**AALIM MUHAMMED SALEGH COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER APPLICATION**

**SEMESTER- I**

**MC5102 – PROBLEM SOLVING AND PROGRAMMING**

**UNIT – V NOTES**

**SYLLABUS:**

**UNIT V USER-DEFINED DATATYPES & FILES**

Structures – initialization - nested structures – structures and arrays – structures and pointers – union – typedef and enumeration types - bit fields - File Management in C – Files and Streams – File handling functions – Sequential access file- Random access file – Command line arguments.

**Structures**

* Arrays allow defining type of variables that can hold several data items of the same kind.
* Similarly structure is another user defined data type available in C that allows combining data items of different kinds.
* Structures are used to represent a record.

For example:

  struct gun

{

 char name[50];

 int magazinesize;

 float calibre;

};

struct gun arnies;

defines a new structure gun and makes arnies an instance of it.

NOTE: that gun is a tag for the structure that serves as shorthand for future declarations. We now only need to say struct gun and the body of the structure is implied as we do to make the arnies variable. The tag is optional.

Variables can also be declared between the } and ; of a struct declaration, i.e.:

  struct gun

{

 char name[50];

 int magazinesize;

 float calibre;

} arnies;

Suppose you want to keep track of your books in a library.

You might want to track the following attributes about each book −

Title

Author

Subject

Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

struct [structure tag] {

 member definition;

 member definition;

 ...

 member definition;

} [one or more structure variables];

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

Here is the way you would declare the Book structure −

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

} book;

**Initialization**

struct's can be pre-initialised at declaration:

  struct gun arnies={"Uzi",30,7};

which gives arnie a 7mm. Uzi with 30 rounds of ammunition.

To access a member (or field) of a struct, C provides the . operator. For example, to give arnie more rounds of ammunition:

   arnies.magazineSize=100;

**Accessing Structure Members**

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program −

#include <stdio.h>

#include <string.h>

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

};

int main( ) {

 struct Books Book1; /\* Declare Book1 of type Book \*/

 struct Books Book2; /\* Declare Book2 of type Book \*/

 /\* book 1 specification \*/

 strcpy( Book1.title, "C Programming");

 strcpy( Book1.author, "Nuha Ali");

 strcpy( Book1.subject, "C Programming Tutorial");

 Book1.book\_id = 6495407;

 /\* book 2 specification \*/

 strcpy( Book2.title, "Telecom Billing");

 strcpy( Book2.author, "Zara Ali");

 strcpy( Book2.subject, "Telecom Billing Tutorial");

 Book2.book\_id = 6495700;

 /\* print Book1 info \*/

 printf( "Book 1 title : %s\n", Book1.title);

 printf( "Book 1 author : %s\n", Book1.author);

 printf( "Book 1 subject : %s\n", Book1.subject);

 printf( "Book 1 book\_id : %d\n", Book1.book\_id);

 /\* print Book2 info \*/

 printf( "Book 2 title : %s\n", Book2.title);

 printf( "Book 2 author : %s\n", Book2.author);

 printf( "Book 2 subject : %s\n", Book2.subject);

 printf( "Book 2 book\_id : %d\n", Book2.book\_id);

 return 0;

}

When the above code is compiled and executed, it produces the following result −

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

#include <stdio.h>

#include <string.h>

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

};

/\* function declaration \*/

void printBook( struct Books book );

int main( ) {

 struct Books Book1; /\* Declare Book1 of type Book \*/

 struct Books Book2; /\* Declare Book2 of type Book \*/

 /\* book 1 specification \*/

 strcpy( Book1.title, "C Programming");

 strcpy( Book1.author, "Nuha Ali");

 strcpy( Book1.subject, "C Programming Tutorial");

 Book1.book\_id = 6495407;

 /\* book 2 specification \*/

 strcpy( Book2.title, "Telecom Billing");

 strcpy( Book2.author, "Zara Ali");

 strcpy( Book2.subject, "Telecom Billing Tutorial");

 Book2.book\_id = 6495700;

 /\* print Book1 info \*/

 printBook( Book1 );

 /\* Print Book2 info \*/

 printBook( Book2 );

 return 0;

}

void printBook( struct Books book ) {

 printf( "Book title : %s\n", book.title);

 printf( "Book author : %s\n", book.author);

 printf( "Book subject : %s\n", book.subject);

 printf( "Book book\_id : %d\n", book.book\_id);

}

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

**Pointers to Structures (or) Structures and pointers**

We can define pointers to structures in the same way as you define pointer to any other variable −

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable.

To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

struct\_pointer->title;

Let us re-write the above example using structure pointer.

#include <stdio.h>

#include <string.h>

struct Books {

 char title[50];

 char author[50];

 char subject[100];

 int book\_id;

};

/\* function declaration \*/

void printBook( struct Books \*book );

int main( ) {

 struct Books Book1; /\* Declare Book1 of type Book \*/

 struct Books Book2; /\* Declare Book2 of type Book \*/

 /\* book 1 specification \*/

 strcpy( Book1.title, "C Programming");

 strcpy( Book1.author, "Nuha Ali");

 strcpy( Book1.subject, "C Programming Tutorial");

 Book1.book\_id = 6495407;

 /\* book 2 specification \*/

 strcpy( Book2.title, "Telecom Billing");

 strcpy( Book2.author, "Zara Ali");

 strcpy( Book2.subject, "Telecom Billing Tutorial");

 Book2.book\_id = 6495700;

 /\* print Book1 info by passing address of Book1 \*/

 printBook( &Book1 );

 /\* print Book2 info by passing address of Book2 \*/

 printBook( &Book2 );

 return 0;

}

void printBook( struct Books \*book ) {

 printf( "Book title : %s\n", book->title);

 printf( "Book author : %s\n", book->author);

 printf( "Book subject : %s\n", book->subject);

 printf( "Book book\_id : %d\n", book->book\_id);

}

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

**Unions**

A union is a variable which may hold (at different times) objects of different sizes and types.

A union is a special data type available in C that allows to store different data types in the same memory location.

You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

**Defining a Union**

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

union [union tag] {

 member definition;

 member definition;

 ...

 member definition;

} [one or more union variables];

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

union Data {

 int i;

 float f;

 char str[20];

} data;

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

#include <stdio.h>

#include <string.h>

union Data {

 int i;

 float f;

 char str[20];

};

int main( ) {

 union Data data;

 printf( "Memory size occupied by data : %d\n", sizeof(data));

 return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by data : 20

**Accessing Union Members**

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program −

#include <stdio.h>

#include <string.h>

union Data {

 int i;

 float f;

 char str[20];

};

int main( ) {

 union Data data;

 data.i = 10;

 data.f = 220.5;

 strcpy( data.str, "C Programming");

 printf( "data.i : %d\n", data.i);

 printf( "data.f : %f\n", data.f);

 printf( "data.str : %s\n", data.str);

 return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

#include <stdio.h>

#include <string.h>

union Data {

 int i;

 float f;

 char str[20];

};

int main( ) {

 union Data data;

 data.i = 10;

 printf( "data.i : %d\n", data.i);

 data.f = 220.5;

 printf( "data.f : %f\n", data.f);

 strcpy( data.str, "C Programming");

 printf( "data.str : %s\n", data.str);

 return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

C uses the union statement to create unions, for example:

 union number

 {

 short shortnumber;

 long longnumber;

 double floatnumber;

 } anumber

defines a union called number and an instance of it called anumber. number is a union tag and acts in the same way as a tag for a structure.

Members can be accessed in the following way:

          printf("%ld\n",anumber.longnumber);

This clearly displays the value of longnumber.

When the C compiler is allocating memory for unions it will always reserve enough room for the largest member (in the above example this is 8 bytes for the double).

**typedef and enumeration types**

**Defining New Data Types**

typedef can also be used with structures. The following creates a new type agun which is of type struct gun and can be initialised as usual:

  typedef struct gun

 {

 char name[50];

 int magazinesize;

 float calibre;

 } agun;

 agun arnies={"Uzi",30,7};

Here gun still acts as a tag to the struct and is optional. Indeed since we have defined a new data type it is not really of much use,

agun is the new data type. arnies is a variable of type agun which is a structure.

C also allows arrays of structures:

  typedef struct gun

{

 char name[50];

 int magazinesize;

 float calibre;

} agun;

agun arniesguns[1000];

This gives arniesguns a 1000 guns. This may be used in the following way:

arniesguns[50].calibre=100;

gives Arnie's gun number 50 a calibre of 100mm, and:

itscalibre=arniesguns[0].calibre;

assigns the calibre of Arnie's first gun to itscalibre.

**Coercion or Type-Casting**

C is one of the few languages to allow coercion, that is forcing one variable of one type to be another type. C allows this using the cast operator (). So:

  int integernumber;

 float floatnumber=9.87;

 integernumber=(int)floatnumber;

assigns 9 (the fractional part is thrown away) to integernumber.

And:

 int integernumber=10;

 float floatnumber;

 floatnumber=(float)integernumber;

assigns 10.0 to floatnumber.

Coercion can be used with any of the simple data types including char, so:

 int integernumber;

 char letter='A';

 integernumber=(int)letter;

assigns 65 (the ASCII code for `A') to integernumber.

Some typecasting is done automatically -- this is mainly with integer compatibility.

A good rule to follow is: If in doubt cast.

Another use is the make sure division behaves as requested: If we have two integers internumber and anotherint and we want the answer to be a float then :

e.g.

 floatnumber =

  (float) internumber / (float) anotherint;

ensures floating point division.

**Enumerated Types**

Enumerated types contain a list of constants that can be addressed in integer values.

We can declare types and variables as follows.

  enum days {mon, tues, ..., sun} week;

  enum days week1, week2;

NOTE: As with arrays first enumerated name has index value 0. So mon has value 0, tues 1, and so on.

week1 and week2 are variables.

We can define other values:

  enum escapes { bell = `\a',

 backspace = `\b', tab = `\t',

 newline = `\n', vtab = `\v',

 return = `\r'};

We can also override the 0 start value:

  enum months {jan = 1, feb, mar, ......, dec};

Here it is implied that feb = 2 etc.

**Exercises**

Exercise

Write program using enumerated types which when given today's date will print out tomorrow's date in the for 31st January, for example.

Exercise

Write a simple database program that will store a persons details such as age, date of birth, address etc.

**Bit Fields**

**Bit Fields**

***Bit Fields*** allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

* Packing several objects into a machine word. ***e.g.*** 1 bit flags can be compacted -- Symbol tables in compilers.
* Reading external file formats -- non-standard file formats could be read in. ***E.g.*** 9 bit integers.

C lets us do this in a structure definition by putting :***bit length*** after the variable. ***i.e.***

struct packed\_struct {

 unsigned int f1:1;

 unsigned int f2:1;

 unsigned int f3:1;

 unsigned int f4:1;

 unsigned int type:4;

 unsigned int funny\_int:9;

} pack;

Here the packed\_struct contains 6 members: Four 1 bit ***flags*** f1..f3, a 4 bit type and a 9 bit funny\_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit fiels portability below).

Access members as usual via:

   pack.type = 7;

**NOTE**:

* Only *n* lower bits will be assigned to an *n* bit number. So type cannot take values larger than 15 (4 bits long).
* Bit fields are always converted to integer type for computation.
* You are allowed to mix "normal" types with bit fields.
* The unsigned definition is important - ensures that no bits are used as a  flag.

**Bit Fields: Practical Example**

Frequently device controllers (***e.g.*** disk drives) and the operating system need to communicate at a low level. Device controllers contain several ***registers*** which may be packed together in one integer (Figure [12.1](https://users.cs.cf.ac.uk/Dave.Marshall/C/node13.html#fig:drive)).

 

**Fig.**[**12.1**](https://users.cs.cf.ac.uk/Dave.Marshall/C/node13.html#fig:drive)**Example Disk Controller Register** We could define this register easily with bit fields:

struct DISK\_REGISTER {

 unsigned ready:1;

 unsigned error\_occured:1;

 unsigned disk\_spinning:1;

 unsigned write\_protect:1;

 unsigned head\_loaded:1;

 unsigned error\_code:8;

 unsigned track:9;

 unsigned sector:5;

 unsigned command:5;

};

To access values stored at a particular memory address, DISK\_REGISTER\_MEMORY we can assign a pointer of the above structure to access the memory via:

struct DISK\_REGISTER \*disk\_reg = (struct DISK\_REGISTER \*) DISK\_REGISTER\_MEMORY;

The disk driver code to access this is now relatively straightforward:

/\* Define sector and track to start read \*/

disk\_reg->sector = new\_sector;

disk\_reg->track = new\_track;

disk\_reg->command = READ;

/\* wait until operation done, ready will be true \*/

while ( ! disk\_reg->ready ) ;

/\* check for errors \*/

if (disk\_reg->error\_occured)

 { /\* interrogate disk\_reg->error\_code for error type \*/

 switch (disk\_reg->error\_code)

 ......

 }

**A note of caution: Portability**

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

* integers may be signed or unsigned
* Many compilers limit the maximum number of bits in the bit field to the size of an integer which may be either 16-bit or 32-bit varieties.
* Some bit field members are stored left to right others are stored right to left in memory.
* If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

If portability of code is a premium you can use bit shifting and masking to achieve the same results but not as easy to express or read. For example:

unsigned int \*disk\_reg = (unsigned int \*) DISK\_REGISTER\_MEMORY;

/\* see if disk error occured \*/

disk\_error\_occured = (disk\_reg & 0x40000000) >> 31;

**Exercises**

**Exercise 12507**

 Write a function that prints out an 8-bit (unsigned char) number in binary format.

**Exercise 12514**

Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of an unsigned char variable y (leaving other bits unchanged).

E.g. if *x* = 10101010 (170 decimal) and *y* = 10100111 (167 decimal) and *n* = 3 and *p* = 6 say then you need to strip off 3 bits of y (111) and put them in x at position 10*xxx*010 to get answer 10111010.

Your answer should print out the result in binary form (see Exercise [12.1](https://users.cs.cf.ac.uk/Dave.Marshall/C/node13.html#ex:bin) although input can be in decimal form.

Your output should be like this:

 x = 10101010 (binary)

 y = 10100111 (binary)

 setbits n = 3, p = 6 gives x = 10111010 (binary)

**Exercise 12515**

Write a function that inverts the bits of an unsigned char x and stores answer in y.

Your answer should print out the result in binary form (see Exercise [12.1](https://users.cs.cf.ac.uk/Dave.Marshall/C/node13.html#ex:bin) although input can be in decimal form.

Your output should be like this:

 x = 10101010 (binary)

 x inverted = 01010101 (binary)

**Exercise 12516**

Write a function that rotates (**NOT shifts**) to the right by n bit positions the bits of an unsigned char x.ie no bits are lost in this process.

Your answer should print out the result in binary form (see Exercise [12.1](https://users.cs.cf.ac.uk/Dave.Marshall/C/node13.html#ex:bin) although input can be in decimal form.

Your output should be like this:

 x = 10100111 (binary)

 x rotated by 3 = 11110100 (binary)

**Note**: All the functions developed should be as concise as possible

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

struct {

 unsigned int widthValidated;

 unsigned int heightValidated;

} status;

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

struct {

 unsigned int widthValidated : 1;

 unsigned int heightValidated : 1;

} status;

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept −

#include <stdio.h>

#include <string.h>

/\* define simple structure \*/

struct {

 unsigned int widthValidated;

 unsigned int heightValidated;

} status1;

/\* define a structure with bit fields \*/

struct {

 unsigned int widthValidated : 1;

 unsigned int heightValidated : 1;

} status2;

int main( ) {

 printf( "Memory size occupied by status1 : %d\n", sizeof(status1));

 printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

 return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by status1 : 8

Memory size occupied by status2 : 4

**Bit Field Declaration**

The declaration of a bit-field has the following form inside a structure −

struct {

 type [member\_name] : width ;

};

The following table describes the variable elements of a bit field −

|  |  |
| --- | --- |
| Elements | Description |
| type | An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| member\_name | The name of the bit-field. |
| width | The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

struct {

 unsigned int age : 3;

} Age;

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

#include <stdio.h>

#include <string.h>

struct {

 unsigned int age : 3;

} Age;

int main( ) {

 Age.age = 4;

 printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

 printf( "Age.age : %d\n", Age.age );

 Age.age = 7;

 printf( "Age.age : %d\n", Age.age );

 Age.age = 8;

 printf( "Age.age : %d\n", Age.age );

 return 0;

}

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

struct {

 unsigned int widthValidated;

 unsigned int heightValidated;

} status;

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

struct {

 unsigned int widthValidated : 1;

 unsigned int heightValidated : 1;

} status;

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept −

#include <stdio.h>

#include <string.h>

/\* define simple structure \*/

struct {

 unsigned int widthValidated;

 unsigned int heightValidated;

} status1;

/\* define a structure with bit fields \*/

struct {

 unsigned int widthValidated : 1;

 unsigned int heightValidated : 1;

} status2;

int main( ) {

 printf( "Memory size occupied by status1 : %d\n", sizeof(status1));

 printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

 return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by status1 : 8

Memory size occupied by status2 : 4

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

struct {

 type [member\_name] : width ;

};

The following table describes the variable elements of a bit field −

|  |  |
| --- | --- |
| Sr.No. | Element & Description |
| 1 | typeAn integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| 2 | member\_nameThe name of the bit-field. |
| 3 | widthThe number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

struct {

 unsigned int age : 3;

} Age;

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

#include <stdio.h>

#include <string.h>

struct {

 unsigned int age : 3;

} Age;

int main( ) {

 Age.age = 4;

 printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

 printf( "Age.age : %d\n", Age.age );

 Age.age = 7;

 printf( "Age.age : %d\n", Age.age );

 Age.age = 8;

 printf( "Age.age : %d\n", Age.age );

 return 0;

}

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

**Files in C**

C files I/O functions handles data on secondary storage device, such as a hard disk.

C can handle files as Stream-oriented data (Text) files and System oriented data (Binary) files.

|  |  |
| --- | --- |
| Stream oriented data files | The data is stored in same manner as it appears on the screen. The I/O operations like buffering, data conversions etc. take place automatically. |
| System oriented data files | System-oriented data files are more closely associated with the OS and data stored in memory without converting into text format. |

**C File Operations**

There are five major operations that can be performed on a file are:

1. Creation of a new file.
2. Opening an existing file.
3. Reading data from a file.
4. Writing data in a file.
5. Closing a file.

**Steps for Processing a File**

1. Declare a file pointer variable.
2. Open a file using fopen() function.
3. Process the file using suitable function.
4. Close the file using fclose() function.

**File operation functions in C**

To handling files in C, file input/output functions available in the stdio library are:

|  |  |
| --- | --- |
| **Function** | **Uses/Purpose** |
| [fopen](https://www.w3schools.in/c-tutorial/file-handling/fopen/) | Opens a file. (or) Creates a new file for use (or) Opens a new existing file for use. |
| [fclose](https://www.w3schools.in/c-tutorial/file-handling/fclose/) | Closes a file. (or) Closes a file which has been opened for use |
| [getc](https://www.w3schools.in/c-tutorial/file-handling/getc/) | Reads a character from a file. (or) Reads a character from a file |
| [putc](https://www.w3schools.in/c-tutorial/file-handling/putc/) | Writes a character to a file (or) Writes a character to a file |
| [getw](https://www.w3schools.in/c-tutorial/file-handling/getw/) | Read integer (or) Reads a integer from a file |
| [putw](https://www.w3schools.in/c-tutorial/file-handling/putw/) | Write integer (or) Writes an integer to the file |
| [fprintf](https://www.w3schools.in/c-tutorial/file-handling/fprintf/) | Prints formatted output to a file (or) Writes a set of data values to a file |
| [fscanf](https://www.w3schools.in/c-tutorial/file-handling/fscanf/) | Reads formatted input from a file (or) Reads a set of data values from a file |
| [fgets](https://www.w3schools.in/c-tutorial/file-handling/fgets/) | Read string of characters from file |
| [fputs](https://www.w3schools.in/c-tutorial/file-handling/fputs/) | Write string of characters to file |
| [feof](https://www.w3schools.in/c-tutorial/file-handling/feof/) | Detects end-of-file marker in a file |
| fseek() | Sets the position to a desired point in the file |
| Ftell() | Gives the current position in the file |
| rewind() | Sets the position to the begining of the file. |

C supports a number of functions that have the ability to perform basic file operations, which include:

1. Naming a file

2. Opening a file

3. Reading from a file

4. Writing data into a file

5. Closing a file

* Real life situations involve large volume of data and in such cases, the console oriented I/O operations pose two major problems
* It becomes cumbersome and time consuming to handle large volumes of data through terminals.
* The entire data is lost when either the program is terminated or computer is turned off therefore it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data.

**Defining and opening a file:**

* If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system.
* They include the fielname, data structure, purpose.

The general format of the function used for opening a file is

FILE \*fp;

fp=fopen(“filename”,”mode”);

* The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library.
* The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.
* This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.
* The second statement also specifies the purpose of opening the file. The mode does this job.

**File opening Mode:**

R open the file for read only.

W open the file for writing only.

A open the file for appending data to it.

Consider the following statements:

FILE \*p1, \*p2;

p1=fopen(“data”,”r”);

p2=fopen(“results”,”w”);

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

**Closing a file:**

The input output library supports the function to close a file; it is in the following format.

fclose(file\_pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.

Observe the following program.

FILE \*p1 \*p2;

p1=fopen (“Input”,”w”);

p2=fopen (“Output”,”r”);

….

…

fclose(p1);

fclose(p2)

* The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.
* The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time.
* The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

**File operations**

The program shown below displays use of a file operations.

The data enter through the keyboard and the program writes it. Character by character, to the file input.

The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

#include< stdio.h >

main()

{

file \*f1;

printf(“Data input output”);

f1=fopen(“Input”,”w”); /\*Open the file Input\*/

while((c=getchar())!=EOF) /\*get a character from key board\*/

putc(c,f1); /\*write a character to input\*/

fclose(f1); /\*close the file input\*/

printf(“\nData output\n”);

f1=fopen(“INPUT”,”r”); /\*Reopen the file input\*/

while((c=getc(f1))!=EOF)

printf(“%c”,c);

fclose(f1);

}

**The getw and putw functions:**

These are integer-oriented functions.

They are similar to get c and putc functions and are used to read and write integer values.

These functions would be usefull when we deal with only integer data.

The general forms of getw and putw are:

putw(integer,fp);

getw(fp);

/\*Example program for using getw and putw functions\*/

#include< stdio.h >

main()

{

FILE \*f1,\*f2,\*f3;

int number, i;

printf(“Contents of the data file\n\n”);

f1=fopen(“DATA”,”W”);

for(i=1;i< 30;i++)

{

scanf(“%d”,&number);

if(number==-1)

break;

putw(number,f1);

}

fclose(f1);

f1=fopen(“DATA”,”r”);

f2=fopen(“ODD”,”w”);

f3=fopen(“EVEN”,”w”);

while((number=getw(f1))!=EOF) /\* Read from data file\*/

{

if(number%2==0)

putw(number,f3);/\*Write to even file\*/

else

putw(number,f2);/\*write to odd file\*/

}

fclose(f1);

fclose(f2);

fclose(f3);

f2=fopen(“ODD”,”r”);

f3=fopen(“EVEN”,”r”);

printf(“\n\nContents of the odd file\n\n”);

while(number=getw(f2))!=EOF)

printf(“%d%d”,number);

printf(“\n\nContents of the even file”);

while(number=getw(f3))!=EOF)

printf(“%d”,number);

fclose(f2);

fclose(f3);

}

**The fprintf & fscanf functions:**

The fprintf and scanf functions are identical to printf and scanf functions except that they work on files. The first argument of theses functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,”control string”, list);

Where fp id a file pointer associated with a file that has been opened for writing.

The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f”,name,age,7.5);

Here name is an array variable of type char and age is an int variable

The general format of fscanf is

fscanf(fp,”controlstring”,list);

This statement would cause the reading of items in the control string.

Example:

fscanf(f2,”5s%d”,item,&quantity”);

Like scanf, fscanf also returns the number of items that are successfully read.

/\*Program to handle mixed data types\*/

#include< stdio.h >

main()

{

FILE \*fp;

int num,qty,I;

float price,value;

char item[10],filename[10];

printf(“Input filename”);

scanf(“%s”,filename);

fp=fopen(filename,”w”);

printf(“Input inventory data\n\n”0;

printf(“Item namem number price quantity\n”);

for( i=1;i< =3;i++)

{

fscanf(stdin,”%s%d%f%d”,item,&number,&price,&quality);

fprintf(fp,”%s%d%f%d”,itemnumber,price,quality);

}

fclose (fp);

fprintf(stdout,”\n\n”);

fp=fopen(filename,”r”);

printf(“Item name number price quantity value”);

for(I=1;I< =3;I++)

{

fscanf(fp,”%s%d%f%d”,item,&number,&prince,&quality);

value=price\*quantity”);

fprintf(“stdout,”%s%d%f%d%d\n”,item,number,price,quantity,value);

}

fclose(fp);

}

This C program merges two files and stores their contents in another file.

The files which are to be merged are opened in read mode and the file which contains content of both the files is opened in write mode.

To merge two files first we open a file and read it character by character and store the read contents in another file then we read the contents of another file and store it in file, we read two files until EOF (end of file) is reached.

Example:

#include<stdio.h>

#include<stdlib.h>

int main()

{

 FILE \*fs1, \*fs2, \*ft;

 char ch, file1[20], file2[20], file3[20];

 printf("Enter name of first file\n");

 gets(file1);

 printf("Enter name of second file\n");

 gets(file2);

 printf("Enter name of file which will store contents of two files\n");

 gets(file3);

 fs1 = fopen(file1,"r");

 fs2 = fopen(file2,"r");

 if( fs1 == NULL || fs2 == NULL )

 {

 perror("Error ");

 printf("Press any key to exit...\n");

 getch();

 exit(EXIT\_FAILURE);

 }

 ft = fopen(file3,"w");

 if( ft == NULL )

 {

 perror("Error ");

 printf("Press any key to exit...\n");

 exit(EXIT\_FAILURE);

 }

 while( ( ch = fgetc(fs1) ) != EOF )

 fputc(ch,ft);

 while( ( ch = fgetc(fs2) ) != EOF )

 fputc(ch,ft);

 printf("Two files were merged into %s file successfully.\n",file3);

 fclose(fs1);

 fclose(fs2);

 fclose(ft);

 return 0;

}

Program Output:



**Program to count number of lines in the file**

Number of lines in a file can be determined by counting the number of new line characters present.

#include <stdio.h>

int main()

/\* Ask for a filename and count number of lines in the file\*/

{

 //a pointer to a FILE structure

 FILE \*fp;

 int no\_lines = 0;

 //consider 40 character string to store filename

 char filename[40], sample\_chr;

 //asks user for file name

 printf("Enter file name: ");

 //receives file name from user and stores in a string named 'filename'

 scanf("%s", filename);

 //open file in read mode

 fp = fopen(filename, "r");

 //get character from file and store in sample\_chr

 sample\_chr = getc(fp);

 while (sample\_chr != EOF) {

 //Count whenever sample\_chr is blank (new line) is encountered

 if (sample\_chr == '')

 {

 //increment variable 'no\_lines' by 1

 no\_lines=no\_lines+1;

 }

 //take next character from file.

 sample\_chr = getc(fp);

 }

 fclose(fp); //close file.

 printf("There are %d lines in %s

 ", no\_lines, filename);

 return 0;

}

**Program Output:**

Enter file name:abc.txt

There are 4 lines in abc.txt

**Explanation:**

* In this program, name of the file to be read is taken as input. A file by the given name is opened in read-mode using a File pointer ‘fp’.
* Characters from the file are read into a char variable ‘sample\_chr’ with the help of getc function. If a new line character(‘ ‘) is encountered, the integer variable ‘no\_lines’ is incremented.
* If the character read into ‘sample\_char’ is not a new line character, next character is read from the file.
* This process is continued until the last character of the file(EOF) is encountered.
* The file pointer is then closed and the total number of lines is shown as output.

**C program for deletes a file**

This C program deletes a file which is entered by the user, the file to be deleted should be present in the directory in which the executable file of this program is present.

Extension of the file should also be entered, remove macro is used to delete the file. If there is an error in deleting the file then an error will be displayed using perror function.

Example:

#include<stdio.h>

main()

{

 int status;

 char file\_name[25];

 printf("Enter the name of file you wish to delete\n");

 gets(file\_name);

 status = remove(file\_name);

 if( status == 0 ){

 printf("%s file deleted successfully.\n",file\_name);

 }

 else

 {

 printf("Unable to delete the file\n");

 perror("Error");

 }

 return 0;

}

Program Output:



Deleted file doesn’t go to trash or recycle bin so you may not be able to recover it. Deleted files can be recovered using special recovery software if the files are not overwritten on the storage medium.

**Files**

Files are the most common form of a stream.

The first thing we must do is ***open*** a file. The function fopen() does this:

   FILE \*fopen(char \*name, char \*mode)

fopen returns a pointer to a FILE. The name string is the name of the file on disc that we wish to access. The mode string controls our type of access. If a file cannot be accessed for any reason a NULL pointer is returned.

Modes include: "r" -- read,

 "w" -- write and

 "a" -- append.

To open a file we must have a stream (file pointer) that ***points*** to a FILE structure.

So to open a file, called ***myfile.dat*** for reading we would do:

  FILE \*stream, \*fopen();

 /\* declare a stream and prototype fopen \*/

 stream = fopen("myfile.dat","r");

it is good practice to to check file is opened

correctly:

 if ( (stream = fopen( "myfile.dat",

 "r")) == NULL)

 {  printf("Can't open %s\n"",

 "myfile.dat");

   exit(1);

 }

 ......

**Reading and writing files**

The functions fprintf and fscanf a commonly used to access files.

int fprintf(FILE \*stream, char \*format, args..)

 int fscanf(FILE \*stream, char \*format, args..)

These are similar to printf and scanf except that data is read from the ***stream*** that must have been opened with fopen().

The stream pointer is automatically incremented with ALL file read/write functions. We **do not** have to worry about doing this.

  char \*string[80]

 FILE \*stream, \*fopen();

 if ( (stream = fopen(...)) != NULL)

 fscanf(stream,"%s", string);

Other functions for files:

int getc(FILE \*stream), int fgetc(FILE \*stream)

int putc(char ch, FILE \*s), int fputc(char ch, FILE \*s)

These are like getchar, putchar.

getc is defined as preprocessor MACRO in stdio.h. fgetc is a C library function. Both achieve the same result!!

   fflush(FILE \*stream) -- flushes a stream.

   fclose(FILE \*stream) -- closes a stream.

We can access predefined streams with fprintf ***etc.***

   fprintf(stderr,"Cannot Compute!!\n");

    fscanf(stdin,"%s",string);

**sprintf and sscanf**

These are like fprintf and fscanf except they read/write to a string.

int sprintf(char \*string, char \*format, args..)

int sscanf(char \*string, char \*format, args..)

For Example:

  float full\_tank = 47.0; /\* litres \*/

 float miles = 300;

 char miles\_per\_litre[80];

 sprintf( miles\_per\_litre,"Miles per litre

 = %2.3f", miles/full\_tank);

**Stream Status Enquiries**

There are a few useful stream enquiry functions, prototyped as follows:

 int feof(FILE \*stream);

 int ferror(FILE \*stream);

 void clearerr(FILE \*stream);

 int fileno(FILE \*stream);

Their use is relatively simple:

**feof()**

-- returns true if the stream is currently at the end of the file. So to read a stream,fp, line by line you could do:

while ( !feof(fp) )

 fscanf(fp,"%s",line);

**ferror()**

-- reports on the error state of the stream and returns true if an error has occurred.

**clearerr()**

-- resets the error indication for a given stream.

**fileno()**

-- returns the integer file descriptor associated with the named stream.

**Low Level I/O**

This form of I/O is UNBUFFERED -- each read/write request results in accessing disk (or device) directly to fetch/put a specific number of **bytes**.

There are no formatting facilities -- we are dealing with bytes of information.

This means we are now using binary (and not text) files.

Instead of file pointers we use ***low level*** file handle or file descriptors which give a unique integer number to identify each file.

To Open a file use:

   int open(char \*filename, int flag, int perms) -- this returns a file descriptor or -1 for a **fail**.

The flag controls file access and has the following predefined in fcntl.h:

   O\_APPEND, O\_CREAT, O\_EXCL, O\_RDONLY, O\_RDWR, O\_WRONLY + others see online man pages or reference manuals.

perms -- best set to 0 for most of our applications.

The function:

   creat(char \*filename, int perms)

can also be used to create a file.

int close(int handle) -- close a file

int read(int handle, char \*buffer,
unsigned length)

int write(int handle, char \*buffer, unsigned length)

are used to read/write a specific number of bytes from/to a file (handle) stored or to be put in the memory location specified by buffer.

The sizeof() function is commonly used to specify the length.

read and write return the number of bytes read/written or -1 if they fail.

/\* program to read a list of floats from a binary file \*/

/\* first byte of file is an integer saying how many \*/

/\* floats in file. Floats follow after it, File name got from \*/

/\* command line \*/

#include<stdio.h>

#include<fcntl.h>

float bigbuff[1000];

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

{  int fd;

 int bytes\_read;

 int file\_length;

 if ( (fd = open(argv[1],O\_RDONLY)) = -1)

 { /\* error file not open \*/....

 perror("Datafile");

 exit(1);

 }

 if ( (bytes\_read = read(fd,&file\_length,

 sizeof(int))) == -1)

 { /\* error reading file \*/...

 exit(1);

 }

 if ( file\_length > 999 ) {/\* file too big \*/ ....}

 if ( (bytes\_read = read(fd,bigbuff,

 file\_length\*sizeof(float))) == -1)

 { /\* error reading open \*/...

 exit(1);

 }

}

**Exercises**

1. Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.

Copy the file a block (512 bytes) at a time.

Check: that the program has two arguments

 or print "Program need two arguments"

 that the first name file is readable

 or print "Cannot open file .... for reading"

 that the second file is writable

 or print "Cannot open file .... for writing"

**Exercise 12577**

Write a program last that prints the last *n* lines of a text file, by *n* and the file name should be specified form command line input. By default *n* should be 5, but your program should allow an optional argument so that

 last -n file.txt

prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

**Exercise 12578**

Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.

**Command Line Arguments**

* 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 argumentsand 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 −

#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

**STRUCTURE**

· Structure is the collection of variables of different types under a single name for better

handling.

· Arrays allow defining type of variables that can hold several data items of the **same kind but**

**structure allows combining data items** of different kinds.

**Structure Definition in C**

Keyword struct is used for creating a structure.

**Syntax of structure**

struct structure\_name

{

data\_type member1;

data\_type member2;

.

.

data\_type memeber;

};

**Example to create the structure for a person as mentioned above as:**

struct person

{

char name[50];

int cit\_no;

float salary;

};

This declaration above creates the derived data type struct person.

**Structure variable declaration**

When a structure is defined, it creates a user-defined type but, no storage is allocated. For

the above structure of person, variable can be declared as:

struct person

{

char name[50];

int cit\_no;

float salary;

};

**Inside main function:**

**struct person** p1, p2, p[20];

**Another way of creating sturcture variable is:**

struct person

{

char name[50];

int cit\_no;

float salary;

}p1 ,p2 ,p[20];

In both cases, 2 variables *p1*, *p2* and array *p* having 20 elements of type **struct person**

are created.

**Accessing members of a structure**

· There are two types of operators used for accessing members of a structure.

**i. Member operator(.)**

· Any member of a structure can be accessed as: structure\_variable\_name.member\_name

· Suppose, we want to access salary for variable *p2*. Then, it can be accessed as:

**p2.salary**

· **Example:**

#include<stdio.h>

struct Vehicle

{

int wheels;

char vname[20];

char color[10];

}v1 = {4,"Nano","Red"};

int main()

{

printf("Vehicle No of Wheels : %d",v1.wheels);

printf("Vehicle Name : %s",v1.vname);

printf("Vehicle Color : %s",v1.color);

return(0);

}

Output :

Vehicle No of Wheels : 4

Vehicle Name : Nano

Vehicle Color : Red

**ii. Structure pointer operator(->)**

#include<stdio.h>

#include<malloc.h>

struct emp {

int eid;

char name[10];

}\*ptr;

int main() {

int i;

printf("Enter the Employee Details : ");

ptr = (struct emp \*) malloc(sizeof(struct emp));

printf("\nEnter the Employee ID : ");

scanf("%d", &ptr->eid);

printf("\nEnter the Employee Name : ");

scanf("%s", ptr->name);

printf("\nEmployee Details are : ");

printf("\nRoll Number : %d", ptr->eid);

printf("\nEmployee Name : %s", ptr->name);

return (0);

}

Output :

Enter the Employee Details :

Enter the Employee ID : 1

Enter the Employee Name : Pritesh

Employee Details are :

Employee ID : 1

Name : Pritesh

**Example of C Program on Student details using structure.**

/\* Student details using structure \*/

#include<stdio.h>

#include<conio.h>

struct stu

{ int stuno;

char sname[10];

int mar;

} ;

void main()

{ struct stu s[10];

int n,i;

clrscr();

printf("\n how many records");

scanf("%d",&n);

printf("\n enter the records");

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

scanf("%d%s%d",&s[i].stuno,s[i].sname,&s[i].mar);

printf("\n entered records are");

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

printf("\n %d\n %s\n %d",s[i].stuno,s[i].sname,s[i].mar);

getch();

}

**Keyword typedef while using structure**

· Programmer generally uses typedef while using structure in C language. For example:

typedef struct complex{

int imag;

float real;

}comp;

**Inside main:**

comp c1,c2;

· Here, typedef keyword is used in creating a type comp(which is of type as struct

complex). Then, two structure variables c1 and c2 are created by this comp type.

**Structures within structures**

**Consider**





**Consider an example to access structure's member through pointer.**

#include <stdio.h>

struct name{

int a;

float b;

};

int main(){

struct name \*ptr,p;

ptr=&p; /\* Referencing pointer to memory address of p \*/

printf("Enter integer: ");

scanf("%d",&(\*ptr).a);

printf("Enter number: ");

scanf("%f",&(\*ptr).b);

printf("Displaying: ");

printf("%d%f",(\*ptr).a,(\*ptr).b);

return 0;

}

· In this example, the pointer variable of type struct name is referenced to the address of *p*.

Then, only the structure member through pointer can can accessed.

· Structure pointer member can also be accessed using -> operator.

(\*ptr).a is same as ptr->a

(\*ptr).b is same as ptr->b

**Accessing structure member through pointer using dynamic memory allocation**

· To access structure member using pointers, memory can be allocated dynamically using

malloc() function defined under "stdlib.h" header file.

· Syntax to use malloc()

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

**Example to use structure's member through pointer using malloc() function.**

#include <stdio.h>

#include<stdlib.h>

struct name {

int a;

float b;

char c[30];

};

int main(){

struct name \*ptr;

int i,n;

printf("Enter n: ");

scanf("%d",&n);

ptr=(struct name\*)malloc(n\*sizeof(struct name));

/\* Above statement allocates the memory for n structures with pointer ptr pointing to base

address \*/

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

printf("Enter string, integer and floating number respectively:\n");

scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);

}

printf("Displaying Infromation:\n");

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

printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);

return 0;

}

**Output**

Enter n: 2

Enter string, integer and floating number respectively:

Programming

2

3.2

Enter string, integer and floating number respectively:

Structure

6

2.3

Displaying Information

Programming 2 3.20

Structure 6 2.30

**In C, structure can be passed to functions by two methods:**

· Passing by value (passing actual value as argument)

Passing by reference (passing address of an argument)

**i. Passing structure by value**

· A structure variable can be passed to the function as an argument as normal variable. If

structure is passed by value, change made in structure variable in function definition

does not reflect in original structure variable in calling function.

· Example: **C program to create a structure student, containing name and roll. Ask**

**user the name and roll of a student in main function. Pass this structure to a**

**function and display the information in that function.**

#include <stdio.h>

struct student{

char name[50];

int roll;

};

void Display(struct student stu);

/\* function prototype should be below to the structure declaration otherwise compiler shows

error \*/

int main(){

struct student s1;

printf("Enter student's name: ");

scanf("%s",&s1.name);

printf("Enter roll number:");

scanf("%d",&s1.roll);

Display(s1); // passing structure variable s1 as argument

return 0;

}

void Display(struct student stu){

printf("Output\nName: %s",stu.name);

printf("\nRoll: %d",stu.roll);

}

**Output**

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

**ii. Passing structure by reference**

· The address location of structure variable is passed to function while passing it by

reference.

· If structure is passed by reference, change made in structure variable in function

definition reflects in original structure variable in the calling function.

· Example: **C program to add two distances(feet-inch system) entered by user. To**

**solve this program, make a structure. Pass two structure variable (containing**

**distance in feet and inch) to add function by reference and display the result in**

**main function without returning it.**

#include <stdio.h>

struct distance{

int feet;

float inch;

};

void Add(struct distance d1,struct distance d2, struct distance \*d3);

int main()

{

struct distance dist1, dist2, dist3;

printf("First distance\n");

printf("Enter feet: ");

scanf("%d",&dist1.feet);

printf("Enter inch: ");

scanf("%f",&dist1.inch);

printf("Second distance\n");

printf("Enter feet: ");

scanf("%d",&dist2.feet);

printf("Enter inch: ");

scanf("%f",&dist2.inch);

Add(dist1, dist2, &dist3);

/\*passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by

reference \*/

printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);

return 0;

}

void Add(struct distance d1,struct distance d2, struct distance \*d3)

{

/\* Adding distances d1 and d2 and storing it in d3 \*/

d3->feet=d1.feet+d2.feet;

d3->inch=d1.inch+d2.inch;

if (d3->inch>=12) { /\* if inch is greater or equal to 12, converting it to feet. \*/

d3->inch-=12;

++d3->feet;

}

}

Output

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18'-2.3"

**Explanation**

· In this program, structure variables *dist1* and *dist2* are passed by value (because value of

*dist1* and *dist2* does not need to be displayed in main function) and *dist3* is passed by

reference ,i.e, address of *dist3* (&dist3) is passed as an argument.

· Thus, the structure pointer variable *d3* points to the address of *dist3*. If any change is

made in *d3* variable, effect of it is seed in *dist3* variable in main function.

· Unions are quite similar to the structures in C. Union is also a derived type as structure.

Union can be defined in same manner as structures just the keyword used in defining

union in **union** where keyword used in defining structure was **struct.**

union car{

char name[50];

int price;

};

Union variables can be created in similar manner as structure variable.

union car{

char name[50];

int price;

}c1, c2, \*c3;

OR;

union car{

char name[50];

int price;

};

-------Inside Function-----------

union car c1, c2, \*c3;

· In both cases, union variables *c1*, *c2* and union pointer variable *c3* of type union car is

created.

· Accessing members of an union

· The member of unions can be accessed in similar manner as that structure. Suppose, we

you want to access price for union variable *c1* in above example, it can be accessed as

c1.price. If you want to access price for union pointer variable *c3*, it can be accessed as

(\*c3).price or as c3->price.

**Source Code to Store Information of 10 students Using Structure**

#include <stdio.h>

struct student{

char name[50];

int roll;

float marks;

};

int main(){

struct student s[10];

int i;

printf("Enter information of students:\n");

for(i=0;i<10;++i)

{

s[i].roll=i+1;

printf("\nFor roll number %d\n",s[i].roll);

printf("Enter name: ");

scanf("%s",s[i].name);

printf("Enter marks: ");

scanf("%f",&s[i].marks);

printf("\n");

}

printf("Displaying information of students:\n\n");

for(i=0;i<10;++i)

{

printf("\nInformation for roll number %d:\n",i+1);

printf("Name: ");

puts(s[i].name);

printf("Marks: %.1f",s[i].marks);

}

return 0;

}

**Output**

Enter information of students:

For roll number 1

Enter name: Tom

Enter marks: 98

For roll number 2

Enter name: Jerry

Enter marks: 89

Displaying information of students:

Information for roll number 1:

Name: Tom

Marks: 98

**2. Union**

· C Union is also like structure, i.e. collection of different data types which are grouped

together. Each element in a union is called member.

· Union and structure in C are same in concepts, except allocating memory for their

members.

· Structure allocates storage space for all its members separately.

· **Union allocates one common storage space for all its members**

· One can access only one member of union at a time. We **can’t access all member**

**values at the same time in union.** But, structure can access all member values at the

same time. **Union allocates one common storage space for all its members**.

Wher**ea**s Structure allocates storage space for all its members separately.

· Many union variables can be created in a program and memory will be allocated for each

union variable separately.

· Below table will help you how to form a C union, declare a union, initializing and

accessing the members of the union.

**Example for union**

#include <stdio.h>

union job { //defining a union

char name[32];

float salary;

int worker\_no;

}u;

struct job1 {

char name[32];

float salary;

int worker\_no;

}s;

int main(){

printf("size of union = %d",sizeof(u));

printf("\nsize of structure = %d", sizeof(s));

return 0;

}

**Output**

size of union = 32

size of structure = 40

**Program explanation**

· The amount of memory required to store a structure variables is the sum of memory size

of all members.



But, the memory required to store a union variable is the memory required for largest

element of a union.



**Example :structure and union :**all members of structure can be accessed at any time. But, only

one member of union can be accessed at a time in case of union and other members will contain

garbage value.

#include <stdio.h>

union job {

char name[32];

float salary;

int worker\_no;

}u;

int main(){

printf("Enter name:\n");

scanf("%s",&u.name);

printf("Enter salary: \n");

scanf("%f",&u.salary);

printf("Displaying\nName :%s\n",u.name);

printf("Salary: %.1f",u.salary);

return 0;

}

**Output**

Enter name

Enter salary

1234.23

Displaying

Name: f%Bary

Salary: 1234.2

Note: You may get different garbage value of name.

· Initially, *Hillary* will be stored in u.name and other members of union will contain

garbage value. But when user enters value of salary, 1234.23 will be stored in u.salary

and other members will contain garbage value. Thus in output, salary is printed

accurately but, name displays some random string.

**3. typedef and enumumerated type**

(i) **typedef** is a powerful tool that **allows programmers to define and then use their own data types**.

 For example:

**typedef int Integer;**

**Integer nStudents, nCourses, studentID;**

o Note that in the typedef statement, the newly defined type name goes in the place

where a variable name would normally go.

(ii) An enumeration is a user-defined data type consists of integral constants and each

integral constant is give a name. Keyword enum is used to defined enumerated data type.

**enum type\_name{ value1, value2,...,valueN };**

o Here, *type\_name* is the name of enumerated data type or tag. And *value1*,

*value2*,....,*valueN* are values of type *type\_name*.

o By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the

programmer can change the default value as below:

enum suit{

club=0;

diamonds=10;

hearts=20;

spades=3;

};

**Declaration of enumerated variable :**Variable of type enum can be created as:

enum boolean{

false;

true;

};

enum boolean check;

· Here, a variable check is declared which is of type enum boolean.

**Example1 of enumerated type**

#include <stdio.h>

enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main(){

enum week today;

today=wednesday;

printf("%d day",today+1);

return 0;

}

**Output**

4 day

**Example2 of enumerated type**

#include< stdio.h>

void main()

{

int i;

enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};

clrscr();

for(i=JAN;i<=DEC;i++)

printf("\n%d",i);

}

**Output**

01234567891011

**4. File Management in C**

* In C programming, file is a place on disk where a group of related data is stored.
* A file represents a sequence of bytes, regardless of it being a text file or a binary file.
* C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

**Importance of files .**

When the program is terminated, the entire data is lost in C programming. If you want to

keep large volume of data, it is time consuming to enter the entire data. But, if file is created, this

information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language.

**High level file I/O functions can be categorized as:**

i. Text file

ii. Binary file

**File Operations**

i. Creating a new file

ii. Opening an existing file

iii. Reading from and writing information to a file

iv. Closing a file

**Working with file**

While working with file, you need to declare a pointer of type file. This declaration is

needed for communication between file and program.

FILE \*ptr;

**Opening a file**

Opening a file is performed using library function fopen().

|  |  |
| --- | --- |
| Mode | Description |
| r | Opens an existing text file for reading purpose. |
| w | Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file. |
| a | Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| r+ | Opens a text file for both reading and writing. |
| w+ | Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist. |
| a+ | Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones −

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

 **The syntax for opening a file**

in standard I/O is:

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

OR

FILE \*fopen( const char \* filename, const char \* mode );

Here, filename is a string literal, which you will use to name your file, and access mode

For Example:

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

/\* --------------------------------------------------------- \*/

E:\\cprogram\program.txt is the location to create file.

"w" represents the mode for writing.

/\* --------------------------------------------------------- \*/

Here, the program.txt file is opened for writing mode.

**Closing a File:** The file should be closed after reading/writing of a file.

Closing a file is

performed using library function fclose().

fclose(ptr); //ptr is the file pointer associated with file to be closed.

int fclose( FILE \*fp );

**The Functions fprintf() and fscanf() functions.**

The functions fprintf() and fscanf() are the file version of printf() and fscanf().

The only difference while using fprintf() and fscanf() is that, the first argument is a pointer to the structure

FILE

**Writing to a file**

#include <stdio.h>

int main()

{

int n;

FILE \*fptr;

fptr=fopen("C:\\program.txt","w");

if(fptr==NULL){

printf("Error!");

exit(1);

}

printf("Enter n: ");

scanf("%d",&n);

fprintf(fptr,"%d",n);

fclose(fptr);

return 0;

}

Output:



This program takes the number from user and stores in file.

· After compile and run this program, a text file program.txt created in C drive of the

computer. Similarly, fscanf() can be used to read data from file.

**Reading from file**

#include <stdio.h>

int main()

{

int n;

FILE \*fptr;

if ((fptr=fopen("C:\\program.txt","r"))==NULL){

printf("Error! opening file");

exit(1); /\* Program exits if file pointer returns NULL. \*/

}

fscanf(fptr,"%d",&n);

Value of n=%d",n);

fclose(fptr);

return 0;

}

If you have run program above to write in file successfully, you can get the integer back

entered in that program using this program.

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

**Example:**

You can also use int fscanf(FILE \*fp, const char \*format, ...) function to read strings from a file, but it stops reading after encountering the first space character.

#include <stdio.h>

main() {

 FILE \*fp;

 char buff[255];

 fp = fopen("/tmp/test.txt", "r");

 fscanf(fp, "%s", buff);

 printf("1 : %s\n", buff );

 fgets(buff, 255, (FILE\*)fp);

 printf("2: %s\n", buff );

 fgets(buff, 255, (FILE\*)fp);

 printf("3: %s\n", buff );

 fclose(fp);

}

**Binary Files**

* Depending upon the way file is opened for processing, a file is classified into text file and
* binary file.
* If a large amount of numerical data it to be stored, text mode will be insufficient. In such
* case binary file is used.
* Working of binary files is similar to text files with few differences in opening modes,
* reading from file and writing to file.

**Opening modes of binary files**

* Opening modes of binary files are rb, rb+, wb, wb+,ab and ab+.
* The only difference
* between opening modes of text and binary files is that, b is appended to indicate that, it is binary file.

**Reading and writing of a binary file.**

* Functions fread() and fwrite() are used for reading from and writing to a file on the disk
* respectively in case of binary files.
* Function fwrite() 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 and pointer to the file where you want to write.

 fwrite(address\_data,size\_data,numbers\_data,pointer\_to\_file);

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

**The following example shows the usage of fread() and fwrite() functions.**

#include <stdio.h>

#include <string.h>

int main()

{

FILE \*fp;

char c[] = "Welcome to FXEC";

char buffer[20];

/\* Open file for both reading and writing \*/

fp = fopen("file.txt", "w+");

/\* Write data to the file \*/

fwrite(c, strlen(c) + 1, 1, fp);

/\* Seek to the beginning of the file \*/

fseek(fp, SEEK\_SET, 0);

/\* Read and display data \*/

fread(buffer, strlen(c)+1, 1, fp);

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

fclose(fp);

return(0);

}

**Output**

Welcome to FXEC

**5. File operation functions in C:**

**Function Name Operation**

fopen() Creates a new file for use

 Opens a new existing file for use

Fclose Closes a file which has been opened for use

getc() Reads a character from a file

putc() Writes a character to a file

fprintf() Writes a set of data values to a file

fscanf() Reads a set of data values from a file

getw() Reads a integer from a file

putw() Writes an integer to the file

fseek() Sets the position to a desired point in the file

ftell() Gives the current position in the file

rewind() Sets the position to the begining of the file

**Defining and opening a file:**

If we want to store data in a file into the secondary memory, we must specify certain

things about the file to the operating system.

They include the fielname, data structure, purpose.

· The general format of the function used for opening a file is

FILE \*fp;

fp=fopen(“filename”,”mode”);

* The first statement declares the variable fp as a pointer to the data type FILE.
* As stated earlier, File is a structure that is defined in the I/O Library.
* The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.
* This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.
* The second statement also specifies the purpose of opening the file. The mode does this job.

**R open the file for read only.**

**W open the file for writing only.**

**A open the file for appending data to it.**

Consider the following statements:

FILE \*p1, \*p2;

p1=fopen(“data”,”r”);

p2=fopen(“results”,”w”);

· In these statements the p1 and p2 are created and assigned to open the files data and

results respectively the file data is opened for reading and result is opened for writing.

· In case the results file already exists, its contents are deleted and the files are opened

as a new file. If data file does not exist error will occur.

**Closing a file:**

· The input output library supports the function to close a file; it is in the following

format.

fclose(file\_pointer);

A file must be closed as soon as all operations on it have been completed. This would

close the file associated with the file pointer.

· Observe the following program.

….

FILE \*p1 \*p2;

p1=fopen (“Input”,”w”);

p2=fopen (“Output”,”r”);

….

…

fclose(p1);

fclose(p2)

· The above program opens two files and closes them after all operations on them are

completed, once a file is closed its file pointer can be reversed on other file.

· The getc and putc functions are analogous to getchar and putchar functions and handle

one character at a time.

The putc function writes the character contained in character

variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc

function is used to read a character from a file that has been open in read mode.

c=getc(fp2).

· The program shown below displays use of a file operations.

The data enter through the

keyboard and the program writes it. Character by character, to the file input.

The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

#include< stdio.h >

main()

{

file \*f1;

printf(“Data input output”);

f1=fopen(“Input”,”w”); /\*Open the file Input\*/

while((c=getchar())!=EOF) /\*get a character from key board\*/

putc(c,f1); /\*write a character to input\*/

fclose(f1); /\*close the file input\*/

printf(“\nData output\n”);

f1=fopen(“INPUT”,”r”); /\*Reopen the file input\*/

while((c=getc(f1))!=EOF)

printf(“%c”,c);

fclose(f1);

}

**The getw and putw functions:**

· These are integer-oriented functions.

They are similar to get c and putc functions and

are used to read and write integer values.

These functions would be usefull when we

deal with only integer data.

The general forms of getw and putw are:

putw(integer,fp);

getw(fp);

/\*Example program for using getw and putw functions\*/

#include< stdio.h >

main()

{

FILE \*f1,\*f2,\*f3;

int number I;

printf(“Contents of the data file\n\n”);

f1=fopen(“DATA”,”W”);

for(I=1;I< 30;I++)

{

scanf(“%d”,&number);

if(number==-1)

break;

putw(number,f1);

}

fclose(f1);

f1=fopen(“DATA”,”r”);

f2=fopen(“ODD”,”w”);

f3=fopen(“EVEN”,”w”);

while((number=getw(f1))!=EOF)/\* Read from data file\*/

{

if(number%2==0)

putw(number,f3);/\*Write to even file\*/

else

putw(number,f2);/\*write to odd file\*/

}

fclose(f1);

fclose(f2);

fclose(f3);

f2=fopen(“ODD”,”r”);

f3=fopen(“EVEN”,”r”);

printf(“\n\nContents of the odd file\n\n”);

while(number=getw(f2))!=EOF)

printf(“%d%d”,number);

printf(“\n\nContents of the even file”);

while(number=getw(f3))!=EOF)

printf(“%d”,number);

fclose(f2);

fclose(f3);

}

**6. File access:**

One can access the data stored in the file in two ways.

· **Sequentially:** Data accessed serially. To access the forty fourth record then first forty

three records read **one after the other** to reach forty four records.

· **Randomly:** In random access, data can be accessed and processed **directly**.

There is no need to read each record sequentially. To access a particular record random access takes less time than the sequential access.

**Random access to files:**

· Random access means we can move to any part of a file and read or write data from it

without having to read through the entire file.

· Sometimes it is required to access only a particular part of the and not the complete file.

This can be accomplished by using the following function:

fseek( ) Function

ftell ( ) Function

**i. fseek function:**

· This function is used for setting the file position pointer at the specified bytes.

fseek is a function belonging to the ANCI C standard Library and included in the file stdio.h.

· Its purpose is to **change the file position indicator for the specified stream**.

int fseek(FILE **\*stream\_pointer**, **long offset**, int **origin**);

**Argument meaning:**

o **stream\_pointer** is a pointer to the stream FILE structure of which the

position indicator should be changed;

o **offset** is a long integer which specifies the number of bytes from origin

where the position indicator should be placed;

o **origin** is an integer which specifies the origin position. It can be:

**Constant Description**

SEEK\_SET Beginning of file

SEEK\_CUR Current position of the file pointer

SEEK\_END End of file

**The following example shows the usage of fseek() function.**

#include <stdio.h>

int main ()

{

FILE \*fp;

fp = fopen("file.txt","w+");

fputs("This is a sample program", fp);

fseek( fp, 7, SEEK\_SET );

fputs(" C Programming Langauge", fp);

fclose(fp);

return(0);

}

**Output**

This is C Programming Langauge

**ftell ( ) Function**

The C library function long int ftell(FILE \*stream) returns the current file position of the

given stream.

**Declaration**

· Following is the declaration for ftell() function.

· long int ftell(FILE \*stream)

**Parameters**

· stream -- This is the pointer to a FILE object that identifies the stream.

**Return Value**

· This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

**Example**

#include <stdio.h>

int main ()

{

FILE \*fp;

int len;

fp = fopen("file.txt", "r");

if( fp == NULL )

{

perror ("Error opening file");

return(-1);

}

fseek(fp, 0, SEEK\_END);

len = ftell(fp);

fclose(fp);

printf("Total size of file.txt = %d bytes\n", len);

return(0);

}

· **Assuming we have a text file file.txt, which has the following content:**

**This is tutorialspoint.com**

**Output**

Total size of file.txt = 27 bytes

**ii. The rewind() Function**

· The rewind() function can be used in sequential or random access C file programming, and simply tells the file system to position the file pointer at the start of the file. Any error flags will also be cleared, and no value is returned.

**Description**

The C library function void rewind(FILE \*stream) sets the file position to the beginning of the file of the given stream.

**Declaration**

Following is the declaration for rewind() function.

void rewind(FILE \*stream)

**Parameters**

stream -- This is the pointer to a FILE object that identifies the stream.

**Return Value**

This function does not return any value.

**Example**

#include <stdio.h>

int main()

{

char str[] = "This is tutorialspoint.com";

FILE \*fp;

int ch;

/\* First let's write some content in the file \*/

fp = fopen( "file.txt" , "w" );

fwrite(str , 1 , sizeof(str) , fp );

fclose(fp);

printf("Total size of file.txt = %d bytes\n", len);

return(0);

}

· **Assuming we have a text file file.txt, which has the following content:**

**This is tutorialspoint.com**

**Output**

Total size of file.txt = 27 bytes

**ii. The rewind() Function**

· The rewind() function can be used in sequential or random access C file programming, and simply tells the file system to position the file pointer at the start of the file. Any error flags will also be cleared, and no value is returned.

**Description**

The C library function void rewind(FILE \*stream) sets the file position to the beginning of the file of the given stream.

**Declaration**

Following is the declaration for rewind() function.

void rewind(FILE \*stream)

**Parameters**

stream -- This is the pointer to a FILE object that identifies the stream.

**Return Value**

This function does not return any value.

**Example**

#include <stdio.h>

int main()

{

char str[] = "This is tutorialspoint.com";

FILE \*fp;

int ch;

/\* First let's write some content in the file \*/

fp = fopen( "file.txt" , "w" );

fwrite(str , 1 , sizeof(str) , fp );

fclose(fp);

fp = fopen( "file.txt" , "r" );

while(1)

{

ch = fgetc(fp);

if( feof(fp) )

{

break ;

}

printf("%c", ch);

}

rewind(fp);

printf("\n");

while(1)

{

ch = fgetc(fp);

if( feof(fp) )

{

break ;

}

printf("%c", ch);

}

fclose(fp);

return(0);

}

**Assuming we have a text file file.txt having the following content:**

This is tutorialspoint.com

**Output:**

This is tutorialspoint.com

This is tutorialspoint.com

**fseek(), ftell() and rewind()**

fseek() - It is used to moves the reading control to different positions using fseek function.

ftell() - It tells the byte location of current position in file pointer.

rewind() - It moves the control to beginning of a file.

Program

1

2

3

4

5

6

7

8

9

10

11

12

13

14

#include

void main(){

FILE \*fp;

int i;

clrscr();

fp = fopen("CHAR.txt","r");

for (i=1;i&lt;=10;i++){

printf("%c : %d\n",getc(fp),ftell(fp));

fseek(fp,ftell(fp),0);

if (i == 5)

rewind(fp);

}

fclose(fp);

}

Output

W : 0

e : 1

l : 2

c : 3

o : 4

W : 0

e : 1

l : 2

c : 3

o : 4

**Exercises**

1. Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.

Copy the file a block (512 bytes) at a time.

Check: that the program has two arguments

 or print "Program need two arguments"

 that the first name file is readable

 or print "Cannot open file .... for reading"

 that the second file is writable

 or print "Cannot open file .... for writing"

1. Write a program last that prints the last *n* lines of a text file, by *n* and the file name should be specified form command line input. By default *n* should be 5, but your program should allow an optional argument so that

 last -n file.txt

prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

1. Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.

**7. command line arguments**

· It is possible to pass some values from the command line to the C programs when they are executed.

 These values are called command line arguments and many times they are

important for the c program.

· 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.

· main() function of a C program accepts arguments from command line or from other shell scripts by following commands.

They are,

argc

argv[]

· where,

o argc - Number of arguments in the command line including program name

o argv[] – This is carrying all the arguments



