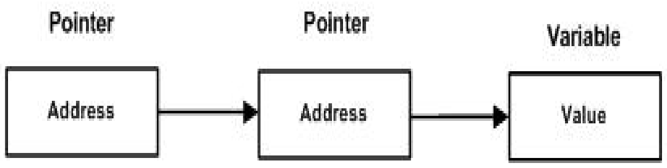
1. Define double pointer with example.

Double pointer or pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include<stdio.h>

void main ()
{
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   // take the address of var
   ptr = &var;

   // take the address of ptr using address of operator &
   pptr = &ptr;

   // accessing the value 3000
   printf("Value of var : %d\n",var);
   printf("Value available at *ptr :%d\n",*ptr);;
   printf("Value available at **pptr :%d\n",**pptr);
   return 0;
}
```

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

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

2. Write short note on Macros with suitable example. [Very Imp]

A macro is a name given to a block of C statements as a pre-processor directive. Being a pre-processor, the block of code is communicated to the compiler before entering into the actual coding (main () function). A macro is defined with the preprocessor directive, #define.

The advantage of using macro is 1) The speed of the execution of the program is the major advantage of using a macro 2) It saves a lot of time that is spent by the compiler for invoking / calling the functions. 3) It reduces the length of the program.

The disadvantage of the macro is the size of the program. The reason is, the pre-processor will replace all the macros in the program by its real definition prior to the compilation process of the program

Preprocessing directive #define has two forms. The first form is:

1) Simple Macros

```
#define identifier token_string
```

`token_string` part is optional but, are used almost every time in program.

Example of #define

```
#define c 299792458 /*speed of light in m/s */
```

The token string in above line 2299792458 is replaced in every occurance of symbolic constant *c*.

C Program to find area of a cricle. [Area of circle=$\pi r^{2]}$

```
#include <stdio.h>
#define PI 3.1415
int main(){
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d",&radius);
    area=PI*radius*radius;
    printf("Area=%f",area);
    return 0;
}
```

Output

```
Enter the radius: 3
Area=28.27
```

Second form of preprocessing directive with #define is:

2) Macros with argument

Preprocessing directive #define can be used to write macro definitions with parameters as well in the form below:

```
#define identifier(argument 1,.....argument n) token_string
```

Again, the token string is optional but, are used in almost every case. Let us consider an example of macro definition with argument.

```
#define area(r) (3.1415*(r)*(r))
```

Here, the argument passed is *r*. Every time the program encounters **`area(argument)`**, it will be replace by **`(3.1415*(argument)*(argument))`**. Suppose, we passed `(r1+5)` as argument then, it expands as below:

```
area(r1+5) expands to (3.1415*(r1+5)*(r1+5))
```

C Program to find area of a circle, passing arguments to macros. [Area of circle=πr^2]

```
#include <stdio.h>
#define PI 3.1415
#define area(r) (PI*(r)*(r))
int main(){
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d",&radius);
    area=area(radius);
    printf("Area=%.2f",area);
    return 0;
}
```

3. Explain the following: Preprocessor, Conditional Operator.

The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

Conditional Operator (Not conditional compilation)- ? :

The conditional operator in C is also known as ternary operator. It is called ternary operator because it takes three arguments. The conditional operator evaluates an expression returning a value if that expression is true and different one if the expression is evaluated as false.

Syntax:
condition ? result1 : result2;

If the condition is true, result1 is returned else result2 is returned.

Examples:
(10==5)? 11: 12; // returns 12, since 10 not equal to 5.
(10!=5)? 4 : 3; // returns 4, since 10 not equal to 5.
(12>8)? a : b; // returns the value of a, since 12 is greater than 8.

4. Write a program in C to copy content from one file to another file.

```
/* A program to copy the contents of a file to another file */
#include<stdio.h>
void main()
{
   FILE *fp1,*fp2;
   char c;
   fp1=fopen("test","r");
   fp2=fopen("testCopy","w");
   c=getc(fp1);
   while(c!=EOF)
   {
      fputc(c,fp2); //To display character on screen
      c=fgetc(fp1);
   }
   fclose(fp1);
   fclose(fp2);
}
```

5. Write the difference between call by value and call by reference with suitable example. [UPTU 2013-14], [UPTU 2012-13], [UPTU 2008-09] [VERY IMP]

Now we will take a look at call by value and call by reference (also known as pass-by-value and pass-by-reference). These methods are different ways of passing (or calling) data to functions.

Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function. Let’s take a look at a call by value example:

```
#include <stdio.h>

void call_by_value(int x) {
        printf("Inside call_by_value x = %d before adding 10.\n", x);
        x =x + 10;
        printf("Inside call_by_value x = %d after adding 10.\n", x);
}

void main() {
        int a=10;
        printf("a = %d before function call_by_value.\n", a);
        call_by_value(a);
        printf("a = %d after function call_by_value.\n", a);
}
```

The output of this call by value code example will look like this:

```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

Ok, let's take a look at what is happening in this call-by-value source code example. In the main() we create a integer that has the value of 10. We print some information at every stage, beginning by printing our variable a. Then function call_by_value is called and we input the variable a. This variable (a) is then copied to the function variable x. In the function we add 10 to x (and also call some print statements). Then when the next statement is called in main() the value of variable a is printed. We can see that the value of variable a isn't changed by the call of the function call_by_value().

Call by Reference

If data is passed by reference, address of the data is copied inside a pointer in the CALLED function instead of the actual variable as is done in a call by value. Because an address is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example:

```
#include <stdio.h>
void call_by_reference(int *y) {
        printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        *y= *y + 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
void main() {
        int b=10;
        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);
}
```

The output of this call by reference source code example will look like this:

```
b = 10 before function call_by_reference.
```

```
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.
```

Let’s explain what is happening in this source code example. We start with an integer b that has the value 10. The function call_by_reference() is called and the address of the variable b is passed to this function. Inside the function there is some before and after print statement done and there is 10 added to the value at the memory pointed by y. Therefore at the end of the function the value is 20. Then in main() we again print the variable b and as you can see the value is changed (as expected) to 20.

Swapping Program using Pass by Value

```
#include<stdio.h>
void main()
{
   int x,y;
   printf("Enter two values:\n");
   scanf("%d%d",&x,&y);
   swap(x,y);
   printf("Inside main():\n");
   printf("First value:%d Second value:%d\n",x,y);
}
void swap(int a, int b)
{
   int t;
   t=a;
   a=b;
   b=t;
   printf("Inside swap()\n");
   printf("First value:%d Second value:%d\n",a,b);
}
```

Output:

```
Enter two values:
4
5
Inside swap()
First value:5 Second value:4
Inside main():
First value:4 Second value:5
```

Swapping Program using Pass By Reference: [VERY IMP]

```
#include<stdio.h>
void swap(int *a, int *b);
void main()
{
   int x,y;
   printf("Enter the value of a and b\n");
   scanf("%d%d",&x,&y);
   swap(&x,&y); /* Pass by reference */
```

```
    printf("Inside main():\n");
    printf("First value:%d Second value:%d\n",x,y);
}
void swap(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
    printf("Inside swap()\n");
    printf("First value:%d Second value:%d\n",*a,*b);
}
```

Output:

```
Enter the value of a and b
4
5
Inside swap()
First value:5 Second value:4
Inside main():
First value:5 Second value:4
```

6. What is pointer arithmetic? Write advantage and disadvantage of using pointer variable. [UPTU 2012-13]

Pointer arithmetic involves following four operations:- adding an integer to a pointer, subtracting an integer from pointer, subtracting 2 pointers, compairing 2 pointers.
We can see address of an variable after performing arithmetic operations.

Expression	Result
Address + Number	Address
Address – Number	Address
Address – Address	Number
Address + Address	Illegal

Adding integer value with Pointer

In C Programming we can add any integer number to Pointer variable. It is perfectly legal in c programming to add integer to pointer variable.
In order to compute the final value we need to use following formulae :

```
final value = (address) + (number * size of data type)
```

Consider the following example –

```
int *ptr , n;
ptr = &n ;
ptr = ptr + 3;
```

Example 1 : Increment Integer Pointer

```
#include<stdio.h>

void main(){
```

```
int *ptr=(int *)1000; /* ptr will contain address 1000*/

ptr=ptr+3;
printf("New Value of ptr : %u",ptr);
}

Output :

New Value of ptr : 1006
```

Subtracting integer value with Pointer
Suppose we have subtracted “n” from pointer of any data type having initial addess as “init_address” then after subtraction we can write –

```
ptr = initial_address - n * (sizeof(data_type))
```

Consider the following example –

```
int *ptr , n;
ptr = &n ;
ptr = ptr - 3;
```

Example 1 : Decrement Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000; /* ptr will contain address 1000*/

ptr=ptr-3;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

```
New Value of ptr : 994
```

Subtracting two Pointer in C Programming Language :

1. Differencing Means **Subtracting two Pointers**.
2. Subtraction gives the Total number of objects between them .
3. Subtraction indicates “How apart the two Pointers are ?”

C Program to Compute Difference Between Pointers :

```
#include<stdio.h>

int main()
{
int num , *ptr1 ,*ptr2 ;

ptr1 = &num ;
ptr2 = ptr1 + 2 ;

printf("%d",ptr2 - ptr1);
return(0);
}
```

Output :

```
2
```

- ptr1 stores the **address of Variable** num
- Value of ptr2 is incremented by **4 bytes**
- Differencing two Pointers

Comparison between two Pointers :

1. **Pointer comparison is Valid** only if the **two pointers are Pointing to same array**
2. All Relational Operators can be used for comparing pointers of **same type**
3. **All Equality and Inequality Operators** can be used with all Pointer types
4. Pointers **cannot be Divided or Multiplied**

Example Pointer Comparison

```
#include<stdio.h>

int main()
{
int *ptr1,*ptr2;

ptr1 = (int *)1000;  /* Address 1000 is stored in ptr1 */
ptr2 = (int *)2000;   /* Address 2000 is stored in ptr2 */

if(ptr2 > ptr1)
   printf("Ptr2 is far from ptr1");

return(0);
}
```

Following operators can be used for compairing two pointers :

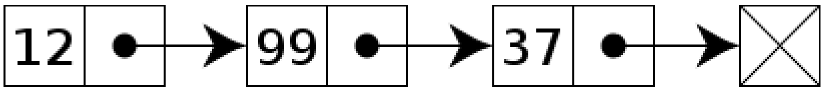
>	Greater Than
<	Less Than
>=	Greater Than And Equal To
<=	Less Than And Equal To
==	Equals
!=	Not Equal

7. Write short notes on: Macros, Linked List, and Mathematical Function. [UPTU 2012-13]

Macros- Already Explained [VERY IMP]

<u>**Linked list**</u>

In computer science, a linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require sequential scanning of most or all of the list elements. The advantages and disadvantages of using linked lists are as follows:-

Advantages:

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Linear data structures such as stacks and queues are easily executed with a linked list.
- They can reduce access time and may expand in real time without memory overhead.

Disadvantages:

- They have a tendency to waste memory due to pointers requiring extra storage space.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. Singly linked lists are extremely difficult to navigate backwards, and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

Mathematical Functions:

There are many standard library functions defined under "math.h" to perform mathematical operations in C programming.

Library functions under "math.h"

Function	**Work of function**
acos	Computes arc cosine of the argument
acosh	Computes hyperbolic arc cosine of the argument
asin	Computes arc sine of the argument
asinh	Computes hyperbolic arc sine of the argument
atan	Computes arc tangent of the argument
atanh	Computes hyperbolic arc tangent of the argument
atan2	Computes arc tangent and determine the quadrant using sign
cbrt	Computes cube root of the argument
ceil	Returns nearest integer greater than argument passed
cos	Computes the cosine of the argument

Function	Work of function
cosh	Computes the hyperbolic cosine of the argument
exp	Computes the e raised to given power
fabs	Computes absolute argument of floating point argument
floor	Returns nearest integer lower than the argument passed.
hypot	Computes square root of sum of two arguments (Computes hypotenuse)
log	Computes natural logarithm
log10	Computes logarithm of base argument 10
pow	Computes the number raised to given power
sin	Computes sine of the argument
sinh	Computes hyperbolic sine of the argument
sqrt	Computes square root of the argument
tan	Computes tangent of the argument
tanh	Computes hyperbolic tangent of the argument

8. Dynamic memory allocation. [UPTU 2011-12]

Dynamic memory allocation is necessary to manage available memory. For example, during compile time, we may not know the exact memory needs to run the program. So for the most part, memory allocation decisions are made during the run time. C also does not have automatic garbage collection like Java does. Therefore a C programmer must manage all dynamic memory used during the program execution. The <stdlib.h> provides four functions that can be used to manage dynamic memory.

NAME

calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

calloc() allocates memory for an array of nmemb element of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

9. Macros. How they are different from C variables. Advantage of macro. Explain conditional compilation and how does it help the programmers. [UPTU 2011-12]

Macros- Already Explained

Difference between Macros Vs Variables:

- Macros are handled by preprocessor while variables are handled by compiler
- The name and value of a macro is fixed while a variable has only its name fixed while its value can vary.

Advantages of macros- Already explained

Conditional Compilation : [VERY IMP]

These Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

- `#if`
- `#ifdef`
- `#ifndef`

Optionally, an alternative block of text can be set aside with one of two directives:

- `#else`
- `#elif`

The end of the block or alternative block is marked by the `#endif` directive.

If the condition checked by `#if` , `#ifdef` , or `#ifndef` is true (nonzero), then all lines between the matching `#else` (or `#elif`) and an `#endif` directive, if present, are ignored.

If the condition is false (0), then the lines between the `#if` , `#ifdef` , or `#ifndef` and an `#else` , `#elif` , or `#endif` directive are ignored.

Syntax:

```
#ifdef MACRONAME
    Statement_block;
#endif
```

1. If the MACRONAME specified after #ifdef is defined previously in #define then statement_block is followed otherwise it is skipped

2. We say that the conditional succeeds if ***MACRO*** is defined, fails if it is not.

Example 1 :

```
#include<stdio.h>
#define NUM 10

void main()
{
// Define another macro if MACRO NUM is defined

#ifdef NUM
      #define MAX 20
#endif

printf("MAX number is : %d",MAX);
}
```

Output :

```
MAX Number is 20
```

10. Pointers. Declare, initialize. Swap two numbers using pointers. [UPTU 2011-12]

Pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %p\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Pointer Initialization:

Pointer Initialization is the process of assigning address of a variable to pointer variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;          //pointer declaration
ptr = &a ;          //pointer initialization
or,
int *ptr = &a ;        //initialization and declaration together
```

Swapping Program using Pass By Reference (Pointers): [VERY IMP]

```
#include<stdio.h>
void swap(int *a, int *b);
void main()
{
   int x,y;
   printf("Enter the value of a and b\n");
   scanf("%d%d",&x,&y);
   swap(&x,&y); /* Pass by reference */
   printf("Inside main():\n");
   printf("First value:%d Second value:%d\n",x,y);
}
void swap(int *a, int *b)
{
   int t;
   t=*a;
   *a=*b;
   *b=t;
   printf("Inside swap()\n");
   printf("First value:%d Second value:%d\n",*a,*b);
}
```

Output:

```
Enter the value of a and b
```

```
4
5
Inside swap()
First value:5 Second value:4
Inside main():
First value:5 Second value:4
```

11. A File contains some numbers. WAP create two more files ODD and EVEN. Read the numbers from the original file and store the Odd numbers in ODD file and even numbers in EVEN file. [UPTU 2011-12]

```
#include<stdio.h>
void main()
{
   FILE *fp,*fp1,*fp2;
   int n,c;
   fp=fopen("testNum","r");
   fp1=fopen("even.dat","w");
   fp2=fopen("odd.dat","w");
   while(feof(fp)==0)
   {
   c=fscanf(fp,"%d",&n);
   if(n%2==0)
      fprintf(fp1,"%d ",n);
   else
      fprintf(fp2,"%d ",n);
   }
   fclose(fp);
   fclose(fp1);
   fclose(fp2);
}
```

12. What is dynamic memory allocation? Explain malloc() with example. [UPTU 2008-09]

Already Explained

13. Write a program in C that takes ten integers from a file and write square of these integers into another file

```
#include<stdio.h>
void main()
{
   FILE *fp,*fp1;
   int n,sq;
   fp=fopen("testNum","r");
   fp1=fopen("testNumSquare","w");
   while(feof(fp)==0)
   {
   fscanf(fp,"%d",&n);
   sq=n*n;
   fprintf(fp1,"%d ",sq);
```

```
    }
    fclose(fp);
    fclose(fp1);
}
```

14. How macros are defined and called in C? Explain with example. [UPTU 2008-09]
Already Explained

15. What do you understand by pointer arithmetic? Explain. [UPTU 2007-08]
Already explained

ANOTHER VERY IMPORTANT QUESTION:

Explain various file handling functions?

Function	**Syntax**	**Example**

fopen - Open file	FILE * fopen(char * filename, char * mode)
	Filename – string filename with path
	Mode – mode of opening
	r read only
	w write only
	a append file
	+ read and write (r+/w+/a+)
	
	Return value – **FILE pointer** on success
	NULL on failure
		FILE * infile ;
		Infile = fopen("a.txt","r");
		
		FILE *outfile = fopen("b.txt","w");
		if(infile==NULL || outfile==NULL)
		{
		printf("error");
		}

fclose – close the open file	fclose(FILE *fptr) fptr – FILE * of the open file Returns void	fclose(infile);

fgetc – read one char from file	int fgetc(FILE *fptr)
	
	fptr – FILE * of the open file
	returns – character read
	-1 on eof or error
		while((ch=fgetc())!=-1)
		putchar(ch);
		//display content of file
		//on screen

fputc – write one character to the file	int fputc(int ch,FILE *fptr)
	ch – character to write
	fptr – open file pointer
	returns character written
	EOF if there is error
		for(ch='A';ch<='Z';ch++)
		fputc(ch,fptr);

fscanf – Formatted read from file	int fscanf(FILE *ptr,char * fmt, ...) ptr – FILE pointer fmt – format specifier for	fscanf(infile, "%d %d",&n1,&n2);

	variables to be read Returns – number of values read successfully	
fprintf – formatted write to file	int fprintf(FILE *ptr,char * fmt, ...) ptr – FILE pointer fmt – format specifier for variables to be read Returns – number of characters printed or negative number on error	fprintf(outfile,"Value is %d\n",n1);

fread – read from binary file	int fread(void *buffer, size_t size, size_t num, FILE *ptr)
	
	buffer – stores the value read
	size – size of the buffer
	num – number of blocks to be read
	ptr – file pointer
	Returns – number of values read
		Char *str = malloc(50);
		fread(str,50,1,infile);

fwrite – write to a binary file	int fwrite(void *buffer, size_t size, size_t num, FILE *ptr)
	
	buffer – stores the value read
	size – size of the buffer
	num – number of blocks to be read
	ptr – file pointer
	Returns – number of values written
		for(i=0;i<10;i++)
		   fwrite(&i,sizeof(i),1,outfile);
		char a[]="end";
		fwrite(a,strlen(a)+1,1,outfile);

fseek – move the file pointer by given offset	int fseek(FILE*ptr,long offset,int whence)
	
	ptr – file pointer
	offset – offset in bytes from third parameter
	whence – SEEK_SET – from beginning of file
	SEEK_CUR – from current position
	SEEK_END – from end of file
	Returns – zero on success
	Non-zero on failure
		if(fseek(infile,4L,SEEK_SET)==0)
		{
		     char ch=fgetc(infile);
		     printf("The fourth char of the file is %c\n",ch);
		}

ftell – get the position of file pointer	int ftell(FILE *ptr)
	
	ptr – FILE pointer
	Returns – position of file pointer
	-1 on error
		FILE *ptr = fopen(b[1],"r");
		fseek(ptr,0L,SEEK_END);
		int size_of_file = ftell(ptr);

rewind – moves the file pointer to the beginning of the file	void rewind(FILE *ptr)
	
	ptr – FILE pointer
		rewind(infile);
		int n = fscanf(infile,"%d",&n);

Other important file functions:

feof()	- finds end of file. It returns 0 untill end of file has not been reached. As soon as end of file is reached it returns a non zero value.
remove()	- deletes a file
getw()	- reads a word from file
putw()	- writes a word to file