**Index Page No.**

Operation on Bits---------------------------------------------------------------30

**ARRAYS And POINTERS**

1- Arrays are set of similar data types

2- However big an array its elements are always stored in contiguous memory locations.

3- Initialize an array while declaring it:

int num[6] = { 2, 4, 12, 5, 45, 5 } ;

int n[ ] = { 2, 4, 12, 5, 45, 5 } ;

float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;

4- The following operations can be performed on a pointer:

(a) Addition of a number to a pointer.

(b) Subtraction of a number from a pointer. For example,

(c) Subtraction of one pointer from another. Subtraction of one pointer from another. One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements.

(d) Comparison of two pointer variables Pointer variables can be compared provided both variables

point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparison can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (usually expressed as NULL).

5- Do not attempt the following operations on

pointers... they would never work out:

(a) Addition of two pointers

(b) Multiplication of a pointer with a constant

(c) Division of a pointer with a constant

6- A pointer when incremented always points to an immediately next location of its type.

7- Accessing array elements by pointers is **always** faster than accessing them by subscripts.

8- The following notations are same:

num[i]

\*( num + i )

\*( i + num )

i[num]

9- Two dimensional arrays:

Initialization of 2D arrays:

(a) int num[3][2] = {

{43,56},

{56,54},

{65,98}

};

(b) int num[3][2] = {43, 56, 56, 54, 65, 98};

(c) int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;

Or int num[][2] = {

{43,56},

{56,54},

{65,98}{87,86}

};

whereas,

int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;

int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;

would never work.

10- We have already studied while learning

one-dimensional arrays that **num[i]** is same as **\*( num + i )**.

Similarly, **\*( s[2] + 1 )** is same as, **\*( \*( s + 2 ) + 1 )**.

s[2][1]

\* ( s[2] + 1 )

\* ( \* ( s + 2 ) + 1 )

11- **Pointer to an Array**

int ( \*p )[2] ;

Here **p** is a pointer to an array of two integers. The parentheses in the declaration of **p** are necessary. Absence of them would make **p** an array of 2 integer pointers.

12- **Array of Pointers**

The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

13- **Three-Dimensional Array**

arr[2][3][1]

\*( \*( \*( arr + 2 ) + 3 ) + 1 )

**Strings**

1- Initialization of Strings:char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ; //always end up with a null character(\0).

char name[ ] = "HAESLER" ; C inserts the null character automatically

2- The **%s** used in **printf( )** is a format specification for printing out a string.

printf ( "%s", name ) ; And scanf ( "%s", name ) ;

3- **scanf( )** is not capable of receiving multi-word strings. We can make **scanf( )** accept multi-word strings by writing it in this manner: scanf ( "%[^\n]s", name ) ;

4- The usage of functions **gets( )** and its counterpart **puts( )** is shown below. The only problem is that “one string at a time” but gets() is able to get one string at a time but with multiple word string.

5- we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer.

main( )

{

char str1[ ] = "Hello" ;

char str2[10] ;

char \*s = "Good Morning" ;

char \*q ;

str2 = str1 ; /\* error \*/

q = s ; /\* works \*/

}

Also, once a string has been defined it cannot be initialized to

another set of characters. Unlike strings, such an operation is

perfectly valid with **char** pointers.

main( )

{

char str1[ ] = "Hello" ;

char \*p = "Hello" ;

str1 = "Bye" ; /\* error \*/

p = "Bye" ; /\* works \*/

}

6-

**String Functions Use:**

**strlen** Finds length of a string

**strlwr** Converts a string to lowercase

**strupr** Converts a string to uppercase

**strcat**  Appends one string at the end of another

**strncat** Appends first n characters of a string at the end of another

**strcpy**  Copies a string into another

**strncpy** Copies first n characters of one string into another

**strcmp** Compares two strings

**strncmp**  Compares first n characters of two strings

**strcmpi** Compares two strings without regard to case ("i" denotes that this function ignores case)

**stricmp**  Compares two strings without regard to case (identical to strcmpi)

**strnicmp**  Compares first n characters of two strings without regard to case

**strdup**  Duplicates a string

**strchr** Finds first occurrence of a given character in a string

**strrchr**  Finds last occurrence of a given character in a string

**strstr**  Finds first occurrence of a given string in another string

**strset** Sets all characters of string to a given character

**strnset**  Sets first n characters of a string to a given character

**strrev** Reverses string

It is good to use arrays of pointers than to use arrays of character to store strings( More efficient use of available memory) And it’s easy to manipulate strings by using arrays of pointers.

**Limitation of Array of Pointers to Strings**

When we are using two dimensional array we are at liberty to initialize string at the time of declaration, and also we can receive input from the keyboard using scanf() function. However when we are using array of pointers to the string, we can initialize string at the time of declaration but we cannot receive string form the keyboard using the scanf() function.

**Solution**

#include "alloc.h"

main( )

{

char \*names[6] ;

char n[50] ;

int len, i ;

char \*p ;

for ( i = 0 ; i <= 5 ; i++ )

{

printf ( "\nEnter name " ) ;

scanf ( "%s", n ) ;

len = strlen ( n ) ;

p = malloc ( len + 1 ) ;

strcpy ( p, n ) ;

names[i] = p ;

}

for ( i = 0 ; i <= 5 ; i++ )

printf ( "\n%s", names[i] ) ;

}

This solution suffers in performance because we need to allocate

memory and then do the copying of string for each name received

through the keyboard.

**Structure**

1- Structure is user define data types which hold variables of different data types.

Example of a structure:

Note: It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the ‘form’ of the structure.

main( )

{

//declaration of structure

struct book

{

char name ; // name, price and pages are the elements of structure

float price ;

int pages ;

} ;

struct book b1, b2, b3 ; // Variable of structure

//Access of structure elements

scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;

printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;

}

7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

If we desire , we can declare structure type and structure variable in one statement:

struct book

{

char name ;

float price ;

int pages ;

} b1, b2, b3 ;

or even...

struct

{

char name ;

float price ;

int pages ;

} b1, b2, b3 ;

struct book b1 = { "Basic", 130.00, 550 } ; //Structure can also be initialize where they are declared.

struct book b2 = { "Physics", 150.80, 800 } ;

2- linkfloat( )

{

float a = 0, \*b ;

b = &a ; /\* cause emulator to be linked \*/

a = \*b ; /\* suppress the warning - variable not used \*/

}

Most processors today have a separate instruction set for operating on Floating Point Numbers. For those machines which cannot process Floating point numbers, an emulator is used to do the job.

What is the function **linkfloat( )** doing here? If you don’t define it you are bound to get the error Floating Point Formats Not Linked" with majority of C Compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it

sets a flag to have the linker link in the floating-point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like **scanf( )** and **atof( )**.

3- The values of a structure variable can be assigned to another structure variable of the same type by:

I- Piece-meal copying

II- Using assignment operator

struct employee

{

char name[10] ;

int age ;

float salary ;

} ;

struct employee e1 = { "Sanjay", 30, 5500.50 } ;

struct employee e2, e3 ;

/\* piece-meal copying \*/

strcpy ( e2.name, e1.name ) ;

e2.age = e1.age ;

e2.salary = e1.salary ;

/\* copying all elements at one go \*/

e3 = e2 ;

C does not allow assigning the contents of one array to another just by equating the two. This copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations.

4- One structure can be nested within another structure. Using this facility complex data types can be created.

main( )

{

struct address

{

char phone[15] ;

char city[25] ;

int pin ;

} ;

struct emp

{

char name[25] ;

struct address a ;

} ;

struct emp e = { "jeru", "531046", "nagpur", 10 };

printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;

printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;

}

We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

maruti.engine.bolt.large.qty

6- Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go.

/\* Passing individual structure elements \*/

main( )

{

struct book

{

char name[25] ;

char author[25] ;

int callno ;

} ;

struct book b1 = { "Let us C", "YPK", 101 } ;

display ( b1.name, b1.author, b1.callno ) ;

}

display ( char \*s, char \*t, int n )

{

printf ( "\n%s %s %d", s, t, n ) ;

}

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time.

struct book

{

char name[25] ;

char author[25] ;

int callno ;

} ;

main( )

{

struct book b1 = { "Let us C", "YPK", 101 } ;

display ( b1 ) ;

}

display ( struct book b )

{

printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;

}

Having collected what is being passed to the **display( )** function, the question comes, how do we define the formal arguments in the function. We cannot say, struct book b1 ; because the data type **struct book** is not known to the function **display( )**. Therefore, it becomes necessary to define

the structure type **struct book** outside **main( )**, so that it becomes known to all functions in the program.

7- we can have a pointer pointing to a **struct**. Such pointers are known as ‘structure pointers’.

main( )

{

struct book

{

char name[25] ;

char author[25] ;

int callno ;

} ;

struct book b1 = { "Let us C", "YPK", 101 } ;

struct book \*ptr ;

ptr = &b1 ;

printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;

printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;

}

C provides an operator **->**, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the **‘**.**’** structure operator, there must always be a structure variable, whereas on the left hand side of the ‘**->’** operator there must always be a pointer to a structure.

8- **IMP**

Consider the following code snippet:

struct emp

{

int a ;

char ch ;

float s ;

} ;

struct emp e ;

printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;

If we execute this program using TC/TC++ compiler we get the addresses as:

65518 65520 65521

As expected, in memory the **char** begins immediately after the **int** and **float** begins immediately after the **char**

However, if we run the same program using VC++ compiler then the output turns out to be:

1245044 1245048 1245052

It can be observed from this output that the **float** doesn’t get stored immediately after the **char**. In fact there is a hole of three bytes after the **char**. Let us understand the reason for this. VC++ is a 32-bit compiler targeted to generate code for a 32-bit microprocessor. The architecture of this microprocessor is such that it is able to fetch the data that is present at an address, which is a multiple of four much faster than the data present at any other address. Hence the VC++ compiler aligns

every element of a structure at an address that is multiple of four. That’s the reason why there were three holes created between the **char** and the **float**.

The **#pragma pack** directive offers a solution:

Turbo C/C++

compiler doesn’t support this feature, VC++ compiler does.

#pragma pack(1)

struct emp

{

int a ;

char ch ;

float s ;

} ;

**Console Input/Output**

C has no provision for input and output, it uses operating system‘s input and output facility by linking the C compiler with the Os input/output facility with the help of standard I/O function. (The standard I/O library has some little program that link the c compiler with the Os input/output facility).

There are two broad categories of Input/Output function:

1- Console I/O function

2- File I/O function

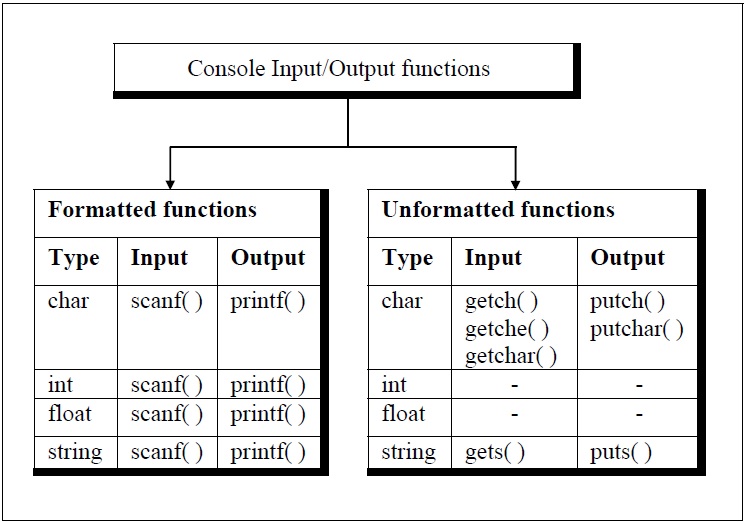
**Console I/O function**

In console I/O function we can further divide it into two categories:

i- Formatted I/O function

ii- Unformatted I/O function

Formatted I/O function is used when we want to get the output in the VDU as per our requirement. For ex: where we want to display the output on the screen, or after how many spaces or after how many line we want to display the output, how may decimal point should be displayed, etc.



Let us now learn more about **printf()** function:

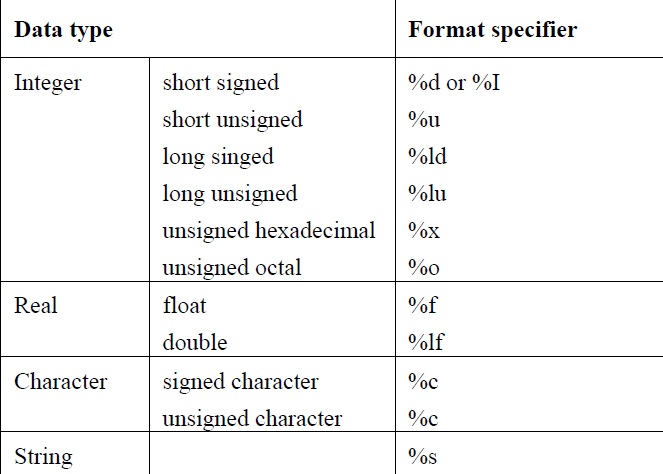
printf(“Format String”,list of variables);

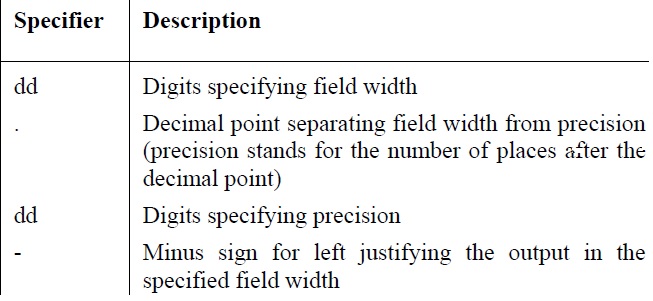
In the format string, the characters are printed as they are, conversion specification starts with a % sign and escape sequence begins with a \ sign.

How does the **printf()** function prints the format string?

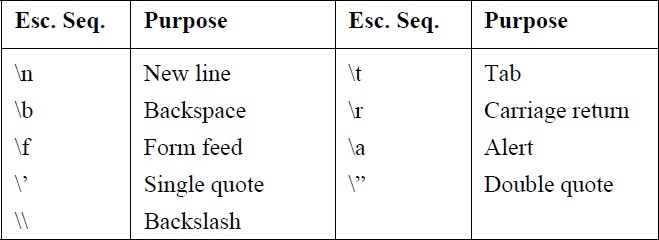
As long as the **printf()** function encounters the character in the format string, it just dumps those character to the screen and when it encounter the conversion specification in the format string, it picks up the first variable in the list of variables and prints its value. Also when it encounter a escape sequence it takes the appropriate action. The process continues till the end of the format string.

**Format Specification:**





**Escape Sequence**



So far we are using **printf()**’s specification as if we are forced to use %d for integers, %c of characters, %s for strings and so on…but the fact is **printf()** get the specification and attempts to perform the specified conversion and tries its best to produce a proper result.

The **scanf()** function:

**scanf**(“format string”,list of variables);

The & operator is use in scanf() function so that the value passed from the keyboard is dropped in the corresponding address of the variables.

***sprintf( )* and *sscanf( )* Functions**

**sprint()** and **sscanf()** are same as **printf() and scanf(),** but with a small difference. Instead of sending the output to the screen as **printf( )** does, this function writes the output to an

array of characters.

Example:

main( )

{

int i = 10 ;

char ch = 'A' ;

float a = 3.14 ;

char str[20] ;

printf ( "\n%d %c %f", i, ch, a ) ;

sprintf ( str, "%d %c %f", i, ch, a ) ;

printf ( "\n%s", str ) ;

}

The counterpart of **sprintf( )** is the **sscanf( )** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats.

**Unformatted Console I/O Functions:**

just remember: Unformatted console I/O functions work faster since they do not have the overheads of formatting the input or output.

**File Input/Output**

# include "stdio.h"

main( )

{

FILE \*fp ;

char ch ;

fp = fopen ( "PR1.C", "r" ) ;

while ( 1 )

{

ch = fgetc ( fp ) ;

if ( ch == EOF ) //EOF file is a special character, its ASCII value is 26

break ;

printf ( "%c", ch ) ;

}

fclose ( fp ) ;

}

fopen() function does three tasks:

(a) First it search the file on the disk.

(b) After the search is completed, it loads the file from the hard disk to a place in the memory called buffer.

(c) Then it sets the file pointer to the first character of the buffer.

Why do we need a buffer at all?

The reason is, whenever anything is read from the hard disk, it takes some time for the reposition of the read/write head correctly. Imagine how difficult and time consuming it would be if each and every character is being read one by one. That why it is easier to store the file in the buffer and then read the content of the file from the buffer.

The same process is applied while writing something on to the file , instead of writing one character at a time, the contents are written on to the buffer and then transfer to the disk.

Mode of opening, size of the file, place from which the reading or writing should start etc all the information are gathered by the fopen() function using a structure called **FILE**. fopen() returns the address of the structure to a structure pointer.

The **FILE** structure has been defined in the stdio.h.

After opening the file from the, we use the **fgetc()** function to read from the file contents.

fgets() read the pointer from its current position and advances the file pointer to the next character and returns the read character to the variable.

Note: When the file is opened, we did not refer the file by its name, we refer it by its file pointer name( in our case it is ‘fp’).

If for some reason, fopen() function cannot open the file. It return a value NULL(define in stdio.h as #define NULL 0. So, to know if fopen() failed to open a file, we need to add:

fp=fopen(“file\_name”,”mode”);

if(fp==0)

{

printf(“cannot open the file\n”);

exit();

}

In the end we need to close the file, and to close the file we use fclose() function. when we close the file buffer associated with the file is removed from the memory. Suppose we intended to write something on a file and open the file in writing mode, while closing the file using the fclose() finction. Three things are going to happen:

a- Contents of the buffer will be written in the file on the disk.

b- At the end of the file, a character with the ASCII value 26 would get written.

c- The buffer will be removed from the memory.

If the buffer gets full before closing the file, then the contents of the buffer will automatically be written into the file and this is managed by the library functions.

The counter part of **fgetc()** is **fputc().** fgetc() accept only one parameter where as fputc() accepts two parameters.

for example: ch=fgetc(fp) , fputc(ch,ft) //where ch is a character variable and ft and fp is a pointer to a structure.

**File Opening Modes**

**"r"** Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened **fopen( )** returns NULL.Operations possible – reading from the file.

"**w"** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file. Operations possible – writing to the file.

**"a"** Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file. Operations possible - adding new contents at the end of file.

**"r+"** Searches file. If is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file. Operations possible - reading existing contents, writing new contents, modifying existing contents of the file.

**"w+"** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file. Operations possible - writing new contents, reading them back and modifying existing contents of the file.

**"a+"** Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file. Operations possible - reading existing contents, appending new contents to end of file. Cannot modify existing contents.

**The Awkward New Line**

When we will try to count the characters in a file using the character count program, we will get for ex: 100 characters and when we use the os program to count the characters, it will say something like 104 characters. It is because, when we use the fputs() function to write to a file, it always convert ‘\n’ into ‘\r\n’ and when we try to read that file by using fgets() function, it converts the ‘\r\n’ again into ‘\n’. This conversion is a feature of the standard library function and not that of Os. Hence the OS counts the ‘\r’ and ‘\n’ as separate characters.

**Record I/O in Files**

If we have to read data from a file or to write data to a file, and we also want to read or write combination of string, number, and floats. The we have to organize this data type together by making a structure and then we have to use **fprintf()** and **fscanf().**

/\* Writes records to a file using structure \*/

#include "stdio.h"

main( )

{

FILE \*fp ;

char another = 'Y' ;

struct emp

{

char name[40] ;

int age ;

float bs ;

} ;

struct emp e ;

fp = fopen ( "EMPLOYEE.DAT", "w" ) ;

if ( fp == NULL )

{

puts ( "Cannot open file" ) ;

exit( ) ;

}

while ( another == 'Y' )

{

printf ( "\nEnter name, age and basic salary: " ) ;

scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;

fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;

printf ( "Add another record (Y/N) " ) ;

fflush ( stdin ) ;

another = getche( ) ;

}

fclose ( fp ) ;

}

/\* Read records from a file using structure \*/

#include "stdio.h"

main( )

{

FILE \*fp ;

struct emp

{

char name[40] ;

int age ;

float bs ;

} ;

struct emp e ;

fp = fopen ( "EMPLOYEE.DAT", "r" ) ;

if ( fp == NULL )

{

puts ( "Cannot open file" ) ;

exit( ) ;

}

while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )

printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

fclose ( fp ) ;

}

**Text files and Binary files**

#include "stdio.h"

main( )

{

FILE \*fs, \*ft ;

int ch ;

fs = fopen ( "pr1.exe", "rb" ) ;

if ( fs == NULL )

{

puts ( "Cannot open source file" ) ;

exit( ) ;

}

ft = fopen ( "newpr1.exe", "wb" ) ;

if ( ft == NULL )

{

puts ( "Cannot open target file" ) ;

fclose ( fs ) ;

exit( ) ;

}

while ( 1 )

{

ch = fgetc ( fs ) ;

if ( ch == EOF )

break ;

else

fputc ( ch, ft ) ;

}

fclose ( fs ) ;

fclose ( ft ) ;

}

There are three main areas where text file are different from binary files.

(a) Handling of newline

(b) Representation of End of file

(c) Storage of numbers

**Newline**

we already know that in a text file when \n is written to the file, it convert into \r\n and when it is read from the disk by using fgetc() library function of C. \r\n is again converted to \n. However in binary no conversion take place.

**End of file**

As we know that, at the end of a text file there is a special character EOF, which has a ASCII value of 26. However, in binary file there is no special character at the end of the file. Binary mode keeps track of the end of file by the number of character present in the directory entry of the binary file.

So, we should know that, when we write data in binary mode, we should read the data back in binary mode, because if we might store a number 26 (hexadecimal 1A) and read back this data in text mode, than when we encounter the number, the reading will be terminated prematurely. So, the modes are not compatible. We should write in binary and read it back using binary mode, and same rule is applied for text mode.

**Storage of numbers**

Only fprint() function is available to store numbers in a file. We know that character takes one byte per character in memory and when written in a text file , it is the same as one byte per character. However numbers, like integers are store in memory as 4 bytes per integer, but when it is stored in a text file, it occupies 1 byte per number. So, 135325 will occupy 4 bytes in memory, but in a text file , it will occupy 6 bytes. So, storing large numbers using text mode is inefficient. The solution is to use fwrite() and fread() (discussed later) which store the numbers in binary format. Means each number will occupy same bytes as it occupies in the memory.

**Record I/O in binary**

Previous record program was not very efficient, because it was writing data in text mode and the integers are converted into characters and taking each no. as one byte. There is a more efficient way of storing data by using the **fwrite() and fread().**

#include "stdio.h"

main( )

{

FILE \*fp ;

char another = 'Y' ;

struct emp

{

char name[40] ;

int age ;

float bs ;

} ;

struct emp e ;

fp = fopen ( "EMP.DAT", "wb" ) ;

if ( fp == NULL )

{

puts ( "Cannot open file" ) ;

exit( ) ;

}

while ( another == 'Y' )

{

printf ( "\nEnter name, age and basic salary: " ) ;

scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;

fwrite ( &e, sizeof ( e ), 1, fp ) ;

printf ( "Add another record (Y/N) " ) ;

fflush ( stdin ) ;

another = getche( ) ;

}

fclose ( fp ) ;

}

Now, in this program the file is open in binary mode and we used the **fwrite()** function to write data on files.

fwrite(&e, sizeof(e), 1, fp);

The first argument is the address of the structure, and the second argument is the size of the structure in bytes. we used the **sizeof()** function to calculate the size of the structure, so that we don’t have to change the second argument, each time the structure element is changed.

The third argument is the number of structure we want to write at one time. In this case we want to write one structure at a time.

The last argument is the pointer to the file we want to write.

/\* Reads records from binary file and displays them on VDU \*/

#include "stdio.h"

main( )

{

FILE \*fp ;

struct emp

{

char name[40] ;

int age ;

float bs ;

} ;

struct emp e ;

fp = fopen ( "EMP.DAT", "rb" ) ;

if ( fp == NULL )

{

puts ( "Cannot open file" ) ;

exit( ) ;

}

while ( fread ( &e, sizeof ( e ), 1, fp ) == 1 )

printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

fclose ( fp ) ;

}

The **fread()** function reads the data from the file stored on the disk. The format of **fread()** is the same as **fwrite().** If **fread()** reached the end of file, means if it cannot read any data. It will return 0. By testing this situation we know when to stop reading. The **fread()** and **fwrite()** function is used for database management. It store the numbers more efficiently on the disk and reading/writing of structure become easy. Note that even if the structure elements belonging to the structure increases , the format of **fread()** and **fwrite()** function remains the same.

**Database Management**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void clean\_strm(FILE \*strm);

void clean\_strm(FILE \*strm)

{

int c;

do

{

c=getchar();

}while(c!='\n' && c!=EOF);

}

void main()

{

FILE \*fp,\*ft;

char fname[30],soption,option,sname[40];

int recsize,i,j;

#pragma pack(1)

struct student

{

char name[40];

int id;

float result;

};

struct student s;

recsize=sizeof(s);

printf("1: For Adding Record\n2: For Listiing Record\n3: For Modifying Record\n4: Deleting Record\n0: To exit\n");

soption=getchar();

clean\_strm(stdin);

puts("Enter the Data file name:");

scanf("%s",fname);

fp=fopen(fname,"rb+");

if(fp==NULL)

{

fp=fopen(fname,"wb+");

if(fp==NULL)

{

puts("Cannot open the file");

exit(0);

}

}

switch(soption)

{

case '1':

option='y';

rewind(fp);

while(option=='y')

{

printf("Enter Student's Name,id And Result:\n");

scanf("%s %d %f",s.name,&s.id,&s.result);

fseek(fp,0,SEEK\_END);

fwrite(&s,recsize,1,fp);

clean\_strm(stdin);

puts("To add Y/N");

option=getchar();

clean\_strm(stdin);

}

fclose(fp);

break;

case '2':

rewind(fp);

while(fread(&s,recsize,1,fp)==1)

{

printf("%10s %10d %10.2f\n\n",s.name,s.id,s.result);

}

fclose(fp);

break;

case '3':

option='y';

while(option=='y')

{

rewind(fp);

puts("Enter the name of the student to modify info");

scanf("%s",sname);

clean\_strm(stdin);

for(rewind(fp);fread(&s,recsize,1,fp)==1;)

{

if(strcmp(s.name,sname)==0)

{

puts("Match found");

puts("Enter student's Name,id And Result to modify");

scanf("%s %d %f",s.name,&s.id,&s.result);

clean\_strm(stdin);

fseek(fp,-recsize,SEEK\_CUR);

fwrite(&s,recsize,1,fp);

}

}

puts("To continue Y/N");

option=getchar();

clean\_strm(stdin);

}

fclose(fp);

break;

case '4':

option='y';

while(option=='y')

{

printf("Enter the name of the student to delete record\n");

scanf("%s",sname);

ft=fopen("temp","wb");

rewind(fp);

while(fread(&s,recsize,1,fp)==1)

{

if(strcmp(s.name,sname)!=0)

{

fwrite(&s,recsize,1,ft);

}

}

fclose(fp);

fclose(ft);

remove(fname);

rename("temp",fname);

fp=fopen(fname,"rb+");

puts("Delete another record");

option=getchar();

clean\_strm(stdin);

}

fclose(fp);

default:

puts("Exiting the program");

exit(0);

}

}

A pointer is initiated whenever we open a file. On opening a file a pointer is set up to the first structure of the file. On using the **fread()** or **fwrite()** function , the pointer moves to the beginning of the next record. On closing the file the pointer is deactivated. Note that the pointer position is important , because the **fread()** function read the data from the record where the pointer is currently placed, similarly the **fwrite()**  function writes the data into the record where the pointer is currently placed.

The **rewind()** function place the pointer to the beginning of the file . Irrespective of where the pointer is present right now.

The **fseek()** function move the pointer from one record to another record.

for example:

fseek ( fp, 0, SEEK\_END ) ;

fseek ( fp, -recsize, SEEK\_CUR ) ; //recsize=sizeof(e);

recsize and SEEK\_END is just the offset that tell the compiler how many bytes the pointer should move from a particular position.

**SEEK**\_**END** means move the pointer from the end of the file, **SEEK**\_**CUR** means move the pointer with reference to its current position and **SEEK**\_**SET** means move the pointer with reference to the beginning of the file.

If we wish to know where the pointer is positioned right now, we can use the function **ftell( )**. It returns this position as a **long int** which is an offset from the beginning of the file. The value

returned by **ftell( )** can be used in subsequent calls to **fseek( )**. A sample call to **ftell( )** is shown below: position = ftell ( fp ) ;

where **position** is a **long int**.

**Low level disk I/O**

In low level disk i/o the data is not written in character or as strings, the data is written and read in lower level disk I/O function in buffer full of bytes.

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

void clean(void);

void clean()

{

int i;

do{

i=getchar();

}while(i!='\n'&&i!=EOF);

}

void main(void)

{

char buffer[512],source[128],target[128];

int src\_fp,trgt\_fp,bytes;

puts("Enter the name of the file");

scanf("%[^\n]s",source);

clean();

puts("Enter name of the file to copy");

scanf("%[^\n]s",target);

clean();

src\_fp=open(source,O\_RDONLY);

if(src\_fp==-1)

{

puts("cannot opent the file");

exit(0);

}

trgt\_fp=open(target,O\_CREAT|O\_WRONLY|S\_IWRITE);

if(trgt\_fp==-1)

{

puts("cannot open the file");

exit(0);

}

while(1)

{

bytes=read(src\_fp,buffer,512);

if(bytes>0)

write(trgt\_fp,buffer,bytes);

else

break;

}

close(src\_fp);

close(trgt\_fp);

}

**Declaring the buffer:**

We have declare a character buffer, by saying:

char buffer[512];

This is the buffer in which the data read from the disk will be placed. The size of the buffer is important for efficient operation. Depending on the Operating System certain buffer size is more efficient than others.

**Opening a File:**

We have to open two files in our program to copy the file. First to open the source file from which the data will be read and Second to open the target file in which we will copy the data from the source file. Just like in high level I/O function we have to open the file. In low level I/O function. We will open the file using:

src\_fp=open(source,O\_RDONLY);

We open the file for the same reason as we did earlier – to establish communication with operating system about the file. We have to supply the filename and the mode in which we want to open the file. The possible file opening modes are given below:

O\_APPEND - Opens a file for appending

O\_CREAT - Creates a new file for writing (has no effect if file already exists)

O\_RDONLY - Creates a new file for reading only

O\_RDWR - Creates a file for both reading and writing

O\_WRONLY - Creates a file for writing only

O\_BINARY - Creates a file in binary mode

O\_TEXT - Creates a file in text mode

The O flags are defined in the fcntl.h file, so fcntl.h file is important to include in the program while using low level disk I/O. When two or more flags are used together, they are combine using the bitwise OR ( | ) operator.

Whenever O\_CREATE flag is used another argument must be added to the **open()** function to indicate read or write status fo the file to be created. This argument is called ‘permission argument’

Following are the permission arguments:

S\_IWRITE - Writing to the file permitted

S\_IREAD - Reading from the file permitted

To use this permission arguments we must use both the file types.h and stat.h in the program.

On opening a file using **open()** function, it return a integer value called ‘file handle’ , this is a number assigned to a particular file and is used thereafter as reference to the file. If **open()** function return a

-1 value, that means the file could not open successfully.

**Interaction between Buffer and File**

The following statement reads the file or as much of it as will fit into the buffer:

bytes = read ( inhandle, buffer, 512 ) ;

The **read( )** function takes three arguments. The first argument is the file handle, the second is the address of the buffer and the third is the maximum number of bytes we want to read. The **read( )** function returns the number of bytes actually read. This is an important number, since it may very well be less than the buffer size (512 bytes), and we will need to know just how full the buffer is before we can do anything with its contents. In our program we have assigned this number to the variable **bytes**. For copying the file, we must use both the **read( )** and the **write( )** functions in a **while** loop. The **read( )** function returns the number of bytes actually read. This is assigned to the variable **bytes**. This value will be equal to the buffer size (512 bytes) until the end of

file, when the buffer will only be partially full. The variable **bytes** therefore is used to tell **write( )**, as to how many bytes to write from the buffer to the target file.

Note that when large buffers are used they must be made global variables otherwise stack overflow occurs.

**More issues in input/output**

***Using argc and argv***

#include <stdio.h>

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

{

FILE \*fs, \*ft ;

char ch ;

if ( argc != 3 )

{

puts ( "Improper number of arguments" ) ;

exit( ) ;

}

fs = fopen ( argv[1], "r" ) ;

if ( fs == NULL )

{

puts ( "Cannot open source file" ) ;

exit( ) ;

}

ft = fopen ( argv[2], "w" ) ;

if ( ft == NULL )

{

puts ( "Cannot open target file" ) ;

fclose ( fs ) ;

exit( ) ;

}

while ( 1 )

{

ch = fgetc ( fs ) ;

if ( ch == EOF )

break ;

else

fputc ( ch, ft ) ;

}

fclose ( fs ) ;

fclose ( ft ) ;

}

The argument we pass in main are called command line arguments. **main()** function can have two argument **argc** anf **argv**. **argv** is an array of pointer to the string and **argc** is a integer variable which is equal to number of string to which **argv** points. To be more precise the string is stored in the memory and the address of the first string is passed to **argv[0]** and the address of the second string is passed to **argv[1]** and so on. The **argc** is automatically set to the number of string passed the command line.

Whenever we pass argument to the **main()** it is a good habit to check if the proper number of argument is passed the command line or not.

There is one more way to write the code for file-copy program:

while(!feof(fs))

{

ch=fget(fs);

fputc(ch,ft);

}

The **feof** is a macro which return 0 if the end of file is not reached and we are using the **!** operator to reverse the 0 to a true value and when the end of file will be reached, the **feof()** will return a true value and it will become a zero value and the loop will be terminated.

**Detecting errors in Reading/Writing**

The standard I\O library function **ferror()** reports any error that might have occurred during a read\write operation on a file. It return zero if a read\write function was successful or return a non-zero value if it was a failure.

#include "stdio.h"

main( )

{

FILE \*fp ;

char ch ;

fp = fopen ( "TRIAL", "w" ) ;

while ( !feof ( fp ) )

{

ch = fgetc ( fp ) ;

if ( ferror( ) )

{

printf ( "Error in reading file" ) ;

break ;

}

else

printf ( "%c", ch ) ;

}

fclose ( fp ) ;

}

When we will run this program it will get an error because the file was opened in writing mode and the **fgetc()** was attempting to read from the file. The **ferror()** function will return a non-zero value and the if block will get executed. Instead of displaying the error using **fprintf()**, we can use **perror()** function, it automatically prints the error specified by the compiler.Thus in the above program the **perror( )**.

function can be used as shown below.

if ( ferror( ) )

{

perror ( "TRIAL" ) ;

break ;

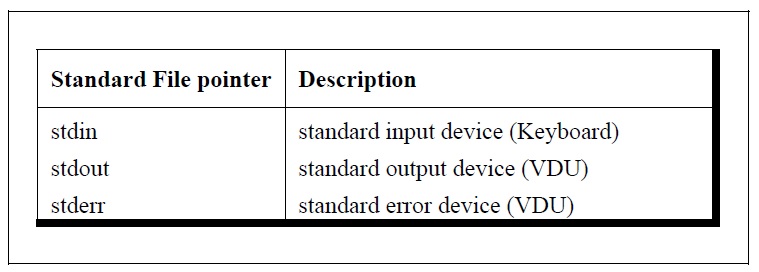
}

Note that when the error occurs the error message that is displayed is:

TRIAL: Permission denied

**Standard I/O Devices**

To read or write to a file we have to use the **fopen()** function, which sets a file pointer to the file. But in most OS there are some predefined standard file and to do operation on these file we do not need to open the file using **fopen()** function. The standard pointer are shown below:



Thsu, **ch=fgetc(stdin)**  will read the characters from the keyboard rather than from a file. We can use this statement with the need to open or close the file using **fopen()** or **fclose()** function.

**I/O Redirction**

Most OS has include a powerful feature which give the ability to read and write files even when the program has no such features in it. This is called redirection.

Normally, C take input from the standard input device which is usually the keyboard and give output to the standard output device which is assume to be the VDU(Visual Display Unit). In other words the operating system make assumption about where input should come from and where output should go. We can change these assumption using the redirection.

For example, the output going to the VDU can be redirected to the printer or to the disk without making any change in the program. Similarly, the redirection can also be used to directly take the input from the disk file rather than taking the input from the keyboard.

**Redirecting the output:**

#include<stdio.h>

int main()

{

char ch;

while(ch=fgetc(stdin) != EOF)

fputc(ch,stdout);

}

This program will print whatever user type in the keyboard.

Now, we will use redirection in the terminal to redirect the output to a file. To do this we have to type: **./samp.c>file.txt**

**Redirecting the input:**

we just need to type in the terminal:

./samp<file.txt

**Both ways indirection:**

A program can use redirection in both input and output, such program is called filter.

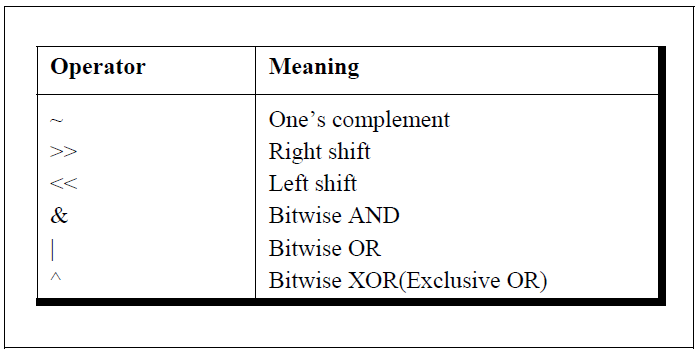
Example:

./samp<input.txt>output.txt

Don’t try to do both the output and the input operation on the same file. Because the output file is first erased before anything is written. So, when we manage to receive input from the file it will already be erased.

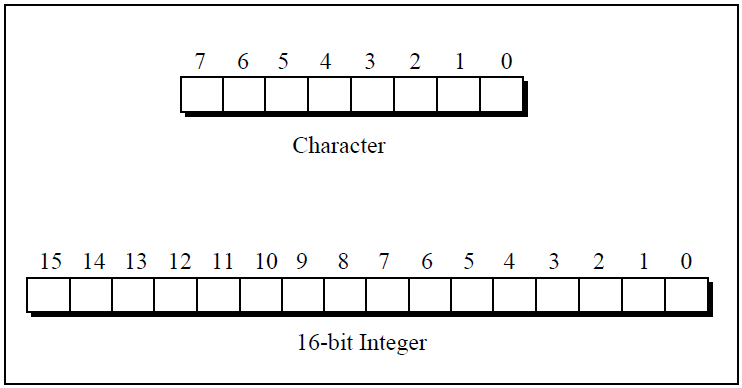
**Operation on Bits**

Programming languages are byte oriented whereas the hardware is bit oriented. One of the C most powerful features is the manipulation on bit. Means a programmer can manipulate individual bit within a piece of data. The various c bitwise operators are:



This operator can operate on **int**s and **char**s, but not on **float**s and **double**s.

This is how bits are numbered:



From right to left as shown.

**Showbits Function**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define BITS\_IN\_BYTE 8

#define INTEGRAL\_TYPE unsigned int

void showbits(INTEGRAL\_TYPE x)

{

int i;

static int intSizeInBits = sizeof(INTEGRAL\_TYPE) \* BITS\_IN\_BYTE;

static char symbol[2] = {'0','1'};

char \* binary = (char\*) malloc(intSizeInBits + 1);

memset(binary, 0, intSizeInBits + 1);

for (i=0; i< intSizeInBits; i++)

{

binary[intSizeInBits-i-1] = symbol[(x>>i) & 0x01];

}

printf("%s\n", binary);

free(binary);

}

**One’s Compliment Operator**

Taking one’s compliment of a number all 1s in the number changes to 0s and all the 0s changes to 1s. For Example: One’s complement of 1101 is 0010. Similarly one’s compliment of 1111 is 0000.

One’s compliment operator is represented by the symbol ‘~ ’.

main( )

{

int j, k ;

for ( j = 0 ; j <= 3 ; j++ )

{

printf ( "\nDecimal %d is same as binary ", j ) ;

showbits ( j ) ;

k = ~j ;

printf ( "\nOne’s complement of %d is ", j ) ;

showbits ( k ) ;

}

}

Since the one’s compliment operator changes the original data beyond recognition , it can be used in real world for file encryption. Below is an simple example:

#include "stdio.h"

main( )

{

encrypt( ) ;

}

encrypt( )

{

FILE \*fs, \*ft ;

char ch ;

fs = fopen ( "SOURCE.C", "r" ) ; /\* normal file \*/

ft = fopen ( "TARGET.C”, "w" ) ; /\* encrypted file \*/

if ( fs == NULL || ft == NULL )

{

printf ( "\nFile opening error!" ) ;

exit ( 1 ) ;

}

while ( ( ch = getc ( fs ) ) != EOF )

putc ( ~ch, ft ) ;

fclose ( fs ) ;

fclose ( ft ) ;

}

**Right Shift Operator**

The right shift operator is represented by >>. It needs two operands. It shift each bit in its left operand to its right. The number of bits which will be shifted is depend on the number following the operand.

For example: If the bit pattern of a variable **ch** is 11001101, then **ch>>1** will change the bit pattern to 01100110. Similarly, **ch>>3** will change the bit pattern to 00011001. Note that: bits are shifted to the right and blanks are created on the left. These blanks should be filled somehow and they are always filled with zeros.

**A word of caution:**

If **a>>b**, and if **b** is negative , the result is unpredictable. If **a** is negative than its left most bit (sign bit) would be 1. On some computer right shift a would result in extending the sign bit. For example, If **a** contains -1, its binary representation would be 11111111. Without sign extension, the operation **a>>4** would be 00001111. However, on the machine on which we executed this expression the result turns out to be 11111111. Thus the sign bit 1 continues to get extended.

**Left Shift Operator**

This is similar to the right shift operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 (zero) is added to the right of the number.

For example: 00001011 is a bit pattern for variable **a.** when left shifted to **ch<<1**, the bit pattern becomes 00010110. For **a<<3**, it will be 01011000.

Here is a program for left shift operation:

main( )

{

int i = 5225, j, k ;

printf ( "\nDecimal %d is same as ", i ) ;

showbits ( i ) ;

for ( j = 0 ; j <= 4 ; j++ )

{

k = i <<j ;

printf ( "\n%d left shift %d gives ", i, j ) ;

showbits ( k ) ;

}

}

**Bitwise AND Operator**

This operator is represented as **&**. The **&** operator operates on two operands. While operating upon these two operands they are compared on a bit by bit basis. Hence both the operand must be of the same type (either char or int). The second operand is often called an AND mask. The & operator operates on a pair of bits to yield a resultant bit. The rules that decide the value of the resultant bit are shown below:

|  |  |  |
| --- | --- | --- |
| First bit | Second bit | First & Second bit |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Truth Table:

|  |  |
| --- | --- |
| & | 0 1 |
| 0 | 0 0 |
| 1 | 0 1 |