CL1002 Programming Fundamentals

LAB 12

Dynamic Memory Allocation, File Handling

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Learning Objectives

1. Dynamic Memory
2. File Handling

1.0 Dynamic Memory

The process of allocating memory during program execution is called dynamic memory allocation. The ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed is known as dynamic memory management.

1.1 Importance of Dynamic memory

Many times, it is not known in advance how much memory will be needed to store particular information in a defined variable and the size of required memory can be determined at run time. For example, we may want to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.

C language offers 4 dynamic memory allocation functions. They are,

Function	**Syntax**
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

Malloc()

- malloc function is used to allocate space in memory during the execution of the program.
- malloc does not initialize the memory allocated during execution. It carries garbage value.
- Malloc function returns null pointer if it couldn't able to allocate requested amount of memory.

Calloc()

- calloc function is also like malloc function. But calloc initializes the allocated memory to zero. But, malloc doesn't.

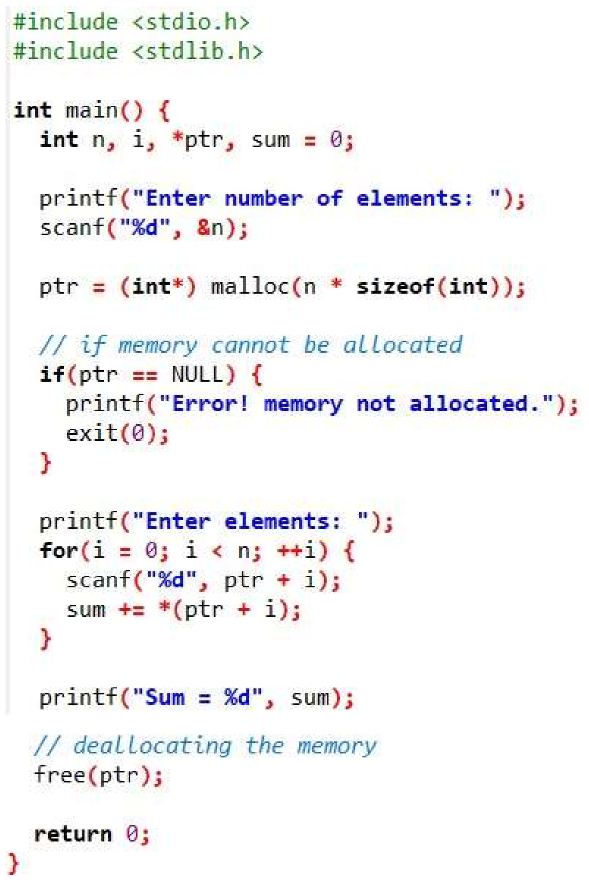
Realloc()

- realloc function modifies the allocated memory size by malloc and calloc functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

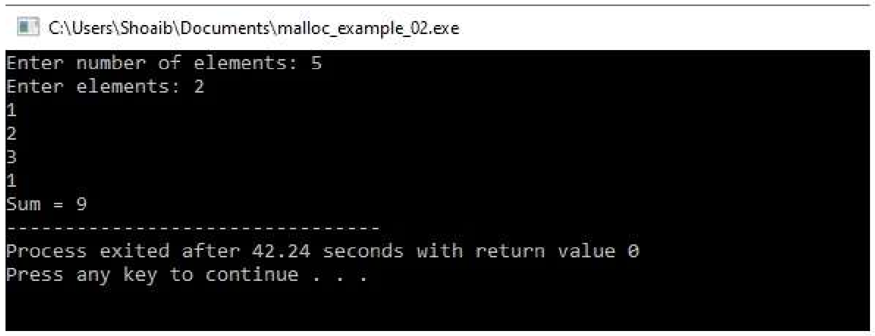
Free()

- free function frees the allocated memory by malloc, calloc, realloc functions and returns the memory to the system.

Malloc & free Sample Code:



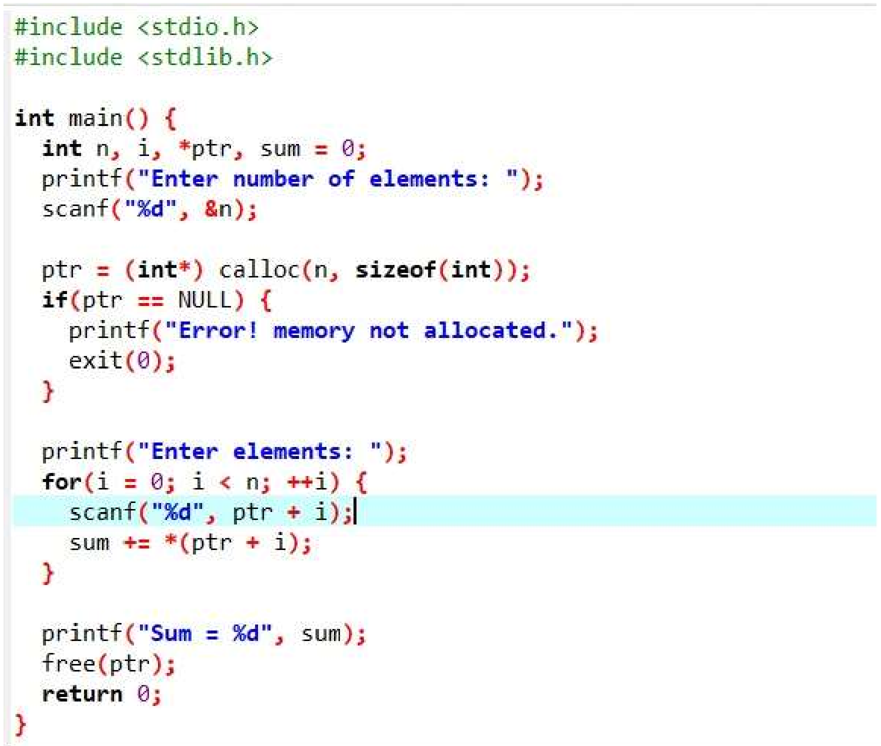
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

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

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);

 // deallocating the memory
 free(ptr);

 return 0;
}
```



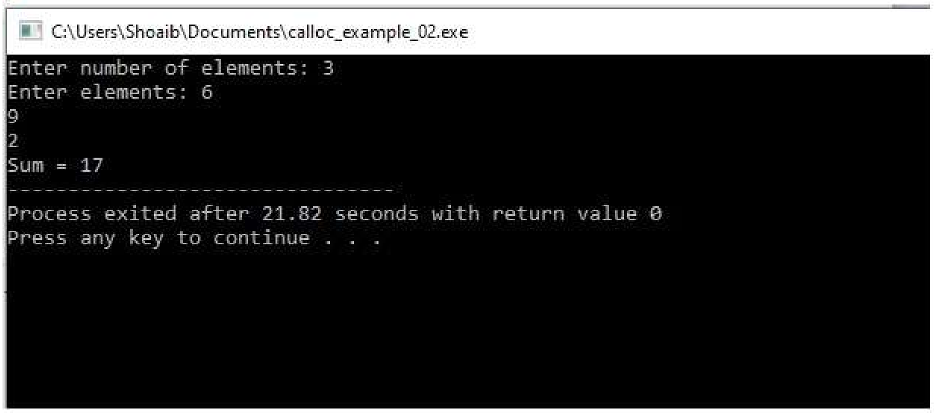
C:\Users\Shoaib\Documents\malloc_example_02.exe

```
Enter number of elements: 5
Enter elements: 2
1
2
3
1
Sum = 9
--------------------------------
Process exited after 42.24 seconds with return value 0
Press any key to continue . . .
```

Calloc & free Sample Code:



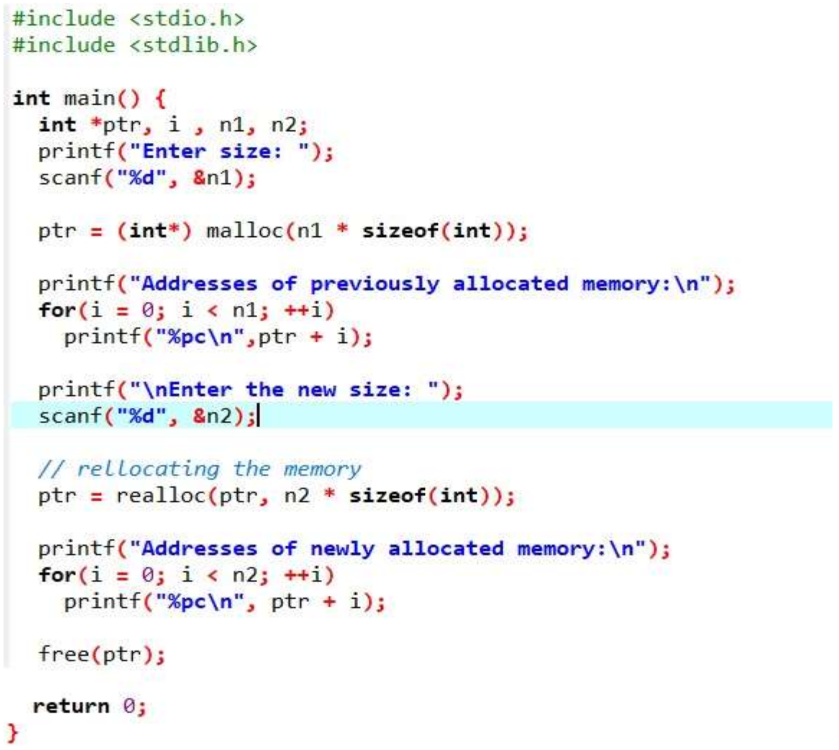
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;
  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) calloc(n, sizeof(int));
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);
  free(ptr);
  return 0;
}
```



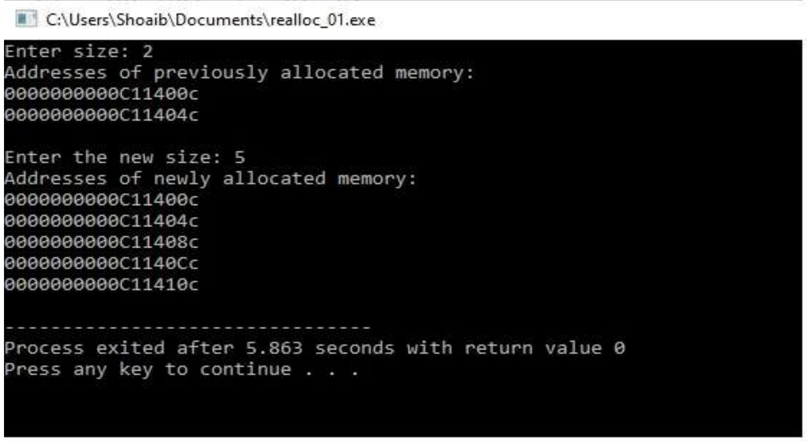
C:\Users\Shoaib\Documents\calloc_example_02.exe

```
Enter number of elements: 3
Enter elements: 6
9
2
Sum = 17
--------------------------------
Process exited after 21.82 seconds with return value 0
Press any key to continue . . .
```

Realloc Sample Code:



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

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:\n");
  for(i = 0; i < n1; ++i)
    printf("%pc\n",ptr + i);

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

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

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

  free(ptr);

 return 0;
}
```



C:\Users\Shoaib\Documents\realloc_01.exe

```
Enter size: 2
Addresses of previously allocated memory:
0000000000C11400c
0000000000C11404c

Enter the new size: 5
Addresses of newly allocated memory:
0000000000C11400c
0000000000C11404c
0000000000C11408c
0000000000C1140Cc
0000000000C11410c

--------------------------------
Process exited after 5.863 seconds with return value 0
Press any key to continue . . .
```

1.2 Difference between static memory allocation and dynamic memory allocation in C

Static memory allocation	**Dynamic memory allocation**
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

C File Handling

file is a container in computer storage devices used for storing data.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all.
  However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.
When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the **.bin** files in your computer.
Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.
The syntax for opening a file in standard I/O is:

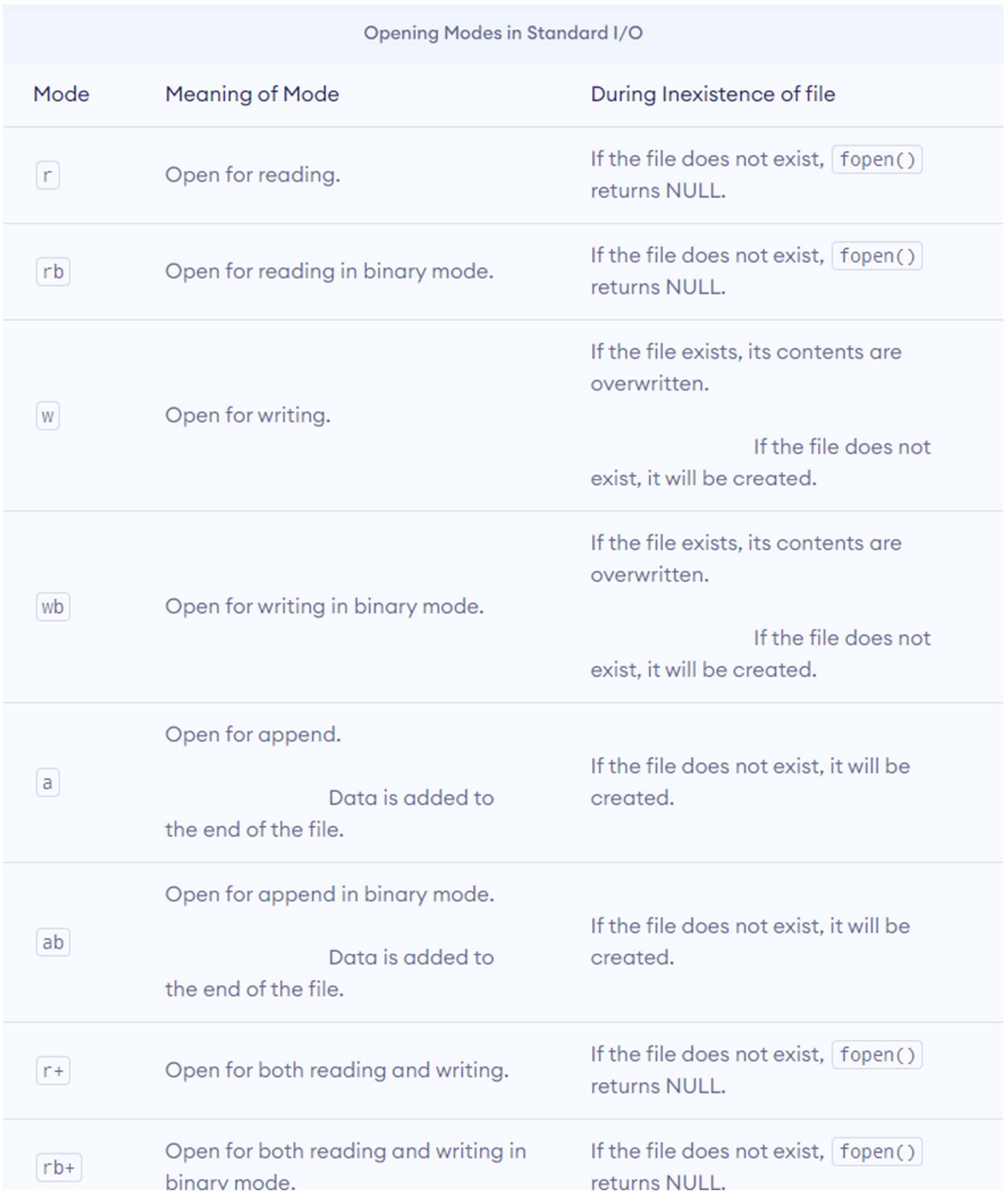
```
ptr = fopen("fileopen","mode");
```

For example,

```
fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");
```

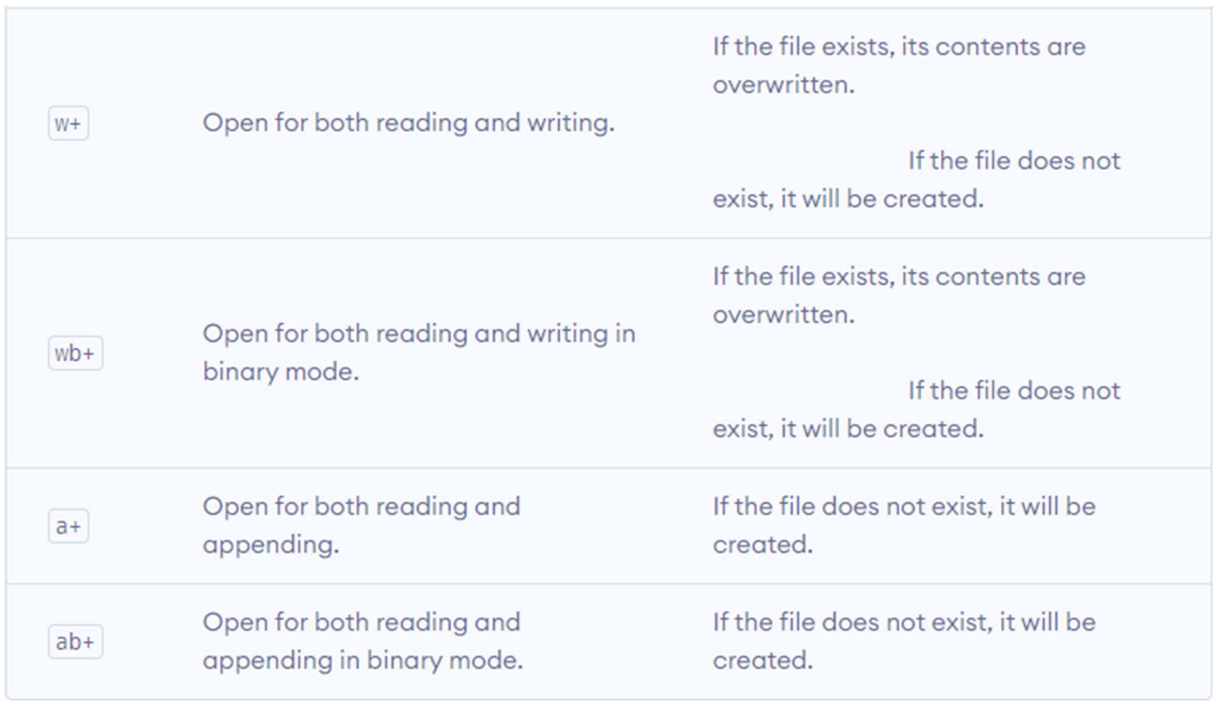
- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode **'w'**.
  The writing mode allows you to create and edit (overwrite) the contents of the file.

- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`. The second function opens the existing file for reading in binary mode **'rb'**.
  The reading mode only allows you to read the file, you cannot write into the file.



Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
`r`	Open for reading.	If the file does not exist, `fopen()` returns NULL.
`rb`	Open for reading in binary mode.	If the file does not exist, `fopen()` returns NULL.
`w`	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`wb`	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`a`	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
`ab`	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
`r+`	Open for both reading and writing.	If the file does not exist, `fopen()` returns NULL.
`rb+`	Open for both reading and writing in binary mode.	If the file does not exist, `fopen()` returns NULL.



`w+`	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`wb+`	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`a+`	Open for both reading and appending.	If the file does not exist, it will be created.
`ab+`	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the `fclose()` function.

```
fclose(fptr);
```

Here, `fptr` is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf().` They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprintf()` and `fscanf()` expects a pointer to the structure FILE.

Example 1: Write to a text file

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

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

   // use appropriate location if you are using MacOS or Linux
   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);

   return 0;
}
```

This program takes a number from the user and stores in the file `program.txt`. After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text 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;
}
```

This program reads the integer present in the `program.txt` file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like `fgetchar()`, `fputc()` etc. can be used in a similar way.

R

eading and writing to a binary file

F

unctions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

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

struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C“\\program.bin",”w””)” == NULL){
       printf("E“ror! opening file")”

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

   for(n = 1; n < 5; ++n)
   {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);

    return 0;
}
```

In this program, we create a new file `program.bin` in the C drive.

We declare a structure `threeNum` with three numbers - `-1, n2 and n3`, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using `fwrite()`.

The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.

Since we'r' only inserting one instance of `num`, the third parameter is `1`. And, the last parameter `*fptr` points to the file we'r' storing the data.

Finally, we close the file.

Reading from a binary file

Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using fread()

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

struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

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

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

   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
   }
   fclose(fptr);

   return 0;
}
```

In this program, you read the same file `program.bin` and loop through the records one by one.

In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

You'll get the same records you inserted in **Example 3**.

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.
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.
`SEEK_CUR`	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

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

struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

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

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

   // Moves the cursor to the end of the file
   fseek(fptr, -sizeof(struct threeNum), SEEK_END);

   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
      fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
   }
   fclose(fptr);

   return 0;
}
```

This program will start reading the records from the file `program.bin` in the reverse order (last to first) and prints it.

Exercises:

Task_01:

Write a C Program to Find Largest Number in an array Using Dynamic Memory Allocation.
Note: The array must be dynamically resized and initially the array size will be taken by the user and then resized by the user
Example arr[5]={1,2,3,4,5}
//After resizing
Arr[7]={1,2,3,4,5,6,7}
Take the same array and resize it.

Task_02:

Write a C program to read name and marks of n number of students and store them in a file

Task_03:

Write C program to read name and marks of n number of students from and store them in a file. If the file previously exits, add the information to the file.

Task_04:

C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

Task_05:

Write C Program that Reads a Text File and Counts the Number of Times a Certain Letter Appears in the Text File

Task_06:

Consider a String entered by the user which he wants to encrypt and then decrypt the information. Perform this functionality via the filling technique by first inserting the encrypted text in the file then read the encrypted text to decode it back to string.
Input:
Enter String = Hello World
Output:
Normal Text
Hello World
Encrypted text inserted in file
Igopt&^w{vo
Decrypted text Read then decoded from file
Hello World

(Note: For Encryption the algorithm would be is to take the length of the string then add the number based on the index position of the character of the string when inserting into the file)
Example:
ABC = //Here A is at position 1 B is at position 2 and c is at position 3 if we add these position values to their ascii code then the ascii values become
Original values = 65 66 67; modified = 66 68 70 which converts the text to B D F