# CS - 114 : Computer Workshop

Prof. Chamakuri Nagaiah
Mahindra-École Centrale, Hyderabad
nagaiah.chamakuri@mechyd.ac.in

# C Programming Files I/O: 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 few commands in C.
- You can easily move your data from one computer to another without any changes.

# What is a file

- A named collection of data, stored in secondary storage (typically).
- Typical operations on files:
  - Open
  - Read
  - Write
  - Close
- How is a file stored?

  – Stored as sequence of bytes, logically contiguous (may not be physically contiguous on disk).

  – The last byte of a file contains the end-of-file character ( EOF), with ASCII code 1A (hex).

# Types of Files

- Text :: contains ASCII codes only
  – Text files are the normal .txt files. They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.
- Binary :: can contain non-ASCII characters
  – Binary files are mostly the .bin files in your computer. They can hold higher amount of data, are not readable easily and provides a better security than text files.
  – Image, audio, video, executable, etc.
  – To check the end of file here, the file size value (also stored on disk) needs to be checked.

# File handling in C

- In C, we use FILE * to represent a pointer to a file.
- fopen is used to open a file. It returns the special value NULL to indicate that it is unable to open the file.

# File handling in C

- In C, we use FILE * to represent a pointer to a file.
- fopen is used to open a file. It returns the special value NULL to indicate that it is unable to open the file.

```
FILE *fptr;
char filename[]= "example.txt";
fptr = fopen (filename,"w");
if (fptr == NULL)
{
  printf ("ERROR IN FILE CREATION");
  /* DO SOMETHING */
}
```

# Opening Modes in Standard I/O

- General Syntax : fptr = fopen(“filename“,”mode“)

# Opening Modes in Standard I/O

- General Syntax : fptr = fopen("filename","mode")

| File 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. i.e, Data is added to end of file. | If the file does not exists, 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 library function fclose().
- fclose(fptr); //fptr is the file pointer associated with file to be closed.

# Writing to a text file

```
 #include <stdio.h>
int main(){
   int num;
   FILE *fptr;
   fptr = fopen("file_write.txt","w");
   if(fptr == NULL) {
      printf("Error!");
      exit(1);
   }
   printf("Enter num: ");
   scanf("%d",&num);

   fprintf(fptr,"%d",num);
   fclose(fptr);
   return 0;
}
```

# Reading from a text file

```
#include <stdio.h>
int main(){
   int num;
   FILE *fptr;
   if ((fptr = fopen("file_read.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;
}
```

# Writing to a binary file

- To write into a binary file, you need to use the function fwrite().
- General syntax :

  fwrite(address_data,size_data,numbers_data,pointer_to_file);
- The functions takes four arguments:
  - Address of data to be written in disk,
  - Size of data to be written in disk,
  - number of such type of data
  - pointer to the file where you want to write.

# Writing to a binary file using fwrite()

```
 #include <stdio.h>
struct Example
{
   int n1, n2, n3;
};
int main()
{
   int n;
   struct Example num;
   FILE *fptr;
   if ((fptr = fopen("example_struct.bin","wb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
```

# Writing to a binary file using fwrite()

```
    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5n;
        num.n3 = 5n + 1;
        fwrite(&num, sizeof(struct Example), 1, *fptr);
    }
    fclose(fptr);
    return 0;
}
```

# The exit() function

- Sometimes error checking means we want an "emergency exit " from a program.
- In main() we can use return to stop.
- In functions we can use exit() to do this.
- Exit is part of the stdlib.h library.

  exit(-1);

  in a function is exactly the same as

  return -1;

  in the main routine

# C Program to Write a Sentence to a File

```
#include <stdio.h>
#include <stdlib.h>  /* For exit() function */
int main(){
   char sentence[1000];
   FILE *fptr;
   fptr = fopen("program.txt", "w");
   if(fptr == NULL) {
      printf("Error!");
      exit(1);
   }
   printf("Enter a sentence:\n");
   gets(sentence);
   fprintf(fptr,"%s", sentence);
   fclose(fptr);
   return 0;
}
```

# Reading Data Using fscanf( )

- We also read data from a file using fscanf().

```
FILE *fptr;
Fptr = fopen ("input.dat", "r");
/* Check it's open */
if (fptr == NULL)
{
  printf("Error in opening file \n");
}
fscanf (fptr, "%d %d",&x, &y);
```

# Reading lines from a file using fgets( )

- We can read a string using fgets().
- fgets() takes 3 arguments - a string, maximum number of characters to read, and a file pointer. It returns NULL if there is an error (such as EOF).

```
FILE *fptr;
char line [1000];
/* Open file and check it is open */
while (fgets(line,1000,fptr) != NULL)
{
  printf ("Read line %s\n",line);
}
```

# Program to read text from a file

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
int main(){
    char c[1000];
    FILE *fptr;
    if ((fptr = fopen("program.txt", "r")) == NULL)    {
        printf("Error! opening file");
        // Program exits if file pointer returns NULL.
        exit(1);
    }
    // reads text until newline
    fscanf(fptr,"%[^\n]", c);

    printf("Data from the file:\n%s", c);
    fclose(fptr);
    return 0;
}
```

# Input File & Output File redirection

- One may redirect the standard input and standard output to other files (other than stdin and stdout).

# Input File & Output File redirection

- One may redirect the standard input and standard output to other files (other than stdin and stdout).
- Usage: Suppose the executable file is a.out:

```
$ ./a.out <in.dat >out.dat
 OR
$ ./a.out <in.dat >>out.dat
```

scanf() will read data inputs from the file ”in.dat“, and printf() will append results at the end of the file ”out.dat”.

# Command Line Arguments : What are they?

- A program can be executed by directly typing a command at the operating system prompt.

```
$ cc -o test test.c
$ ./a.out in.dat out.dat
$ prog_name param_1 param_2 param_3 ...
```

- The individual items specified are separated from one another by spaces.

  First item is the program name.

- Variables argc and argv keep track of the items specified in the command line.

# How to access them?

- Command line arguments may be passed by specifying them under main().

```
int main (int argc, char *argv[]);
```

- argc : Argument Count
- *argv[] : Array of strings as command line arguments including the command itself.

# Example: reading command line arguments

```
#include <stdio.h>
#include <string.h>
int main(int argc,char *argv[])
{
  FILE *ifp, *ofp;
  int i, c;
  char src_file[100],dst_file[100];
  if(argc!=3) {
    printf ("Usage: ./a.out <src_file> <dst_file> \n");
    exit(0);
  }
  else {
    strcpy (src_file, argv[1]);
    strcpy (dst_file, argv[2]);
  }
```

```
  if ((ifp = fopen(src_file,"r")) == NULL)
  {
    printf ("File does not exist.\n");
    exit(0);
  }

  if ((ofp = fopen(dst_file,"w")) == NULL)
  {
    printf ("File not created.\n");
    exit(0);
  }

  while ((c = fgetc(ifp)) != EOF) {
    fputc (c,ofp);
  }
fclose(ifp);
fclose(ofp);
}
```

# Example: with command-line arguments

- Write a program which will take the number of data items, followed by the actual data items on the command line, and print the average.

```
$ ./a.out 7 16 31 10 20 11 19 79
```

# Example: with command-line arguments

- Write a program which will take the number of data items, followed by the actual data items on the command line, and print the average.

```
$ ./a.out 7 16 31 10 20 11 19 79
```

- No. of data items : 7, i.e. argv[1] = 7
- argv[2] = 16, argv[3] = 31 and so on ...
- Once we have got a string with a number in it (either from a file or from the user typing) we can use atoi or atof to convert it to a number.
- Alternatively, we can use sscanf() .

# Example: with command-line arguments

- Write a program which will take the number of data items, followed by the actual data items on the command line, and print the average.

```
$ ./a.out 7 16 31 10 20 11 19 79
```

- No. of data items : 7, i.e. argv[1] = 7
- argv[2] = 16, argv[3] = 31 and so on ...
- Once we have got a string with a number in it (either from a file or from the user typing) we can use atoi or atof to convert it to a number.
- Alternatively, we can use sscanf() .

  sscanf (argv[1], “%d”, &n1);

# Dynamic Memory Allocation : Basic idea

- Many a time we face situations where data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data items keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.
- C language requires the number of elements in an array to be specified at compile time.
  - Often leads to wastage or memory space or program failure.
- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

- The program instructions and the global variables are stored in a region known as permanent storage area .
- The local variables are stored in another area called stack.
- The memory space between these two areas is available for dynamic allocation during execution of the program.
  - This free region is called the heap.
  - The size of the heap keeps changing.

# Memory Allocation Functions in "stdlib.h"

- malloc
  – Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
- calloc
  – Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free
  – deallocate the previously allocated space
- realloc
  – Modifies the size of previously allocated space.

# Allocating a Block of Memory : malloc()

- A block of memory can be allocated using the function malloc.
  – Reserves a block of memory of specified size and returns a pointer of type void.
  – The return pointer can be type-casted to any pointer type.
- General format:
  ptr = (type *) malloc (byte_size);
- Example :
  ptr = (int*) malloc(100 * sizeof(int));
  – A memory space equivalent to 100 times the size of an int bytes is reserved.
  – The address of the first byte of the allocated memory is assigned to the pointer p of type int.

# Examples :

- cptr = (char *) malloc (20);
  – Allocates 20 bytes of space for the pointer cptr of type char.
- sptr = (struct stud *) malloc (10 * sizeof (struct stud));
  – Allocates space for a structure array of 10 elements. sptr points to a structure element of type “struct stud”.

# Points to Note

- malloc always allocates a block of contiguous bytes.
  – The allocation can fail if sufficient contiguous memory space is not available.
  – If it fails, malloc returns NULL.

```
if ((p = (int *) malloc(100 * sizeof(int))) == NULL)
 {
   printf ("\n Memory cannot be allocated");
   exit();
 }
```

- Difference between malloc() and calloc() : malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

# Releasing the Used Space : free()

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.
- By using the free function.
- General syntax:

  free (ptr);

  where ptr is a pointer to a memory block which has been previously created using malloc.

# Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed.
  - Memory allocated is larger than necessary.
- If the original allocation is done as:

  ptr = malloc (size);

  then reallocation of space may be done as:

  ptr = realloc (ptr, newsize);
- The new memory block may or may not begin at the same place as the old one.
  - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns NULL and frees the original block.

# Example: Using C malloc() and free()

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

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

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

    ptr = (int*) malloc(num * sizeof(int));  //memory alloca
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
```

# Example: Using C malloc() and free()

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

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