KIET Group of Institutions, Ghaziabad

Department of Computer Applications

(An ISO – 9001: 2015 Certified & 'A' Grade accredited Institution by NAAC)

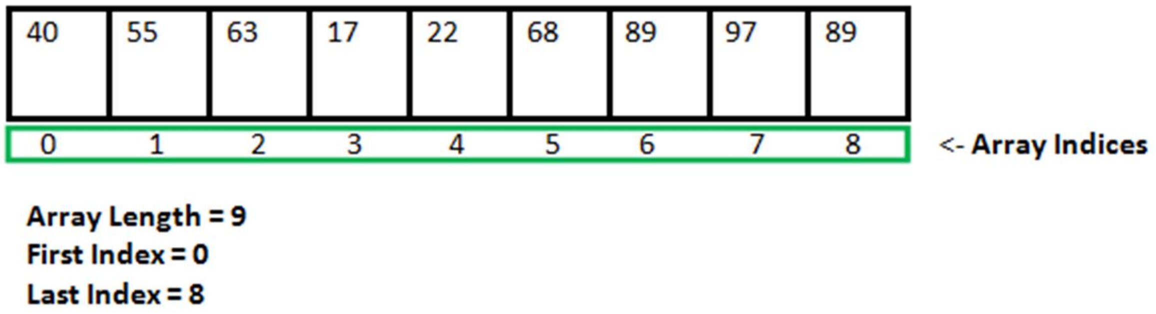
Problem Solving Using C

KCA 102: Session 2020-21

Unit 5	**Lecture 1**	**Dynamic memory allocation**

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

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

4. realloc()

Let's look at each of them in greater detail.

1. C malloc() method

"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 initializes each block with default garbage value.

Syntax:

int a;

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

1. int *p =(int *) malloc(2)
2. float *Q = (float *)malloc(sizeof(int))

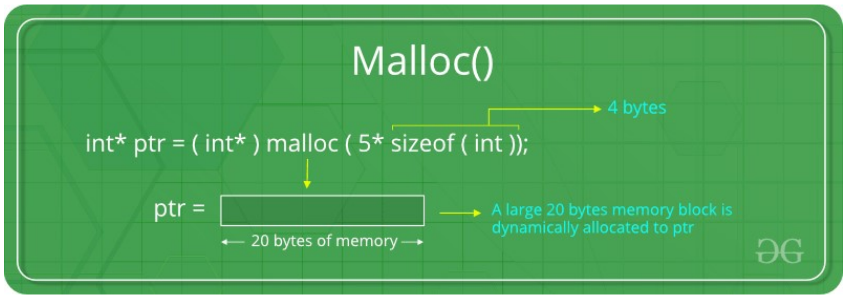
Struct student *s = (struct student *)malloc(sizeof(struct student));

Int *x = (int *)malloc(10*sizeof(int)); //2 byte or 4 byte

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.

Example:

```
#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 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:

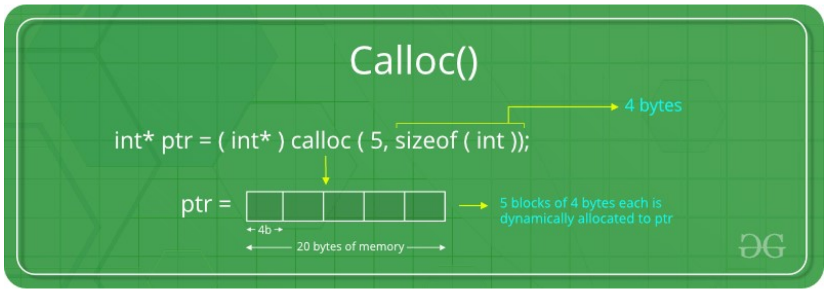
```
Enter number of elements: 5
```

```
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

2. C 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 initializes each block with a default value '0'.
Syntax:
ptr = (cast-type*)calloc(n, element-size);

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

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

Example:

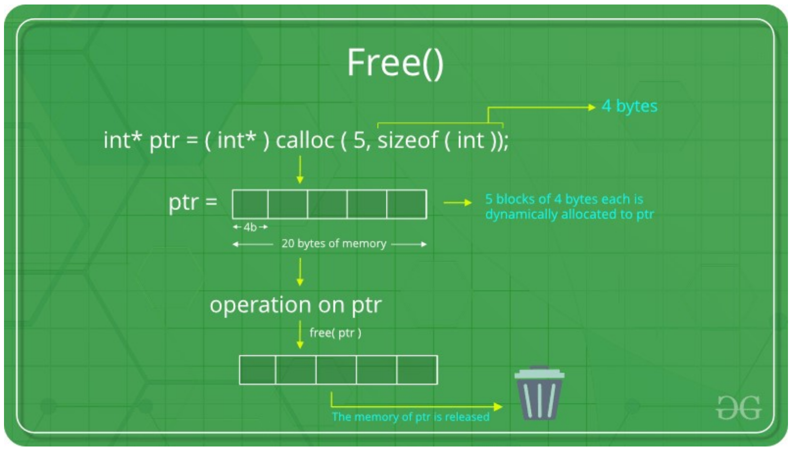
```
#include <stdio.h>
#include <stdlib.h>

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. C 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);


Free()
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
operation on ptr
free(ptr)
The memory of ptr is released

Example:

```
#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.
```

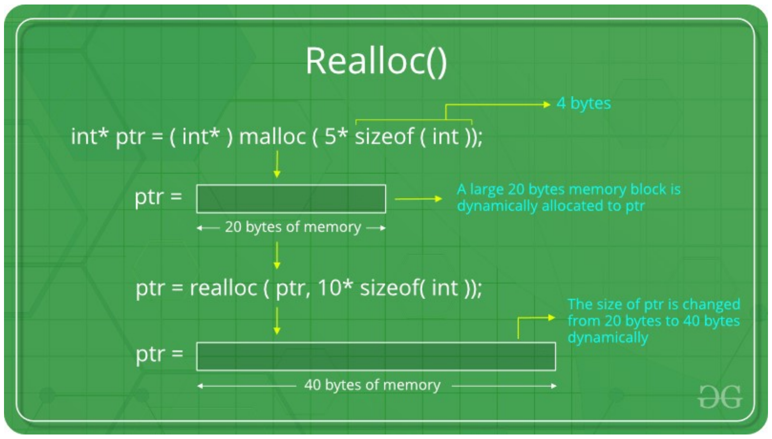
4. **C realloc() method**

"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 default garbage value.

Syntax:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

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

Example:

```
#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 = 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,
```

Problem Solving Using C
KCA 102: Session 2020-21

Unit 5	Lecture 2	File Handling

C File management

A File can be used to store a large volume of persistent data. Like many other languages 'C' provides following file management functions,

1. Creation of a file
2. Opening a file
3. Reading a file
4. Writing to a file
5. Closing a file
6. Rename
7. Delete

Following are the most important file management functions available in 'C,'

function	purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file //writing
fscanf ()	Reading a block data from a file // reading
getc ()	Reads a single character from a file
putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location

ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

How to Create a File

Whenever you want to work with a file, the first step is to create a file. A file is nothing but space in a memory where data is stored.

To create a file in a 'C' program following syntax is used,

```
FILE *fp;
fp = fopen ("file_name", "mode");
```

In the above syntax, the file is a data structure which is defined in the standard library.

fopen is a standard function which is used to open a file.

- If the file is not present on the system, then it is created and then opened.
- If a file is already present on the system, then it is directly opened using this function.

fp is a file pointer which points to the type file.

Whenever you open or create a file, you have to specify what you are going to do with the file. A file in 'C' programming can be created or opened for reading/writing purposes. A mode is used to specify whether you want to open a file for any of the below-given purposes. Following are the different types of modes in 'C' programming which can be used while working with a file.

File Mode	**Description**
r	Open a file for reading. If a file is in reading mode, then no data is deleted if a file is already present on a system.
w	Open a file for writing. If a file is in writing mode, then a new file is created if a file doesn't exist at all. If a file is already present on a system, then all the data inside the file is truncated, and it is opened for writing purposes.
a	Open a file in append mode. If a file is in append mode, then the file is opened. The content within the file doesn't change.

r+	open for reading and writing from beginning
w+	open for reading and writing, overwriting a file
a+	open for reading and writing, appending to file

In the given syntax, the filename and the mode are specified as strings hence they must always be enclosed within double quotes.

Example:

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

Output:

File is created in the same folder where you have saved your code.

You can specify the path where you want to create your file

```
#include <stdio.h>
int main() {
FILE *fp;
fp  = fopen ("D://data.txt", "w");
}
```

How to Close a file

One should always close a file whenever the operations on file are over. It means the contents and links to the file are terminated. This prevents accidental damage to the file.

'C' provides the fclose function to perform file closing operation. The syntax of fclose is as follows,

```
fclose (file_pointer);
```

Example:

```
FILE *fp;
fp  = fopen ("data.txt", "r");
fclose (fp);
```

The fclose function takes a file pointer as an argument. The file associated with the file pointer is then closed with the help of fclose function. It returns 0 if close was successful and EOF (end of file) if there is an error has occurred while file closing.

After closing the file, the same file pointer can also be used with other files.

In 'C' programming, files are automatically close when the program is terminated. Closing a file manually by writing fclose function is a good programming practice.

Writing to a File

In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library offers the necessary functions to write to a file:

- **fputc(char, file_pointer)**: It writes a character to the file pointed to by file_pointer.
- **fputs(str, file_pointer)**: It writes a string to the file pointed to by file_pointer.
- **fprintf(file_pointer, str, variable_lists)**: It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.
- **fwrite()**

The program below shows how to perform writing to a file:

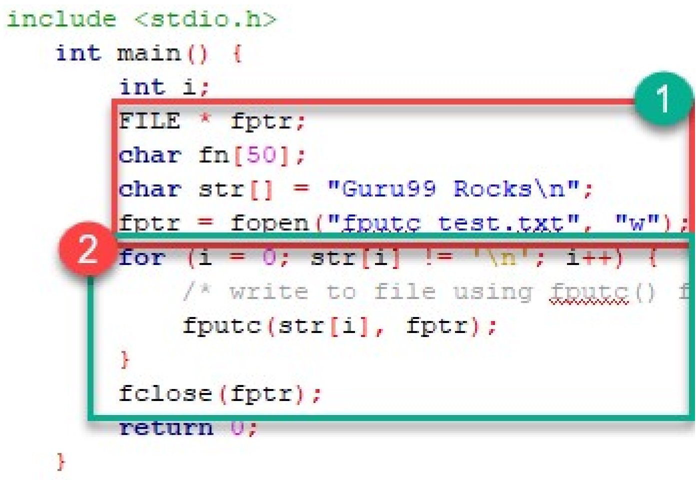
fputc() Function:

```
#include <stdio.h>
int main() {
        int i;
        FILE * fptr;
        char fn[50];
        char str[] = "Guru99 Rocks\n";
        fptr = fopen("fputc_test.txt", "w"); // "w" defines "writing mode"
        for (i = 0; str[i] != '\n'; i++) {
            /* write to file using fputc() function */
            fputc(str[i], fptr);
        }
```

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

Output:

The above program writes a single character into the **fputc_test.txt** file until it reaches the next line symbol "\n" which indicates that the sentence was successfully written. The process is to take each character of the array and write it into the file.



```
include <stdio.h>
   int main() {
       int i;
       FILE * fptr;
       char fn[50];
       char str[] = "Guru99 Rocks\n";
       fptr = fopen("fputc_test.txt", "w");
       for (i = 0; str[i] != '\n'; i++) {
           /* write to file using fputc() f
           fputc(str[i], fptr);
       }
       fclose(fptr);
       return 0;
   }
```

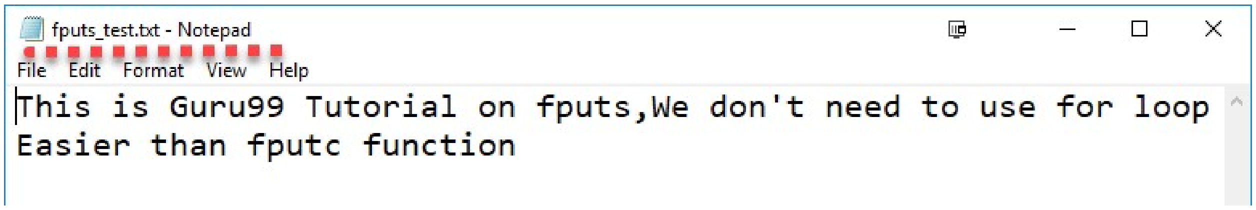
1. In the above program, we have created and opened a file called fputc_test.txt in a write mode and declare our string which will be written into the file.
2. We do a character by character write operation using for loop and put each character in our file until the "\n" character is encountered then the file is closed using the fclose function.

fputs () Function:

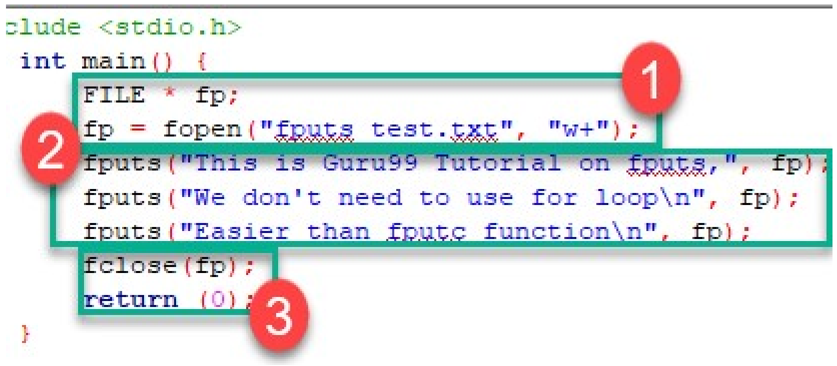
```
#include <stdio.h>
int main() {
        FILE * fp;
        fp = fopen("fputs_test.txt", "w+");
        fputs("This is Guru99 Tutorial on fputs,", fp);
        fputs("We don't need to use for loop\n", fp);
        fputs("Easier than fputc function\n", fp);
        fclose(fp);
        return (0);
```

```
   }
```

OUTPUT:



```
This is Guru99 Tutorial on fputs,We don't need to use for loop
Easier than fputc function
```



```
clude <stdio.h>
 int main() {
     FILE * fp;
     fp = fopen("fputs_test.txt", "w+");
     fputs("This is Guru99 Tutorial on fputs,", fp);
     fputs("We don't need to use for loop\n", fp);
     fputs("Easier than fputc function\n", fp);
     fclose(fp);
     return (0);
 }
```

1. In the above program, we have created and opened a file called fputs_test.txt in a write mode.
2. After we do a write operation using fputs() function by writing three different strings
3. Then the file is closed using the fclose function.

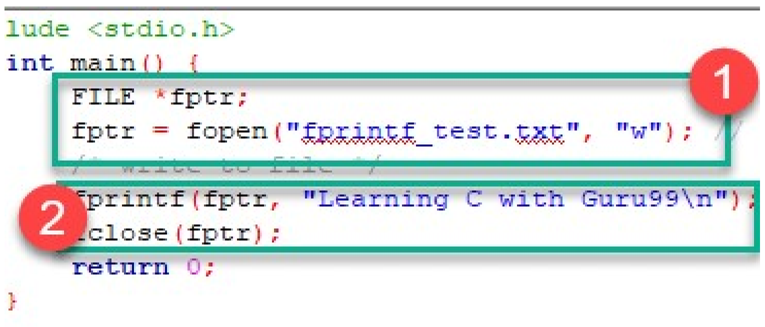
Problem Solving Using C
KCA 102: Session 2020-21

Unit 5	**Lecture 3**	**File Handling cont…**

fprintf()Function:

```
#include <stdio.h>
   int main() {
      FILE *fptr;
      fptr = fopen("fprintf_test.txt", "w"); // "w" defines "writing mode"
      /* write to file */
      fprintf(fptr, "Learning C with Guru99\n");
      fclose(fptr);
      return 0;
   }
```

OUTPUT:



```
lude <stdio.h>
int main() {
    FILE *fptr;
    fptr = fopen("fprintf_test.txt", "w"); //
    /* write to file */
    fprintf(fptr, "Learning C with Guru99\n");
    fclose(fptr);
    return 0;
}
```

1. In the above program we have created and opened a file called fprintf_test.txt in a write mode.
2. After a write operation is performed using fprintf() function by writing a string, then the file is closed using the fclose function.

Reading data from a File

There are three different functions dedicated to reading data from a file

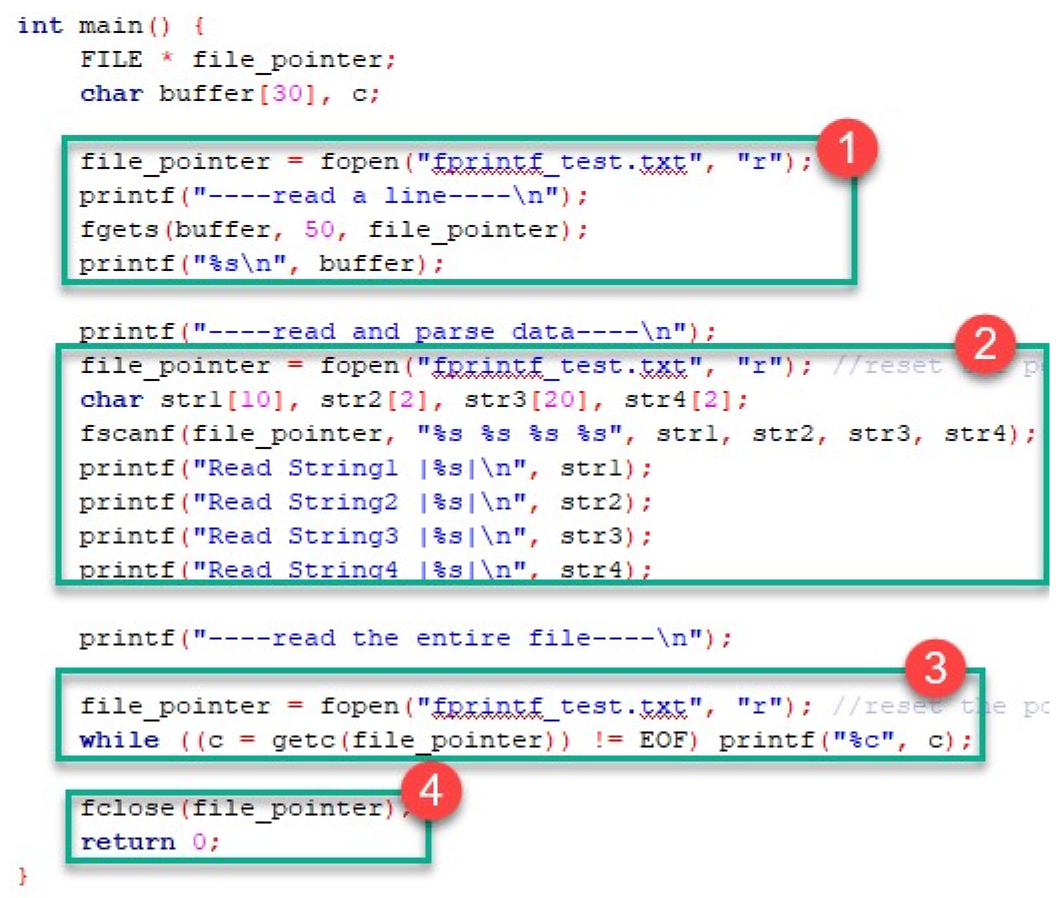
- **fgetc(file_pointer):** It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.
- **fgets(buffer, n, file_pointer):** It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.
- **fscanf(file_pointer, conversion_specifiers, variable_adresses)**: It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable_adresses using conversion specifiers. Keep in mind that as with scanf, fscanf stops reading a string when space or newline is encountered.

The following program demonstrates reading from fputs_test.txt file using fgets(),fscanf() and fgetc () functions respectively :

```
#include <stdio.h>
int main() {
        FILE * file_pointer;
        char buffer[30], c;

        file_pointer = fopen("fprintf_test.txt", "r");
        printf("----read a line----\n");
        fgets(buffer, 50, file_pointer);
        printf("%s\n", buffer);

        printf("----read and parse data----\n");
        file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
        char str1[10], str2[2], str3[20], str4[2];
        fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
        printf("Read String1 |%s|\n", str1);
        printf("Read String2 |%s|\n", str2);
        printf("Read String3 |%s|\n", str3);
        printf("Read String4 |%s|\n", str4);

        printf("----read the entire file----\n");

        file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
        while ((c = getc(file_pointer)) != EOF) printf("%c", c);

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

Result:

```
----read a line----
Learning C with Guru99

----read and parse data----
Read String1 |Learning|
Read String2 |C|
Read String3 |with|
Read String4 |Guru99|
----read the entire file----
Learning C with Guru99
```



```
int main() {
    FILE * file_pointer;
    char buffer[30], c;

    file_pointer = fopen("fprintf_test.txt", "r");
    printf("----read a line----\n");
    fgets(buffer, 50, file_pointer);
    printf("%s\n", buffer);

    printf("----read and parse data----\n");
    file_pointer = fopen("fprintf_test.txt", "r"); //reset
    char str1[10], str2[2], str3[20], str4[2];
    fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
    printf("Read String1 |%s|\n", str1);
    printf("Read String2 |%s|\n", str2);
    printf("Read String3 |%s|\n", str3);
    printf("Read String4 |%s|\n", str4);

    printf("----read the entire file----\n");

    file_pointer = fopen("fprintf_test.txt", "r"); //reset the pc
    while ((c = getc(file_pointer)) != EOF) printf("%c", c);

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

1. In the above program, we have opened the file called "fprintf_test.txt" which was previously written using fprintf() function, and it contains "Learning C with Guru99" string. We read it using the fgets() function which reads line by line where the buffer size must be enough to handle the entire line.
2. We reopen the file to reset the pointer file to point at the beginning of the file. Create various strings variables to handle each word separately. Print the variables to see their contents. The fscanf() is mainly used to extract and parse data from a file.

3. Reopen the file to reset the pointer file to point at the beginning of the file. Read data and print it from the file character by character using getc() function until the EOF statement is encountered
4. After performing a reading operation file using different variants, we again closed the file using the fclose function.

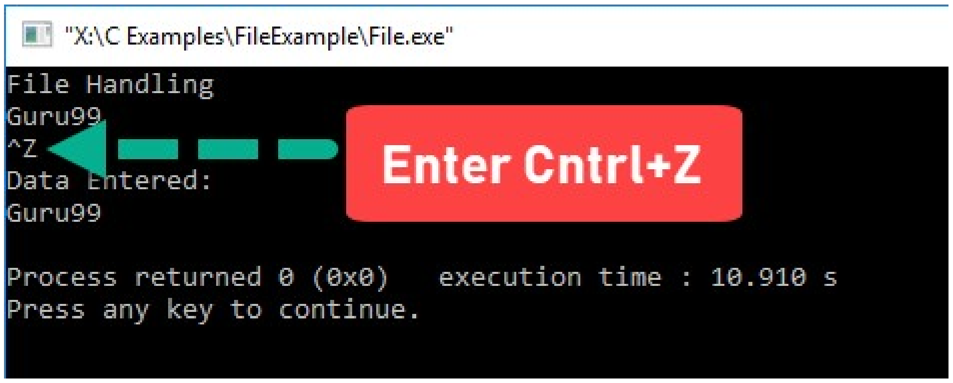
Interactive File Read and Write with getc and putc

These are the simplest file operations. Getc stands for get character, and putc stands for put character. These two functions are used to handle only a single character at a time.

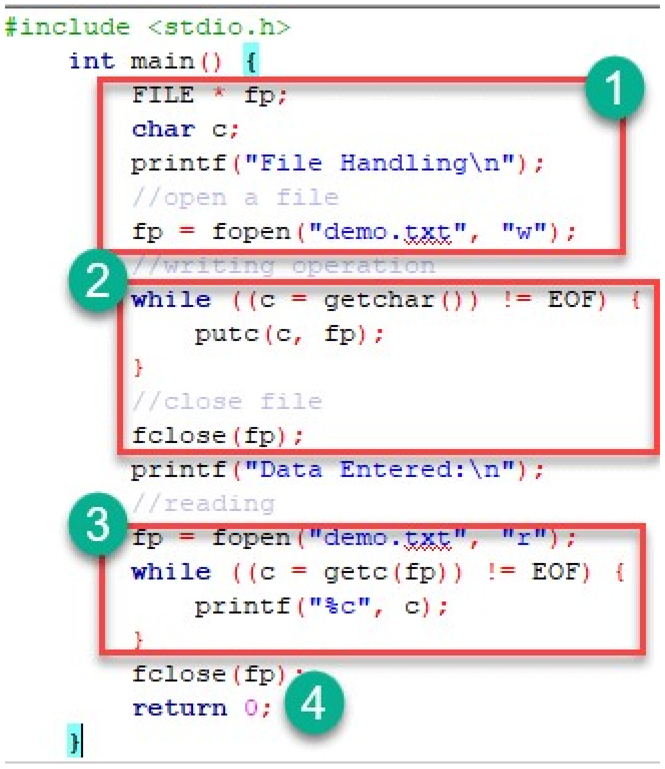
Following program demonstrates the file handling functions in 'C' programming:

```
#include <stdio.h>
int main() {
        FILE * fp;
        char c;
        printf("File Handling\n");
        //open a file
        fp = fopen("demo.txt", "w");
        //writing operation
        while ((c = getchar()) != EOF) {
            putc(c, fp);
        }
        //close file
        fclose(fp);
        printf("Data Entered:\n");
        //reading
        fp = fopen("demo.txt", "r");
        while ((c = getc(fp)) != EOF) {
            printf("%c", c);
        }
        fclose(fp);
        return 0;
    }
```

Output:



```
File Handling
Guru99
^Z
Data Entered:
Guru99

Process returned 0 (0x0)   execution time : 10.910 s
Press any key to continue.
```



```
#include <stdio.h>
    int main() {
        FILE * fp;
        char c;
        printf("File Handling\n");
        //open a file
        fp = fopen("demo.txt", "w");
        //writing operation
        while ((c = getchar()) != EOF) {
            putc(c, fp);
        }
        //close file
        fclose(fp);
        printf("Data Entered:\n");
        //reading
        fp = fopen("demo.txt", "r");
        while ((c = getc(fp)) != EOF) {
            printf("%c", c);
        }
        fclose(fp);
        return 0;
    }
```

1. In the above program we have created and opened a file called demo in a write mode.
2. After a write operation is performed, then the file is closed using the fclose function.
3. We have again opened a file which now contains data in a reading mode. A while loop will execute until the eof is found. Once the end of file is found the operation will be terminated and data will be displayed using printf function.
4. After performing a reading operation file is again closed using the fclose function.

Summary

- A file is a space in a memory where data is stored.
- 'C' programming provides various functions to deal with a file.

- A mechanism of manipulating with the files is called as file management.
- A file must be opened before performing operations on it.
- A file can be opened in a read, write or an append mode.
- Getc and putc functions are used to read and write a single character.
- The function fscanf() permits to read and parse data from a file
- We can read (using **the getc** function) an entire file by looping to cover all the file until the EOF is encountered
- We can write to a file after creating its name, by using the function **fprintf()** and it must have the newline character at the end of the string text.

Problem Solving Using C
KCA 102: Session 2020-21

Unit 5	Lecture 4	Command line argument

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"

Progranm name ./a.out
The argument supplied is testing1 testing2
```

Problem Solving Using C
KCA 102: Session 2020-21

Unit 5	**Lecture 5**	**Graphics in C**

Introduction
To start with graphics programming, Turbo C is a good choice. Even though DOS has its own limitations, it is having a large number of useful functions and is easy to program. To implement graphics algorithms, to give graphical display of statistics, To view signals from any source, we can use C graphics. Here is a article to start programming with Turbo C. 'Run and Learn' is our method. We have used source codes throughout the explanations. Just execute them to understand what is happening.

You should have!!!
1. Graphics.h Header File
2. Graphics.lib library file
3. Graphics driver (BGI file)
4. At least 640×480 VGA monitor
5. Header File : graphics.h
6. All Graphical Functions are Included in Graphics.h
7. After Including graphics.h Header File [You can get access graphical functions]
8. You must Know Following Things before Learning Turbo Graphics
9. InitGraph : Initializes the graphics system.
10. In C Program execution starts with main() similarly Graphics Environment Starts with this function.
11. initgraph() initializes the graphics system by loading a graphics driver from disk then putting the system into graphics mode
12. As this is Our first topic Under Graphics so it is better not to go in details of Parameters.

1. A Simple Program

```
#include<graphics.h>
#include<conio.h>
void main()
{
int gd = DETECT, gm;
initgraph(&gd, &gm, "c:\\tc\\bgi");
///Your Code goes Here
getch();
closegraph();
}
```

Output of this simple program

1. Firstly let me tell you what the output of this program is.

2. This program initializes graphics mode and then closes it after a key is pressed.

3. To begin with we have declared two variables of int type gd and gm for graphics driver and graphics mode respectively.

4. DETECT is a macro defined in "graphics.h" header file.

5. Then we have passed three arguments to initgraph function 1st is the address of gd, 2nd is the address of gm and 3rd is the path where your BGI files are present (**you have to adjust this accordingly where you turbo compiler is installed**).

6. getch helps us to wait until a key is pressed, closegraph function closes the graphics mode and finally return statement returns a value 0 to main indicating successful execution of your program

7. After you have understood initgraph function then you can use functions to draw shapes such as circle, line , rectangle etc, then you can learn how to change colors and fonts using suitable functions, then you can go for functions such as getimage, putimage etc for doing animation.

2. Program---Printing Text in Graphics Using Outtextxy Function

```
#include<graphics.h>
#include<stdio.h>
int main(void)
{
int gdriver = DETECT, gmode;
initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
int x = 200, y = 200;
outtextxy(x, y, "Hello World");
closegraph();
}
```

Explanation

outtextxy(x,y,"Hello World");
This Function is Similar to Printf Statement.
Printf Prints Text on Screen in "Text Mode" while outtextxy() function Prints Text onto Screen in "Graphics Mode".
This Function Accepts 3 Parameters.
Syntax:
outtextxy(x,y,"Hello World");

3. Program --- How to Draw a Cricle

```
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT, gm;
initgraph(&gd, &gm, "c:\\turboc3\\bgi " );
circle(200,100,150);
getch();
closegraph();
}
```

More detail on Syntax

1. InitGraph: Initializes the graphics system.
2. Declaration: void far initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);
3. Remarks: To start the graphics system, you must first call initgraph.
4. initgraph initializes the graphics system by loading a graphics driver from disk (or validating a registered driver) then putting the system into graphics mode.
5. initgraph also resets all graphics settings (color, palette, current position, viewport, etc.) to their defaults, then resets graphresult to 0.

Problem Solving Using C
KCA 102: Session 2020-21

Unit 5	**Lecture 6,7,8**	**Graphics in C**

4. Program --- Draw a Rectangle

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
int gd = DETECT, gm;
initgraph(&gd, &gm, "c:\\tc\\bgi");
line(300, 100, 200, 200); // (from_x,from_y, to_x,to_y)
line(300, 100, 400, 200);
line(200, 200, 400, 200);
getch();
closegraph();
}
```

5. Program --- Setting Styles and More!!!

```
/*Here a sample program to illustrate how to use BARS which are used for visual statistics
*/
#include<graphics.h>
main() {
int gd=DETECT,gm,maxx,maxy,x,y,button;
initgraph(&gd,&gm,"");
line(80,150,200,150);
line(80,150,80,50);
settextstyle(1,HORIZ_DIR,1);
outtextxy(100,153,"<-X axis");
settextstyle(1,VERT_DIR,1);
outtextxy(60,50,"<-Y axis");
bar(100,100,120,150);
bar(130,120,150,150);
getch();
closegraph();
}
```

6. Program --- Draw Op-amp Using Graphics Function

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main() {
int gd = DETECT, gm;
initgraph(&gd, &gm, "c:\\tc\\bgi");
line(50, 125, 100, 125); //Horizontal Line -VE Terminal
line(50, 175, 100, 175); //Horizontal Line +VE Terminal
line(100, 100, 100, 200); //Vertical Line
line(100, 100, 150, 150); //Top to Bottom Slant Line
line(150, 150, 100, 200); //Bottom to Top slant Line
line(125, 100, 125, 125); //Vertical Line +VCC
line(125, 175, 125, 200); //Vertical Line -VCC
line(150, 150, 200, 150); //Horizontal line
getch();
closegraph();
}
```

Important Functions

Clearing the graphics window…
cleardevice();
Delay the program, so that users can see what is happening…sending in the number of msec
delay(milliseconds);
Wait for a keyboard hit:
getch();or,kbhit();

Important functions (Drawing lines)

1. Set Drawing Color (for lines and edges)
2. (colors typically range from 0-15; 0 is usually black and 15 is white)
3. setcolor(color);
4. Set Background Color (usually for text)
5. setbkcolor(color);
6. Set Fill Style and Color (for interiors)
7. Pattern 0-12, 0 = empty, 1 = solid
8. setfillstyle(pattern, color)
9. Set Line Style and Thickness
10. Style: 0 = solid, 1 = dotted, 3 = dashed
11. Thickness is the width in terms of pixels
12. setlinestyle(style, pattern, thickness)

Important Functions(Drawing…Areas)

1. Drawing absolute (from one coordinate to another)
2. linerel(from_x, from_y,
3. Drawing a Circle
4. lGiven center and radius as whole numbers
5. circle (center_x, center_y, radius);

6. lDrawing a filled Rectangle
7. (given upper left and lower right corners)
8. bar(ul_x, ul_y, lr_x,lr_y);
9. lDrawing an unfilled Rectangle
10. (given upper left and lower right corners)
11. rectangle(ul_x, ul_y, lr_x, lr_y); to_x, to_y);

Important Functions(How Text Look)

1. Text Formatting
2. Set the justification
3. Horizontal: (0 = left, 1 = center, 2= right)
4. Vertical: (0 = bottom, 1 = center, 2 = top)
5. settextjustify(horizontal, vertical)
6. Set the text style
7. Font: (0-11)
8. Direction: 0 = left to right direction
9. Character Size: 0 = normal, 6 is really big!
10. settextstyle(font,direction, character size)

BLACK: 0 BLUE: 1 GREEN: 2 CYAN: 3 RED: 4 MAGENTA: 5 BROWN: 6 LIGHTGRAY: 7 DARKGRAY: 8 LIGHTBLUE: 9 LIGHTGREEN: 10 LIGHTCYAN: 11 LIGHTRED: 12 LIGHTMAGENTA: 13 YELLOW: 14 WHITE: 15

Important Functions(Messages)

1. Text Output

2. Set Text color (index ranges 0-15)

3. setcolor(index);

4. Output a message on the graphics window at the current position

5. outtext("messages on graphics window");

6. Output a message on the graphics window at the given x,y coordinate

7. outtextxy(x,y,"message");

8. Basic Shapes and Colors

```
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT, gm;
int poly[12]={350,450, 350,410, 430,400, 350,350, 300,430, 350,450 };
initgraph(&gd, &gm, "");
circle(100,100,50);
outtextxy(75,170, "Circle"); rectangle(200,50,350,150);
outtextxy(240, 170, "Rectangle");
ellipse(500, 100,0,360, 100,50);
outtextxy(480, 170, "Ellipse");
line(100,250,540,250);
outtextxy(300,260,"Line");
sector(150, 400, 30, 300, 100,50);
outtextxy(120, 460, "Sector");
drawpoly(6, poly);
outtextxy(340, 460, "Polygon");
getch();
closegraph();
}
```

Output!!!

9. Program --- Moving Car

```
#include <graphics.h>
#include <dos.h>
int main()
{
int i, j = 0, gd = DETECT, gm;
initgraph(&gd,&gm,"C:\\TC\\BGI");
settextstyle(DEFAULT_FONT,HORIZ_DIR,2);
outtextxy(25,240,"Press any key to view the moving car");
getch();
for( i = 0 ; i <= 420 ; i = i + 10, j++ )
{
rectangle(50+i,275,150+i,400);
rectangle(150+i,350,200+i,400);
circle(75+i,410,10);
circle(175+i,410,10);
setcolor(j);
delay(100);
if( i == 420 )
break;
if ( j == 15 )
j = 2;
cleardevice(); // clear screen
}
getch();
closegraph();
return 0;
}
```

11. Some Graphics Effects Using Random Numbers

```
#include "graphics.h"
#include "conio.h"
#include "stdlib.h"
void main()
{
int gd,gm;
gd=DETECT;
initgraph(&gd, &gm, "");
setcolor(3);
setfillstyle(SOLID_FILL,RED);
bar(50, 50, 590, 430);
setfillstyle(1, 14);
bar(100, 100, 540, 380);
while(!kbhit())
{
putpixel(random(439)+101, random(279)+101,random(16));
setcolor(random(16));
circle(320,240,random(100));
}
getch();
closegraph();
}
```

End