w+f 10"))== NULL)
    {
        printf("Error opening test_file \n");
        exit(1);
    }
    fputs("1111111111", file_ptr);
    fputs("2222222222", file_ptr);
    if ( (position = fseek(file_ptr, 10, SEEK_SET)) != 0)
    {
        printf("Error in seek operation: errno \n");
        exit(1);
    }
    num = 11;
    fgets(buffer, num, file_ptr);
    printf("The record is %s\n", buffer);
    fclose(file_ptr);
}
```

OUTPUT: The record is 2222222222.

Algorithms:

Swap elements using Call by Reference



Since we want the local variables of main to be modified by swap function, we must use pointers in C.

The idea is simple

1. Assign x to a temp variable : `temp = *x`
2. Assign y to x : `*x = *y`
3. Assign temp to y : `*y = temp`

Program:

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

```
void swap(int*, int*);
```

```
int main()
```

```
{  
    int x, y;
```

```
    printf("Enter the value of x and y\n");  
    scanf("%d%d",&x,&y);
```

```
    printf("Before Swapping\nx = %d\ny = %d\n", x, y);
```

```
    swap(&x, &y);
```

```
    printf("After Swapping\nx = %d\ny = %d\n", x, y);
```

```
    return 0;  
}
```

```
void swap(int *a, int *b)
```

```
{  
    int temp;
```

```
    temp = *b;  
    *b = *a;  
    *a = temp;
```

```
}
```

Sorting Arrays using pointers

Given an array of size n, the task is to sort this array using pointers in C.

Examples:

Input: n = 5, arr[] = {0, 23, 14, 12, 9}

Output: {0, 9, 12, 14, 23}

Input: n = 3, arr[] = {7, 0, 2}

Output: {0, 2, 7}

Approach:

The array can be fetched with the help of pointers with the pointer variable pointing to the base address of the array. Hence in order to sort the array using pointers, we need to access the elements of the array using **(pointer + index)** format.(Fig.4)

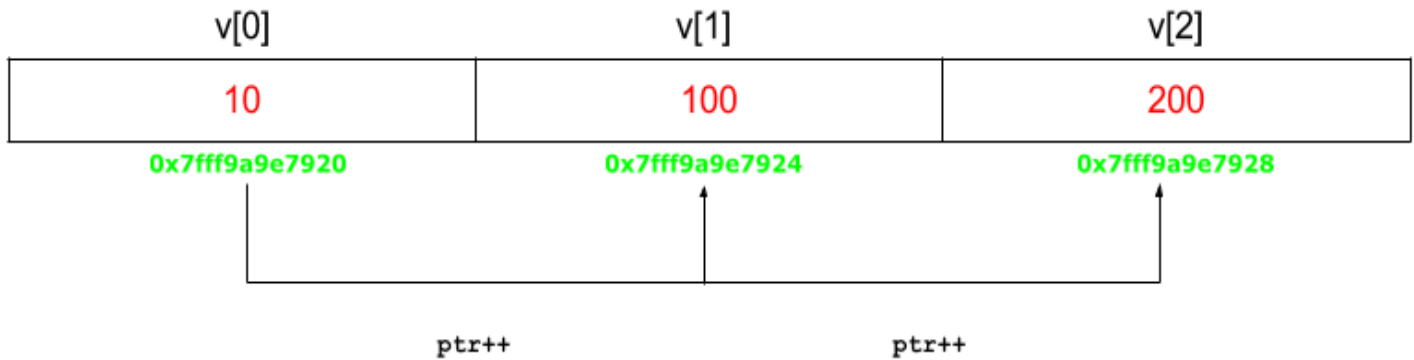


Fig.4.Sorting Arrays using Pointers

Program:

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

```
// Function to sort the numbers using pointers
```

```
void sort(int n, int* ptr)
```

```
{
```

```
    int i, j, t;
```

```
    // Sort the numbers using pointers
```

```
    for (i = 0; i < n; i++) {
```

```
        for (j = i + 1; j < n; j++) {
```

```
            if (*(ptr + j) < *(ptr + i)) {
```

```
                t = *(ptr + i);
```

```
                *(ptr + i) = *(ptr + j);
```

```
                *(ptr + j) = t;
```

```
            }
```

```
        }
```

```
    }
```

```
    // print the numbers
```

```
    for (i = 0; i < n; i++)
```

```
        printf("%d ", *(ptr + i));
```

```

}

// Driver code
int main()
{
    int n = 5;
    int arr[] = { 0, 23, 14, 12, 9 };

    sort(n, arr);

    return 0;
}

```

Finding sum of array elements using Dynamic Memory Allocation:

Dynamic Memory Allocation Example: In this C program, we will declare memory for array elements (limit will be at run time) using malloc(), read element and print the sum of all elements along with the entered elements.

This program is an example of Dynamic Memory Allocation, here we are declaring memory for N array elements at run time using malloc() - which is used to declare memory for N blocks at run time, we will read N elements, will print them and also print the sum of all elements.

Example: C program to find sum of array elements using Dynamic Memory Allocation

```

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

int main()
{
    int *ptr;      //declaration of integer pointer
    int limit;     //to store array limit
    int i;         //loop counter
    int sum;       //to store sum of all elements

    printf("Enter limit of the array: ");
    scanf("%d",&limit);

    //declare memory dynamically
    ptr=(int*)malloc(limit*sizeof(int));

    //read array elements
    for(i=0;i<limit;i++)
    {
        printf("Enter element %02d: ",i+1);
        scanf("%d",(ptr+i));
    }

    //print array elements
    printf("\nEntered array elements are:\n");
    for(i=0;i<limit;i++)
    {

```

```

        printf("%d\n",*(ptr+i));
    }

    //calculate sum of all elements
    sum=0;        //assign 0 to replace garbage value
    for(i=0;i<limit;i++)
    {
        sum+=*(ptr+i);
    }
    printf("Sum of array elements is: %d\n",sum);

    //free memory
    free(ptr);    //hey, don't forget to free dynamically allocated memory.

    return 0;
}

```