Let us swim in C

Dr. O. S. ABDUL QADIR

M. IBRAMSHA

SEMESTER – I

CORE – I: PROGRAMMING IN C

Subject Code: 20UCA1CC1		**Max. Marks:**	**100**
Hours:	**5**	**Internal Marks:**	**25**
Credits:	**5**	**External Marks:**	**75**

Objective To learn the syntax of all the statements and to provide programming skills in C

UNIT I **15 Hours**

Getting Started with C - C Instructions– Decision Control Structure: The if Statement – The if-else Statement - Use of Logical Operators - # **The Conditional Operators** #.

UNIT II **15 Hours**

The Loop Control Structure: The 'while' Loop –The 'for' Loop – The break Statement – The continue Statement – The 'do-while' Loop – The 'odd' loop.

Case Control Structure: Decisions using switch – switch versus if-else Ladder - # **The goto keyword** #.

UNIT III **15 Hours**

Functions and Pointers: Passing values between Functions – Scope Rule of Functions – Calling Convention – Using Library Functions – Advanced Features of Functions – # **Adding Functions to the Library** #.

The C Preprocessor: Features of C Preprocessor – Macro Expansion – File Inclusion – Conditional Compilation – #if and #elif Directives **–** # **Miscellaneous Directives** # – The Build Process.

UNIT IV **15 Hours**

Arrays – More on Arrays – Pointers and Arrays – Two dimensional Arrays – Array of Pointers – # **Three-Dimensional Array** #

Strings: More about Strings – Pointers and Strings – Standard Library String Functions – Two-Dimensional Array of Characters – Array of Pointers to Strings – Limitation of Array of Pointers to Strings.

UNIT V **15 Hours**

Structures: Array of Structures – Additional Features of Structures – Uses of Structures. Console Input / Output – Types of I/O – Console I/O Functions. File Input / Output: Data Organization – File Operations – Counting Characters, Tabs, Spaces – A File-Copy Program – File Opening Modes – # **String (Line) I/O in Files** # - Record I/O in Files.

 # .... # Self-study portion

Text Book

YashavantKanetkar, Let Us C, BPB Publications, New Delhi, 13thEdition, 2013.

Reference Book:

1. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill Education Private Ltd., Fifth Edition, 2011.
2. D. Ravichandran, Programming in C, New Age International (P) Ltd., First Edition, 1996.

Web Reference:

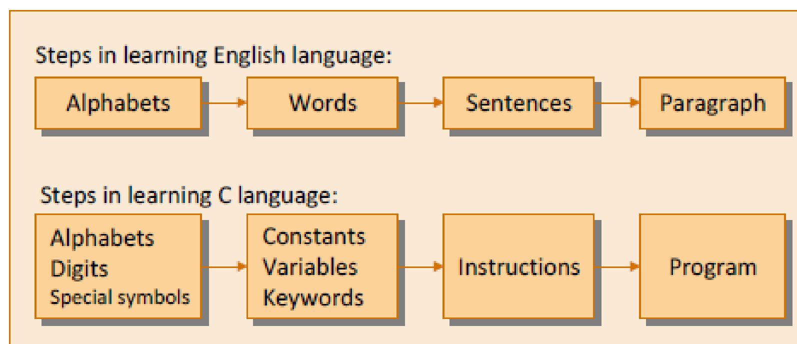
https://www.programiz.com/c-programming

What is C?

- C is a programming language developed by computer scientist named **Dennis Ritchie** at **AT&T** (American Telephone & Telegraph) Bell Laboratories of USA in **1972**.
- Dennis MacAlistair Ritchie or Dennis M. Ritchie, (1941 – 2011), American computer scientist and co-winner of the 1983 A.M. Turing Award, the highest honour in computer science.
- C was created from ALGOL, BCL, and B programming language and it contains all the features of these languages and many more additional concepts that make it unique from other programming languages.

Getting Started with C

- The C Character Set
- Constants, Variables and Keywords
- Types of C Constants
    - Rules for Constructing Integer Constants
    - Rules for Constructing Real Constants
    - Rules for Constructing Character Constants
- Types of C Variables
    - Rules for Constructing Variable Names
- C Keywords



The C Character Set

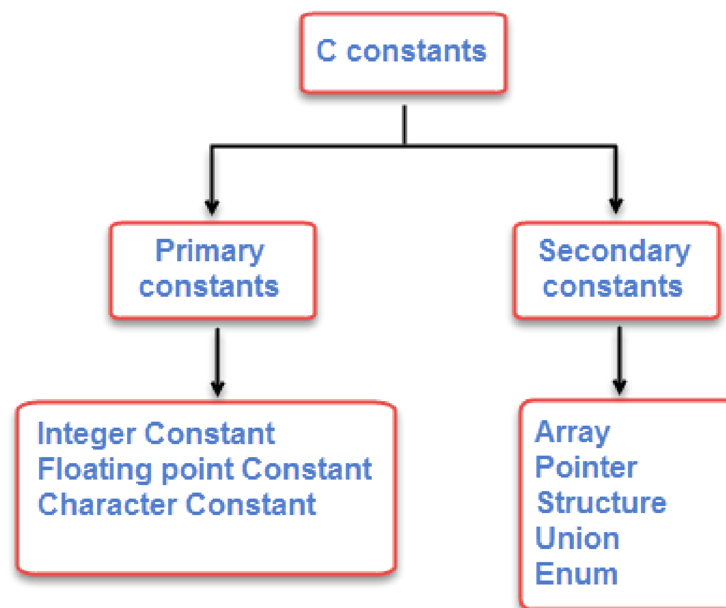
- A character denotes any alphabet, digit or special symbol used to represent information.
- The valid alphabets, numbers and special symbols allowed in C.

<table>
<tr><td>Alphabets</td><td>A, B, ….., Y, Z
a, b, ….., y, z</td></tr>
<tr><td>Digits</td><td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td></tr>
<tr><td>Special symbols</td><td>~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { }
[ ] : ; " ' < > , . ? / $</td></tr>
</table>

Constants

- Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.
- Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.

Types of C Constants



Rules for constructing ***integer*** constants

- © An integer constant must have at least one digit.
- © It must not have a decimal point.
- © It can be either positive or negative.
- © If no sign precedes an integer constant, it is assumed to be positive.
- © No commas or blanks are allowed within an integer constant.

  Ex.: 426
  +782
  -8000
  -7605

Rules for Constructing ***Real*** Constants

- © Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.
- © Following rules must be observed while constructing real constants expressed in fractional form:
  - © A real constant must have at least one digit.
  - © It must have a decimal point.
  - © It could be either positive or negative.
  - © Default sign is positive.
  - © No commas or blanks are allowed within a real constant.

    Ex.: +325.34
    426.0
    -32.76
    -48.5792
- © In exponential form the real constant is represented in two parts. The part appearing before 'e' is called mantissa, whereas the part following 'e' is called exponent.
- © Thus 0.000342 can be written in exponential form as 3.42e-4 (which in normal arithmetic means 3.42 x 10-4).

Rules for Constructing ***Character*** Constants

- © A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
- © Both the inverted commas should point to the left. For example, 'A' is a valid character constant whereas 'A' is not.

  Ex.: 'A'
  'I'
  '5'
  '='

Variables

- © A variable is a name of the memory location.
- © It is used to store data.
- © Its value can be changed, and it can be reused many times.
- © There are three places where variables can be declared in C programming language −
  1. Inside a function or a block which is called local variables.
  2. Outside of all functions which is called global variables.
  3. In the definition of function parameters which are called formal parameters.

  Example:

```
#include <stdio.h>
int g = 20;  /* global variable declaration */
int main ()
{
        int g = 10;  /* local variable declaration */
        printf ("value of g = %d\n",  g);
}
```

Types of C Variables

- © A particular type of variable can hold only the same type of constant.
- © An integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

Rules for Constructing ***Variable Names***

- © A variable name is any combination of 1 to 31 alphabets, digits or underscores. Variable names length up to 247 characters. The rule of 31 characters. Do not create unnecessarily long variable name.
- © The first character in the variable name must be an alphabet or underscore (_).
- © No commas or blanks are allowed within a variable name.
- © No special symbol other than an underscore (as in gross_sal) can be used in a variable name.

  Ex.: si_int
  m_hra pop_e_89 aaa, bBBb

  a ssss // error
  a.b.c //error 1a //error

  Syntax: Datatype variablename;
  Datatype = int, float, char and double

Ex.:

```
int si, m_hra ;
float bassal;
char code;
```

C Keywords

- © Keywords are generally called as pre-defined or reserved words.
- © Every keyword in C language performs a specific function in a program.
- © Keywords cannot be used as variable names.
- © Keywords have fixed meanings, and that meaning cannot be changed.
- © They are the building block of a 'C' program.
- © Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). There are only 32 keywords available in C.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C Instructions

- © C instructions are the commands in the program that instructs the compiler to do certain action. Basically it gives the instruction to the compiler on how to achieve the goal of the program.
- © There are three types of instructions in C.

1. Type Declaration Instructions

- © These instructions inform the compiler about the type of variables used.
- © That means, whenever a variable is used in the program, we have to specify what types of data it can hold – like integer, float, double, character etc.
- © This makes the compiler to store only those specific types of values in it.

Example:

```
void main()
{
    int intVar1, intVar2, intSum;
        float flAvg;
        char chrArr [10];
  ....
}
int subtractNum(int var1, int var2)
{
        int intResult;
        ....
}
```

2. *Arithmetic Instructions*

- These instructions are used to perform arithmetic calculation within the program.
- It uses arithmetic operators like +, -, *, /, %, =, ++,–, +=,-+ etc.
- The variables that participate in the arithmetic operations are termed as operands.
  Example:

  sum = var1+var2+var3;
  result = var1+var2/var3;
  result = (var1+var2)/var3;
  *result = (var1*100) + var2 - var3;*
  *result = a+5*30+sum/ 20*
- These operators have their own precedence while evaluating the instructions.
- It first evaluates any instructions within parenthesis, (), then multiplication and division, then addition and subtraction and finally assigns the value to the resultant variable.

  ( ) → * / → + – → =
- If the instruction contains any logical operation, then it evaluates logical NOT (!) first.
- Then it proceeds to evaluate multiplication/division/modulus (%), then addition/subtraction, then relational operators (==, !=), then logical AND, then logical OR and finally assigns the value.

  NOT → * / % → + – → == != → AND → OR → =

3. *Control Instructions*

- These instructions are used to control the flow of the program execution.
- They maintain certain order in which program needs to be executed.
- These order of execution may be based on certain conditions or may be based on certain values – may be input values or some result values.
- There are four types of control instructions in C.
  - Sequence Control Instructions
    - The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program.
  - Decision / Selection Control Instructions
  - Case Control Instructions
    - Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next.
  - Loop Control Instructions
    - The Loop control instruction helps computer to execute a group of statements repeatedly.

Integer and Float Conversions

- An arithmetic operation between an integer and integer always integer result.
- An operation between a real and real always real result.
- An operation between an integer and real always a real result.

Operation	Result
5 / 2	2
5.0 / 2	2.5
5 / 2.0	2.5
5.0 / 2.0	2.5

Decision Control Structures

A decision control instruction can be implemented in C using:

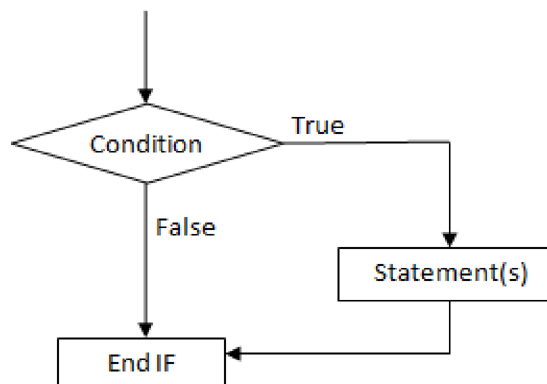
a) The if statement
b) The if-else statement
c) The conditional operators

The if Statement

C uses the keyword if to implement the decision control instruction.
The general form of if statement looks like this:

```
if (condition)
{
    statements;
    ... ... ...
}
```

Flow Chart



© The keyword if tells the compiler that what follows is a decision control instruction.
© The condition following the keyword if is always enclosed within a pair of parentheses.
© If the condition is true, then the statement is executed. Otherwise, the program skips.

this expression	is true if
x == y	x is equal to y
x != y	x is not equal to y
x < y	x is less than y
x > y	x is greater than y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y

Example:

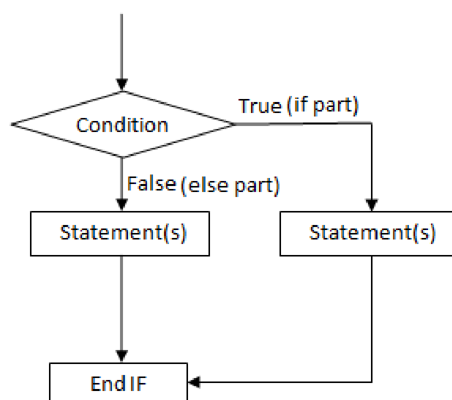
```
#include<stdio.h>
int main()
{
        int number=0;
        printf("enter a number:");
        scanf("%d",&number);
        if(number==10)
                printf("number is equals to 10");
}
```

The if-else Statement

- The ***'if'*** statement by itself will execute a single statement, or a group of statements.
- When the expression following if evaluates to true, then the block of if statements are executed. Otherwise, the condition evaluates to false, the else block will be executed.

```
if (condition)
{
   statements;
   … … …
}
else
{
   statements;
   … … …
}
```

Flow Chart



Use of Logical Operators

- C allows usage of three logical operators, namely, **&& (AND)**, || **(OR)** and **! (NOT)**.
- Don't use the single symbol | and &. These single symbols also have a meaning. The first two operators, && and ||, allow two or more conditions to be combined in an if statement.
- The first two operators, && and ||, allow two or more conditions to be combined in an if statement.

  Consider the following example:

  The marks obtained by a student in 5 different subjects are input through the keyboard.

The student gets a division as per the following rules:
Percentage above or equal to 60 - First division
Percentage between 50 and 59 - Second division
Percentage between 40 and 49 - Third division
Percentage less than 40 - Fail

Write a program to calculate the division obtained by the student.
There are two ways in which we can write a program for this example.
These methods are given below.

```
/* Method – I */
# include <stdio.h>
int main( )
{
    int m1, m2, m3, m4, m5, per ;
    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) * 100 / 500 ;
    if ( per >= 60 )
        printf ( "First division\n" ) ;
    else
    {
        if ( per >= 50 )
            printf ( "Second division\n" ) ;
        else
        {
            if ( per >= 40 )
                printf ( "Third division\n" ) ;
            else
                printf ( "Fail\n" ) ;
        }
    }
    return 0 ;
}
```

- This is a straight-forward program. Observe that the program uses nested if-else.
- Though the program works fine, it has ***three disadvantages***:
    - As the number of conditions go on increasing the level of indentation also goes on increasing. As a result, the whole program creeps to the right. So much so that entire program is not visible on the screen. So if something goes wrong with the program locating what is wrong where becomes difficult.
    - Care needs to be exercised to match the corresponding ifs and else.
    - Care needs to be exercised to match the corresponding pair of braces.
- All these three problems can be eliminated by usage of ‘Logical Operators’.
  The following program illustrates this:

/* Method – II */

```
# include <stdio.h>
void main( )
{
    int m1, m2, m3, m4, m5, per ;
    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 500 * 100 ;
    if ( per >= 60 )
        printf ( "First division\n" ) ;
    if ( ( per >= 50 ) && ( per < 60 ) )
        printf ( "Second division\n" ) ;
    if ( ( per >= 40 ) && ( per < 50 ) )
        printf ( "Third division\n" ) ;
    if ( per < 40 )
        printf ( "Fail\n" ) ;
}
```

- © As can be seen from the second if statement, the && operator is used to combine two conditions. ‘Second division’ gets printed if both the conditions evaluate to true.
- © If one of the conditions evaluate to false then the whole thing is treated as false.

The else if Clause

There is one more way in which we can rewrite the above program
This involves usage of else if blocks as shown below.
/* else if ladder demo */

```
# include <stdio.h>
int main( )
{
    int m1, m2, m3, m4, m5, per ;
    per = ( m1+ m2 + m3 + m4+ m5 ) / 500 * 100 ;
    if ( per >= 60 )
        printf ( "First division\n" ) ;
    else if ( per >= 50 )
        printf ( "Second division\n" ) ;
    else if ( per >= 40 )
        printf ( "Third division\n" ) ;
    else
        printf ( "fail\n" ) ;
    return 0 ;
}
```

You can note that this program reduces the indentation of the statements. In this case, every else is associated with its previous if. The last else goes to work only if all the conditions fail. Also, if a condition is satisfied, other conditions below it are not checked. Even in else if ladder, the last else is optional.

Note: that the else if clause is nothing different. It is just a way of rearranging the else with the ***‘if’*** that follows it.

The Conditional Operators

- © Conditional operators can be used as an alternative to if-else statement if there is a single statement in the 'if block' and a single statement in the 'else block'.
- © The conditional operators ? and : are sometimes called ternary operators since they take three arguments. In fact, they form a kind of foreshortened if-then-else.

Their general form is,

expression1 ? expression2 : expression 3

- "if expression 1 is true (that is, if its value is non-zero), then the value returned will be expression 2, otherwise the value returned will be expression 3".

  Example

```
int x, y ;
scanf ( "%d", &x ) ;
y = ( x > 5 ? 3 : 4 ) ;
```

- This statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

  The equivalent if-else form would be,

```
if ( x > 5 )
        y = 3;
else
        y = 4;
```

- © The following points may be noted about the conditional operators:
- © It's not necessary that the conditional operators should be used only in arithmetic statements. This is illustrated in the following examples:

Ex.:
```
int i ;
scanf ( "%d", &i ) ;
( i == 1 ? printf ( "Amit" ) : printf ( "All and sundry" ) ) ;
```

Ex.:
```
char a = 'z' ;
printf ( "%c", ( a >= 'a' ? a : '!' ) ) ;
```

The conditional operators can be nested as shown below.

```
int big, a, b, c ;
big = ( a > b ? ( a > c ? 3: 4 ) : ( b > c ? 6: 8 ) ) ;
```

Try Yourself

1. Write a C program to find whether a given year is a leap year or not.
2. Write a C program to calculate the root of a Quadratic Equation.
3. Write a C program to read temperature in centigrade and display a suitable message according to temperature state below : Go to the editor
   a. Temp < 0 then Freezing weather
   b. Temp 0-10 then Very Cold weather
   c. Temp 10-20 then Cold weather
   d. Temp 20-30 then Normal in Temp
   e. Temp 30-40 then Its Hot
   f. Temp >=40 then Its Very Hot

4. Write a C program to check whether a character is an alphabet, digit or special character.
5. Write a C program to check whether an alphabet is a vowel or consonant.
6. Write a program in C to calculate and print the Electricity bill of a given customer. The customer id., name and unit consumed by the user should be taken from the keyboard and display the total amount to pay to the customer.

   The charge are as follow

Unit	Charge/unit
upto 199	@1.20
200 and above but less than 400	@1.50
400 and above but less than 600	@1.80
600 and above	@2.00

   If bill exceeds Rs. 400 then a surcharge of 15% will be charged and the minimum bill should be of Rs. 100/-

7. Write a program in C to read any day number in integer and display day name in the word.

8. Write a program in C which is a Menu-Driven Program to compute the area of the various geometrical shape.

THE LOOP CONTROL STRUCTURE

- © Control structures alter the normal sequential flow of a statement execution.
- © A loop statement allows us to execute a statement or group of statements multiple times.
- © C supports the following loop statements
  1. Using a ***while*** statement
  2. Using a ***for*** statement
  3. Using a ***do-while*** statement

The while loop

- © Repeats a statement or group of statements while a given condition is true.
- © It tests the condition before executing the loop body.

Syntax:

```
initialize loop counter ;
while ( condition )
{
        statement(s);
        increment loop counter ;
}
```

- Here, statement(s) may be a single statement or a block of statements.
- The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.
- The condition being tested may use relational or logical operators

```
while ( i <= 10 )
while ( i >= 10 && j <= 15 )
while ( j > 10 && ( b < 15 || c < 20 ) )
```

Example:

```
# include <stdio.h>
int main( )
{
        int i = 1 ;
        while ( i <= 10 )
        {
                printf ( "%d\n", i ) ;
                i = i + 1 ;
        }
        return 0 ;
}
```

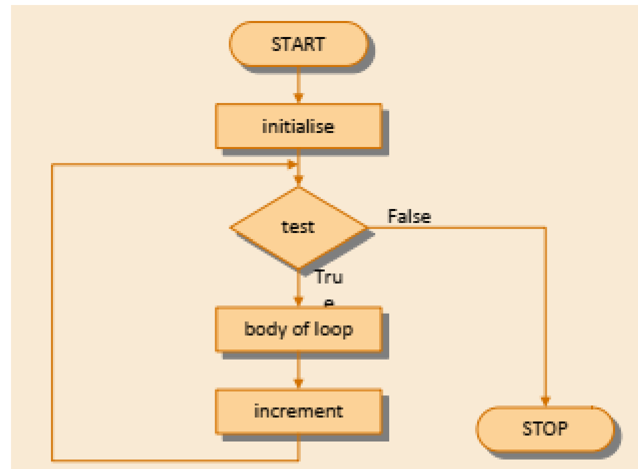
- The statements within the loop may be a single line or a block of statements. In the first case, the braces are optional. For example,

```
while ( i <= 10 )
        i = i + 1 ;
```

- is same as

```
while ( i <= 10 )
{
        i = i + 1 ;
}
```

Flow Chart



Note:

- © The statements within the while loop would keep getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the while loop.
- © In place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.

The for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax

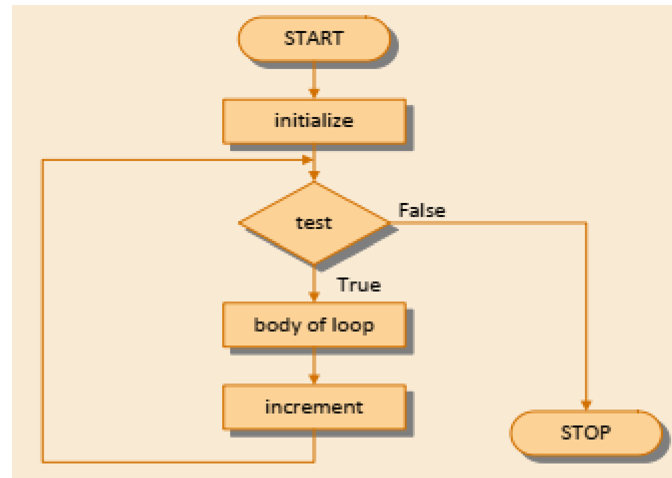
```
for ( init; condition; increment )
{
        statement(s);
}
```

- The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the increment statement.

Example

```
# include <stdio.h>
int main( )
{
        int i = 1 ;
        for ( i=1; i <= 10; i++ )
        {
                printf ( "%d\n", i ) ;
        }
}
```

Flow Chart



The break statement

- Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- This statement has the following two uses
    - When break is encountered inside any loop, control automatically passes to the first statement after the loop.
    - It can be used to terminate a case in the switch statement.
- If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.
- A break is usually associated with an if statement.

  Example:

```
# include <stdio.h>
int main( )
{
        int num, i ;
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        i = 2 ;
        while ( i <= num - 1 )
        {
                if ( num % i == 0 )
                {
                        printf ( "Not a prime number\n" ) ;
                        break ;
                }
                i++ ;
        }
        if ( i == num )
                printf ( "Prime number\n" ) ;
}
```

- In this program, the moment num % i turns out to be zero, (i.e., num is exactly divisible by i), the message “Not a prime number” is printed and the control breaks out of the while loop.

The continue Statement

- © This statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- © When continue is encountered inside any loop, control automatically passes to the beginning of the loop.
- © A continue is usually associated with an if statement

Example:

```
int main( )
{
    int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
            continue ;
            printf ( "%d %d\n", i, j ) ;
        }
    }
    return 0 ;
}
```

The output of the above program would be...

```
1 2
2 1
```

Note

- when the value of i equals that of j, the continue statement takes the control to the for loop (inner) bypassing the rest of the statements pending execution in the for loop (inner).

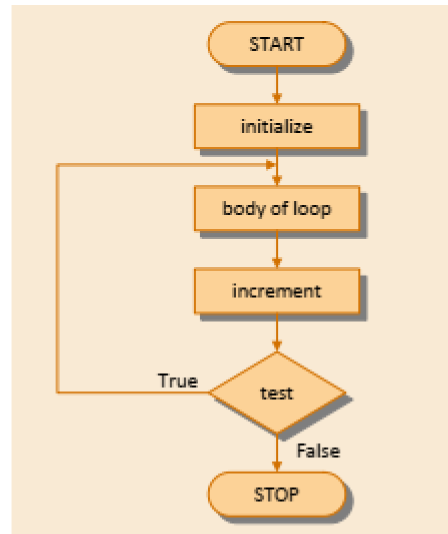
The do-while loop

- © There is a minor difference between the working of while and do-while loops.
- © This difference is the place where the condition is tested.
- © Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming checks its condition at the bottom of the loop.
- © This means that do-while would execute its statements at least once, even if the condition fails for the first time.
- © The while, on the other hand will not execute its statements if the condition fails for the first time.

Syntax

```
do
{
    statement(s);
} while( condition );
```

Flow Chart



Example:

```
# include <stdio.h>
int main( )
{
        int i = 1 ;
        do
        {
                printf ( "%d\n", i ) ;
                i = i + 1 ;
        } while ( i <= 10 );
        return 0 ;
}
```

The Odd loops

Sometimes a user may not know about how many times a loop is to be executed. If we want to execute a loop for unknown number of times, then the concept of odd loops should be implemented. This can be done using for-loop, while-loop or do-while-loops.

Example

```
# include <stdio.h>
int main( )
{
        char another = 'y' ;    int num ;
        for ( ; another == 'y' ; )
        {
                printf ( "Enter a number " ) ;
                scanf ( "%d", &num ) ;
                printf ( "square of %d is %d\n", num, num * num ) ;
                printf ( "Want to enter another number y/n " ) ;
                fflush ( stdin ) ;
                scanf ( "%c", &another ) ;
        }
}
```

```
/* odd loop using a while loop */
# include <stdio.h>
void main( )
{
	char another = 'y' ;
	int num ;
	while ( another == 'y' )
	{
		printf ( "Enter a number " ) ;
		scanf ( "%d", &num ) ;
		printf ( "square of %d is %d\n", num, num * num ) ;
		printf ( "Want to enter another number y/n " ) ;
		fflush ( stdin ) ;
		scanf ( " %c", &another ) ;
	}
}
```

- break and continue are used with do-while just as they would be in a while or a for loop.
- A break takes you out of the do-while bypassing the conditional test.
- A continue sends you straight to the test at the end of the loop.

CASE CONTROL STRUCTURE

Decision using switch

© The control statement that allows us to make a decision from the number of choices is called a ***switch***, or more correctly a switch-case default, since these three keywords go together to make up the control statement.

© They most often appear as follows:

Syntax:

```
switch ( integer expression/ character )
{
	case constant 1:
			do this ;
	case constant 2:
			do this ;
	………………
	default:
			do this ;
}
```

- The integer expression following the keyword switch is any C expression that will yield an integer value.
- The keyword case is followed by an integer or a character constant. Each constant in each case must be different from all the others.
- The “do this” lines in the above form of switch represent any valid C statement.
- The break statement when used in a switch takes the control outside the switch.
- If we have no default case, then the program simply falls through the entire switch and continues with the next instruction that follows the closing brace of switch.

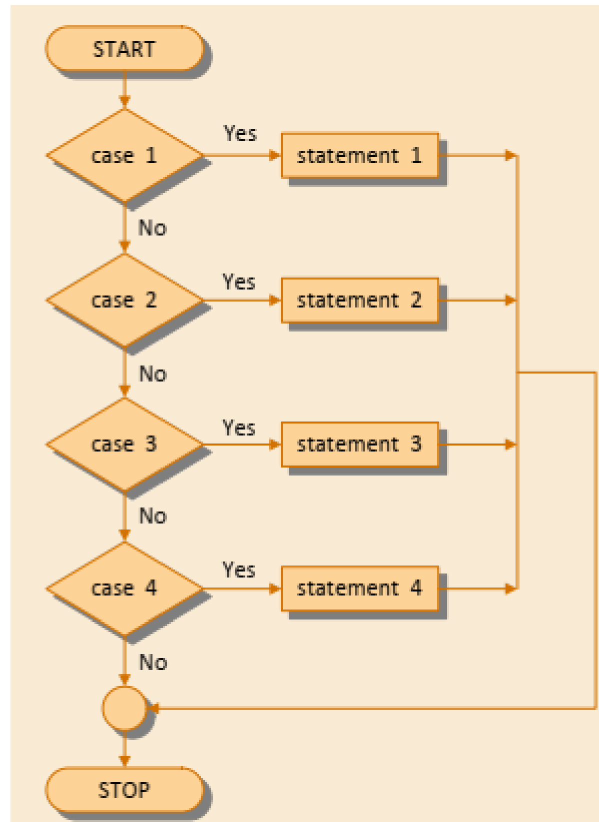
Example:

```
# include <stdio.h>
int main( )
{
        int i = 2 ;
        switch ( i )
        {
                case 1 :
                printf ( "I am in case 1 \n" ) ; break;
                case 2 :
                printf ( "I am in case 2 \n" ) ; break;
                case 3 :
                printf ( "I am in case 3 \n" ) ; break;
                default :
                printf ( "I am in default \n" ) ; break;
        }
        return 0 ;
}
```

//C using multiple case statements

```
# include <stdio.h>
int main( )
{
        char ch ;
        printf ( "Enter any one of the alphabets a, b, or c " ) ;
        scanf ( "%c", &ch ) ;
        switch ( ch )
        {
                case 'a' :
                case 'A' :
                        printf ( "a as in ashar\n" ) ;
                        break ;
                case 'b' :
                case 'B' :
                        printf ( "b as in brain\n" ) ;
                        break ;
                case 'c' :
                case 'C' :
                        printf ( "c as in cookie\n" ) ;
                        break ;
                default :
                        printf ( "wish you knew what are alphabets\n" ) ;
        }
        return 0;
}
```

Flow Chart



switch versus if-else Ladder

- "If-else" and "switch" both are selection statements.
- The selection statements, transfer the flow of the program to the particular block of statements based upon whether the condition is "true" or "false".

Basic Comparison	else if ladder	switch statement
Basic	Which statement will be executeddepend upon the output of the expression inside if statement.	Which statement will beexecuted is decided by user.
Expression	if-else statement uses multiple statement for multiple choices.	switch statement usessingle expression for multiple choices.
Testing	if-else statement test for equality as well as for logical expression.	switch statement testonly for equality.
Evaluation	if statement evaluates integer, character, pointer or floating-point type or boolean type.	switch statement evaluates only characteror integer value.

	if (N >= 2 && N <= 4)
	printf("N is in 2 and 4");
	else
	printf("N is not in range");
		switch(N)
		{
		case 2: case 3: case 4:
		printf("N is in 2 and 4");
		break;
		default:
		printf("N is not in range");
		}

The goto keyword

© The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement.
© The goto statement can be used to jump from anywhere to anywhere within a function.

Syntax

```
label:
//some part of the code;
goto label;
```

Example:

```
#include <stdio.h>
int main()
{
        int num,i=1;
        printf("Enter the number whose table you want to print?");
        scanf("%d",&num);
        table:
        printf("%d x %d = %d\n",num,i,num*i);
        i++;
        if(i<=5)
        goto table;
}
```

Output

```
Enter the number whose table you want to print?10
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
```

FUNCTIONS AND POINTERS

Function

© A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function.

© A function can be called multiple times to provide reusability and modularity to the C program. The function is also known as procedure or subroutine in other programming languages.

© There are basically two types of functions:

1. ***Library functions*** Ex. printf( ), scanf( ), etc.
2. ***User-defined functions*** Ex. message( ), etc.

- As the name suggests, library functions are nothing but commonly required functions grouped together and stored in a Library file on the disk. These library of functions come ready-made with development environments like Turbo C, Visual Studio, NetBeans, gcc, etc. The procedure for calling both types of functions is exactly same.

© There are three aspects needed for user-defined function.

© ***Function declaration*** – A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

© ***Function call*** – A function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration.

© ***Function definition*** – It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

Syntax

```
return_type function_name(data_type parameter...)
{
        //code to be executed
}
```

Example:

```
# include <stdio.h>
void message( ) ; /* function prototype declaration */
int main( )
{
        message( ) ; /* function call */
        printf ( "Cry, and you stop the monotony!\n" ) ;
        return 0 ;
}
void message( ) /* function definition */
{
        printf ( "Smile, and the world smiles with you...\n" ) ;
}
```

- The first is the function prototype declaration and is written as:

  void message( ) ;

  It indicates that message() is a function which after completing its execution does not return any value. This ‘does not return any value’ is indicated using ‘void’.

- The second is the function definition:

```
void message( )
{
    printf ( "Smile, and the world smiles with you...\n" ) ;
}
```

In this definition we are having only printf(), but we can also use if, for, while, switch, etc., within this function definition.

- The third is the function calling

```
message( ) ;
```

Here the function message( ) is being called by main( ).

1. A function gets called when the function name is followed by a semicolon (;).

```
message();
```

2. A function is defined when function name is followed by a pair of braces ({ }) in which one or more statements may be present.

```
void message()
{
    printf("Message1");
}
```

3. Any function can be called from any other function. Even main( ) can be called from other functions.

```
# include <stdio.h>
void message( ) ;
void main( )
{
    message( ) ;
}
```

4. A function can be called any number of times.

```
# include <stdio.h>
void message( ) ;
int main( )
{
    message( ) ;
    message( ) ;
}
```

5. The order in which the functions are defined in a program and the order in which they get called need not necessarily be same.
6. A function can call itself. Such a process is called ‘recursion’.
7. A function can be called from another function, but a function cannot be defined in another function.

Passing values between functions

There are two ways to pass parameters in C: Pass by Value, Pass by Reference.

1. ***Pass by Value.*** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Example:

```
#include <stdio.h>
void swap(int x, int y); /* function declaration */
int main ()
{
        int a = 100, b = 200;
        printf("Before swap, value of a : %d\t", a );
        printf("Before swap, value of b : %d\n", b );
        swap(a, b); /* calling a function to swap the values */
        printf("After swap, value of a : %d\t", a );
        printf("After swap, value of b : %d\n", b );
 }
void swap(int x, int y)
{
        int temp;
        temp = x; /* save the value of x */
        x = y;    /* put y into x */
        y = temp; /* put temp into y */
        return;
}
Before swap, value of a : 100          Before swap, value of b : 200
After swap, value of a : 100           After swap, value of b : 200
```

2. ***Pass by Reference***. This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

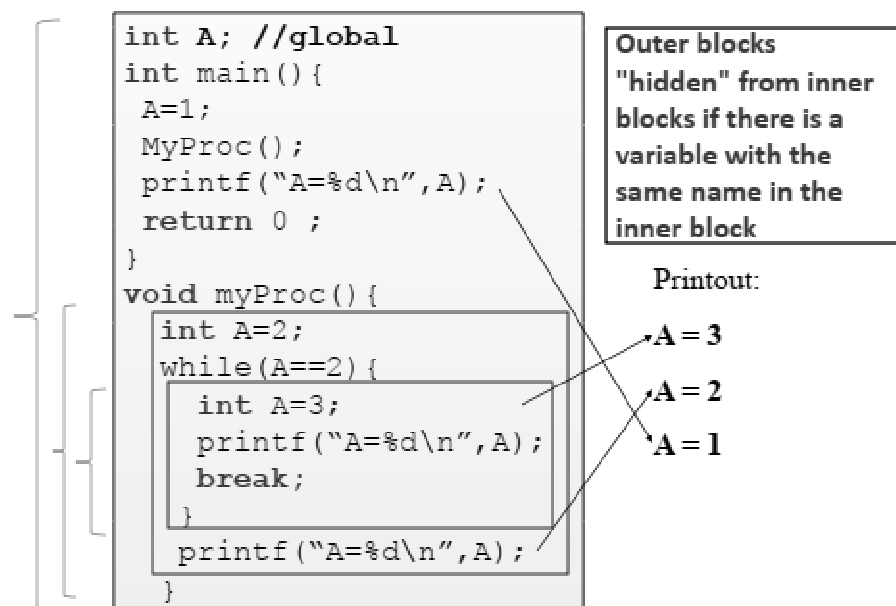
   Example:

```
#include <stdio.h>
void swap(int *x, int *y); /* function declaration */
int main ()
{
        int a = 100, b = 200;
        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );
        swap(&a, &b); /* calling a function to swap the values */
        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
 }
void swap(int *x, int *y)
{
        int temp = *x; /* save the value of x */
        *x = *y;    /* put y into x */
        *y = temp; /* put temp into y */
        return;
}
Before swap, value of a : 100          Before swap, value of b : 200
After swap, value of a : 200           After swap, value of b : 100
```

Scope rule of function

- © A Function scope begins at the opening of the function and ends with the closing of it.
- © A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.
- © A variable declared in a block can only be accessed inside the block and all inner blocks of this block.
- © If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the point of the declaration by inner block.
- © Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

Example:



Calling Conventions

The calling convention decides whether the parameters being passed to the function are pushed on the stack in left-to-right or right-to-left order.

The standard calling convention always uses the right-to-left order.

Library Functions

- © Library functions are inbuilt functions in C programming.
- © The prototype and data definitions of these functions are present in their respective header files.
- © Hence, when the library of functions is provided, a set of '.h' files is also provided. These files contain the prototype declarations of library functions.
- © For example,
  - © Prototypes of all input/output functions are provided in the file 'stdio.h', prototypes of all mathematical functions are provided in the file 'math.h', etc.
- © In the following example, we have called four standard library functions - sin( ), cos( ), tan( ) and pow( ).
- © As we know, before calling any function, we must declare its prototype. But since we didn't define the library functions (we merely called them), we do not know the prototype declarations of library functions.

© Prototypes of functions sin( ), cos( ), tan( ) and pow( ) are declared in the file 'math.h'.

```
# include <stdio.h>
# include <math.h>
int main( )
{
        float a = 0.5 ;
        float w, x, y, z ;
        w = sin ( a ) ;
        x = cos ( a ) ;
        y = tan ( a ) ;
        z = pow ( a, 2 ) ;
        printf ( "%f %f %f %f\n", w, x, y, z ) ;
        return 0 ;
}
```

Advanced features of functions

© *Reusability of Code*: Once a code has developed then it can use at any time.

© *Remove Redundancy:* user doesn't need to write code again and again.

© *Reduce Complexity:* Means a Large program will be Stored in the Two or More Functions. So that this will makes easy for a user to understand that Code and gives it a modular structure.

© Use of functions enhances the *readability* of a program.

© The C compiler follows top-to-down execution, so the control flow can be easily managed in case of functions. The control will always come back to the main() function.

The C Pre-processor

© Pre-processing is a program that will be executed automatically before passing the source program to the compiler.

© Pre-processing is under the control of pre-processor directives.

© All preprocessor directives start with a pound (#) symbol & should be not ended with a semicolon (;).

All types of Preprocessor Directives are as follows:

# define	**# if**
# include	**#else**
# ifdef	**#elif**
# undef	**#endif**
#ifndef	**#error**
	#pragma

In C programming language pre-processor directives are classified into 4 types, such as

1. Macro substitution directives. Example: #define
2. File inclusion directives. Example: #include
3. Conditional compilation directives. Example: #if, #else, #endif, #ifdef, #undef, etc.
4. Miscellaneous directives. Example: #pragma, #error, #line, etc.

1. Macro Definition

A macro is a piece of code in a program that is replaced by the value of the macro.
Macro is defined by #define directive.
Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.
There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants.
For example:

```
#define PI 3.14
```

- Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function-like macro looks like function call.
For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

- Here, MIN is the macro name.

Example:

```
#include <stdio.h>
// Macro definitions
#define IS_UPPER(x) (x >= 'A' && x <= 'Z')
#define IS_LOWER(x) (x >= 'a' && x <= 'z')
int main()
{
        char ch;
        printf("Enter any character: ");
        ch = getchar();
        if (IS_UPPER(ch))
                printf("'%c' is uppercase\n", ch);
        else if (IS_LOWER(ch))
                printf("'%c' is lowercase\n", ch);
        else
                printf("Entered character is not alphabet");
}
```

Output:

```
Enter any character: C
'C' is uppercase
```

2. File Inclusion Directives
   The #include preprocessor directive is used to paste code of given file into current file.
   It is used include system-defined and user-defined header files.
   There are two variants to use #include directive.
   1. #include <filename>
      - tells the compiler to look for the directory where system header files are held.
   2. #include "filename"
      - tells the compiler to look in the current directory from where program is running.

   Example

```
#include<stdio.h>
 int main()
{
        printf("Hello C");
        return 0;
 }
```

3. Conditional Compilation Directives

   1. #ifdef

   The #ifdef preprocessor directive checks if macro is defined by #define.
   If yes, it executes the code otherwise #else code is executed, if present.

   Syntax:

```
#ifdef MACRO
        //code
#endif
```

   Syntax with #else:

```
#ifdef MACRO
        //successful code
#else
        //else code
#endif
```

   Example

```
#include <stdio.h>
#define NOINPUT
void main()
{
        int a=0;
        #ifdef NOINPUT
                a=2;
        #else
                printf("Enter a:");
                scanf("%d", &a);
        #endif
        printf("Value of a: %d\n", a);
}
```

2. #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define.
If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

Syntax with #else:

```
#ifndef MACRO
//successful code
#else
//else code
#endif
```

Example

```
#include <stdio.h>
void main()
{
        int a=0;
        #ifndef INPUT
                a=2;
        #else
                printf("Enter a:");
                scanf("%d", &a);
        #endif
        printf("Value of a: %d\n", a);
}
```

Output:

```
Value of a: 2
```

3. #if

The #if preprocessor directive evaluates the expression or condition.
If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
#if expression
        //code
#endif
```

Example

```
#include <stdio.h>
#define NUMBER 0
void main()
{
        #if (NUMBER==0)
                printf("Value of Number is: %d",NUMBER);
        #endif
}
```

Syntax with #else:

```
#if expression
        //if code
#else
        //else code
#endif
```

Example

```
#include <stdio.h>
#define NUMBER 1
void main()
{
        #if NUMBER==0
                printf("Value of Number is: %d",NUMBER);
        #else
                printf("Value of Number is non-zero");
        #endif
}
```

Output:

Value of Number is non-zero

Syntax with #elif

```
#if expression
        //if code
#elif expression
        //elif code
#else
        //else code
#endif
```

Example

```
#include <stdio.h>
#define checker 5
int main()
{
        #if checker <= 3
                printf("This is the if directive block.\n");
        #elif checker > 3
                printf("This is the elif directive block.\n");
        #endif
}
```

Output

This is the elif directive block.

4. Miscellaneous Directives

1. #error

The #error preprocessor directive indicates error.

The compiler gives fatal error if #error directive is found and skips further compilation process.

Example

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main()
{
        float a;
        a=sqrt(7);
        printf("%f",a);
}
#endif
```

Output:

Compile Time Error: First include then compile

- But, if you include math.h, it does not gives error.

2. #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

```
#pragma token
```

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

#pragma argsused	#pragma exit	#pragma hdrfile
#pragma hdrstop	#pragma inline	#pragma option
#pragma saveregs	#pragma startup	#pragma warn

Example

```
#include<stdio.h>
void func() ;
#pragma startup func
#pragma exit func
void main()
{
        printf("\nI am in main");
}
void func()
{
        printf("\nI am in func");
}
```

Output:

```
I am in func
I am in main
I am in func
```

ARRAYS

The C language provides a capability that enables the user to design a set of similar data types, called array.

An array is a collection of homogeneous elements of same data type in sequenced memory location.

What are Arrays?

```
# include <stdio.h>
int main( )
{
        int x  ;
        x = 5 ;
        x = 10 ;
        printf ( "x = %d\n", x ) ;
        return 0 ;
}
```

This program will print the value of x as 10. Why so? Because, when a value 10 is assigned to x, the earlier value of x, i.e., 5 is lost. Thus, ordinary variables (the ones which we have used so far) are capable of holding only one value at a time (as in this example).

However, there are situations in which we would want to store more than one value at a time in a single variable.

For example, suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case, we have two options to store these marks in memory:

(a) Construct 100 variables to store percentage marks obtained by 100 different students, i.e., each variable containing one student's marks.

(b) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

For example, assume the following group of numbers, which represent percentage marks obtained by five students.

per = { 48, 88, 34, 23, 96 }

Here per is the subscripted variable (array), whereas i is its subscript.

Thus, an array is a collection of similar elements. These similar elements could be all ints, or all floats, or all chars, etc.

Usually, the array of characters is called a ***'string'***, whereas an array of int or floats is called simply an array.

```
# include <stdio.h>
int main( )
{
        int avg, sum = 0 ; int i ;
        int marks[ 30 ] ; /* array declaration */
        for ( i = 0 ; i <= 29 ; i++ )
        {
                printf ( "Enter marks " ) ;
                scanf ( "%d", &marks[ i ] ) ; /* store data in array */
        }
        for ( i = 0 ; i <= 29 ; i++ )
        sum = sum + marks[ i ] ; /* read data from an array*/ avg = sum / 30 ;
        printf ( "Average marks = %d\n", avg ) ;
        return 0 ;
}
```

Array Declaration
To begin with, like other variables, an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program, we have done this with the statement:

```
int marks[ 30 ];
```

Here, int specifies the type of the variable, just as it does with ordinary variables and the word marks specifies the name of the variable. The [30] however is new. The number 30 tells how many elements of the type int will be in our array. This number is often called the 'dimension' of the array. The bracket ([ ]) tells the compiler that we are dealing with an array.

Accessing Elements of an Array
Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, the number in the brackets following the array name.
This number specifies the element's position in the array. All the array elements are numbered, starting with 0. i.e) marks[9] - it represent 10th element

Entering Data into an Array
Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
{
    printf ( "Enter marks " ) ;
    scanf ( "%d", &marks[ i ] ) ;
}
```

- The 'for' loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times.
- The first time through the loop, i has a value 0, so the scanf( ) function will cause the value typed to be stored in the array element marks[0], the first element of the array.
- This process will be repeated until i becomes 29.

Reading Data from an Array
The balance of the program reads the data back out of the array and uses it to calculate the average.
The for loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called sum.
When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

Summary
(a) An array is a collection of similar elements or group of items with similar data
(b) The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
(c) An array is also known as a subscripted variable.
(d) Before using an array, its type and dimension must be declared.
(e) However big an array, its elements are always stored in contiguous memory locations.

More on Arrays
Array is a very popular data type with C programmers. This is because of the convenience with which arrays lend themselves to programming.

Array Initialization

So far we have used arrays that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how to initialize an array while declaring it. Following are a few examples that demonstrate this:

1. int num[6] = { 2, 4, 12, 5, 45, 5 } ;
2. int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
3. float press[ ] = { 12.3, 34.2, -23.4, -11.3 } ;

Note the following points carefully:

(a) Till the array elements are not given any specific values, they are supposed to contain garbage values.

(b) If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd and 3rd examples above.

Array Elements in Memory

Consider the following array declaration:

    int arr[8] ;

What happens in memory when we make this declaration?

32 bytes get immediately reserved in memory, 4 bytes each for the 8 integers.

And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be auto. If the storage class is declared to be static, then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations.

Bounds Checking

In C, there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size.

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value, we pass values of array elements to the function, whereas in the call by reference, we pass addresses of array elements to the function. These two calls are illustrated below.

```
# include <stdio.h>
void display ( int ) ;
int main( )
{
        int i ;
        int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
        for ( i = 0 ; i <= 6 ; i++ )
                display ( marks[ i ] ) ;
}
void display ( int m )
{
        printf ( "%d ", m ) ;
}
```

And here's the output... 55 65 75 56 78 78 90

- Here, we are passing an individual array element at a time to the function display( ) and getting it printed in the function display( ).
- Note that, since at a time only one element is being passed, this element is collected in an ordinary integer variable m, in the function display( ).

```
/* Demonstration of call by reference */
    #include <stdio.h>
    void disp ( int * ) ;
    int main( )
    {
        int i ;
        int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
        for ( i = 0 ; i <= 6 ; i++ )
            disp ( &marks[ i ] ) ;
    }
    void disp ( int *n )
    {
        printf ( "%d ", *n ) ;
    }
```

And here's the output... 55 65 75 56 78 78 90

- Here, we are passing addresses of individual array elements to the function disp( ).
- Hence, the variable in which this address is collected (n), is declared as a pointer variable.
- And since n contains the address of array element, to print out the array element, we are using the 'value at address' operator (*).

Pointers and Arrays

To be able to see what pointers have got to do with arrays, let us first learn some pointer arithmetic. Consider the following example:

```
# include <stdio.h>
int main( )
{
    int i = 3, *x ;
    float j = 1.5, *y ;
    char k = 'c', *z ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of j = %f\n", j ) ;
    printf ( "Value of k = %c\n", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "Original address in x = %u\n", x ) ;
    printf ( "Original address in y = %u\n", y ) ;
    printf ( "Original address in z = %u\n", z ) ; x++ ;
    y++ ; z++ ;
    printf ( "New address in x = %u\n", x ) ;
    printf ( "New address in y = %u\n", y ) ;
    printf ( "New address in z = %u\n", z ) ;
}
```

Here is the output of the program.

```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65528
New address in y = 65524
New address in z = 65520
```

Two-Dimensional Arrays

The two-dimensional array is also called a matrix. Let us see how to create this array and work with it. Here is a sample program that stores roll number and marks obtained by a student side-by-side in a matrix.

```
# include <stdio.h>
int main( )
{
        int stud[ 4 ][ 2 ] ;
        int i, j ;
        for ( i = 0 ; i <= 3 ; i++ )
        {
                printf ( "Enter roll no. and marks" ) ;
                scanf ( "%d %d", &stud[ i ][ 0 ], &stud[ i ][ 1 ] ) ;
        }
        for ( i = 0 ; i <= 3 ; i++ )
        printf ( "%d %d\n", stud[ i ][ 0 ], stud[ i ][ 1 ] ) ;
}
```

There are two parts to the program—in the first part, through a ‘for loop’, we read in the values of roll no. and marks, whereas, in the second part through another for loop, we print out these values.

Initialising a 2-Dimensional Array

How do we initialize a two-dimensional array? As simple as this...

int stud[4][2] = {{1234, 56},{1212, 33},{1434, 80},{1312, 78} } ;

or even this would work...

int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;

It is important to remember that, while initializing a 2-D array, it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

```
int arr[ 2 ][ 3 ] = { 12, 34, 23, 45, 56, 45 } ;
int arr[ ][ 3 ] = { 12, 34, 23, 45, 56, 45 } ;
```

are perfectly acceptable, whereas,

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

would never work.

Pointers and 2-Dimensional Arrays

More specifically, each row of a two dimensional array can be thought of as a one-dimensional array.

This is a very important fact if we wish to access array elements of a 2-D array using pointers. Thus, the declaration,

```
int s[ 5 ][ 2 ] ;
```

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers.

```
# include <stdio.h>
int main( )
{
    int s[ 4 ][ 2 ] = {{ 1234, 56 },{ 1212, 33 },{ 1434, 80 },{ 1312, 78 }} ;
    int i ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "Address of %d th 1-D array = %u\n", i, s[ i ] ) ;
}
```

And here is the output...

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65516
Address of 2 th 1-D array = 65524
Address of 3 th 1-D array = 65532
```

Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointer to a char, then can we not have a pointer to an array? We certainly can. The following program shows how to build and use it:

```
/* Usage of pointer to an array */
# include <stdio.h>
int main( )
{
    int s[ 4 ][ 2 ] = {{ 1234, 56 },{ 1212, 33 },{ 1434, 80 },{ 1312, 78 }} ;
    int ( *p )[ 2 ] ;
    int i, j, *pint ;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        p = &s[ i ] ;
        pint = ( int * ) p ;
        printf ( "\n" ) ;
        for ( j = 0 ; j <= 1 ; j++ )
            printf ( "%d ", *( pint + j ) ) ;
    }
}
```

And here is the output...

```
1234 56
1212 33
1434 80
1312 78
```

Array of Pointers

The way there can be an array of ints or an array of floats, similarly, there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

All rules that apply to an ordinary array apply to the array of pointers as well.

```
# include <stdio.h> int main( )
{
int *arr[ 4 ] ; /* array of integer pointers */ int i = 31, j = 5, k = 19, l = 71, m ;
arr[ 0 ] = &i ; arr[ 1 ] = &j ; arr[ 2 ] = &k ; arr[ 3 ] = &l ;
for ( m = 0 ; m <= 3 ; m++ ) printf ( "%d\n", * ( arr[ m ] ) ) ; return 0 ;
}
```

STRINGS

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings.
Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer.
Character arrays or strings are used by programming languages to manipulate text, such as words and sentences.
A string constant is a one-dimensional array of characters terminated by a null ('\0'). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies 1 byte of memory and the last character is always '\0'.
More about Strings

```
/* Program to demonstrate printing of a string */
# include <stdio.h>
int main( )
{
    char name[ ] = "Klinsman" ; int i = 0 ;
    while ( i <= 7 )
    {
        printf ( "%c", name[ i ] ) ; i++ ;
    }
    printf ( "\n" ) ;
}
```

And here is the output...

```
Klinsman
```

Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer. This is shown below.

```
char str[ ] = "Hello" ; char *p = "Hello" ;
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a char pointer to another char pointer

```
int main( )
{
    char str1[ ] = "Hello" ;
    char str2[ 10 ] ;
    char *s = "Good Morning" ;
    char *q ;
    str2 = str1 ; /* error */
    q = s ; /* works */
}
```

Two-Dimensional Array of Characters

```
# include <stdio.h>
# include <string.h>
# define FOUND 1
# define NOTFOUND 0
int main( )
{
        char masterlist[6][10]={"akshay","parag","raman","srinivas","gopal",”rajesh"};
        int i, flag, a ;
        char yourname[ 10 ] ;
        printf ( "Enter your name " ) ;
        scanf ( "%s", yourname ) ;
        flag = NOTFOUND ;
        for ( i = 0 ; i <= 5 ; i++ )
        {
                a = strcmp ( &masterlist[ i ][ 0 ], yourname ) ;
                if ( a == 0 )
                {
                        printf ( "Welcome, you can enter the palace\n" ) ;
                        flag = FOUND ;
                        break ;
                }
        }
        if ( flag == NOTFOUND )
        printf ( "Sorry, you are a trespasser\n" ) ; return 0 ;
}
```

And here is the output for two sample runs of this program...

```
Enter your name dinesh
Sorry, you are a trespasser
Enter your name raman
Welcome, you can enter the palace
```

Array of Pointers to Strings

As we know, a pointer variable always contains an address.
Therefore, if we construct an array of pointers, it would contain a number of addresses.
Let us see how the names in the earlier example can be stored in the array of pointers.

char *names[ ] = {"akshay", "parag", "raman", "srinivas", "gopal", "rajesh" } ;

- In this declaration, names[ ] is an array of pointers.
- It contains base addresses of respective names.
- That is, base address of “akshay” is stored in names[0], base address of “parag” is stored in names[1] and so on.

Limitation of Array of Pointers to String

When we are using a two-dimensional array of characters, we are at liberty to either initialize the strings where we are declaring the array, or receive the strings using scanf( ) function. However, when we are using an array of pointers to strings, we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using scanf( ).

Example

```
# include <stdio.h>
int main( )
{
        char *names[ 6 ] ; int i ;
        for ( i = 0 ; i <= 5 ; i++ )
        {
                printf ( "Enter name " ) ;
                scanf ( "%s", names[ i ] ) ;
        }
        return 0 ;
}
```

- The program doesn't work because; when we are declaring the array, it is containing garbage values.

And it would be definitely wrong to send these garbage values to scanf( ) as the addresses where it should keep the strings received from the keyboard.

STRUCTURE

Why Use Structures

A structure contains a number of data types grouped together. These data types may or may not be of the same type.

The following example illustrates the use of this data type:

```
# include <stdio.h>
int main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;
    struct book b1, b2, b3 ;
    printf ( "Enter names, prices & no. of pages of 3 books\n" ) ;
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;
    printf ( "And this is what you entered\n" ) ;
    printf ( "%c %f %d\n", b1.name, b1.price, b1.pages ) ;
    printf ( "%c %f %d\n", b2.name, b2.price, b2.pages ) ;
    printf ( "%c %f %d\n", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512
And this is what you entered
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

Array of Structures

/* Usage of an array of structures */

```
# include <stdio.h>
void linkfloat( ) ;
int main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;
```

```
        struct book b[ 100 ] ; int i ;
        for ( i = 0 ; i <= 99 ; i++ )
        {
                printf ( "Enter name, price and pages " ) ;
                fflush ( stdin ) ;
                scanf ( "%c %f %d", &b[ i ].name, &b[ i ].price, &b[ i ].pages ) ;
        }
                for ( i = 0 ; i <= 99 ; i++ )
                printf ( "%c %f %d\n", b[ i ].name, b[ i ].price, b[ i ].pages ) ; return 0 ;
}
void linkfloat( )
{
        float a = 0, *b ;
        b = &a ; /* cause emulator to be linked */
        a = *b ; /* suppress the warning - variable not used */
}
```

Additional Features of Structures

- © The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.
- © One structure can be nested within another structure. Using this facility, complex data types can be created.
- © Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one shot.
- © The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct. Such pointers are known as 'structure pointers'.

Uses of Structures

Where are structures useful? The immediate application that comes to the mind is Database Management.

That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company, etc.

They can be used for a variety of purposes like:

(a) Changing the size of the cursor
(b) Clearing the contents of the screen
(c) Placing the cursor at an appropriate position on screen
(d) Drawing any graphics shape on the screen
(e) Receiving a key from the keyboard
(f) Checking the memory size of the computer
(g) Finding out the list of equipment attached to the computer
(h) Formatting a floppy
(i) Hiding a file from the directory
(j) Displaying the directory of a disk
(k) Sending the output to printer
(l) Interacting with the mouse

CONSOLE INPUT / OUTPUT

Types of I/O

There are numerous library functions available for I/O. These can be classified into two broad categories:

(a) Console I/O functions - Functions to receive input from keyboard and write output to VDU.

(b) File I/O functions - Functions to perform I/O operations on a floppy disk or a hard disk.

Console I/O Functions

Formatted Console I/O Functions

The functions printf() and scanf() fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form.

printf ("format string", list of variables) ;

The format string can contain:

(a) Characters that are simply printed as they are
(b) Format specifications that begin with a % sign
(c) Escape sequences that begin with a \ sign

Example

- printf ("Average = %d\nPercentage = %f\n", avg, per) ;
- The %d and %f used in the printf( ) are called format specifiers. They tell printf( ) to print the value of avg as a decimal integer and the value of per as a float.

The general form of scanf( ) statement is as follows:

scanf ("format string", list of addresses of variables) ;

Example

- scanf ("%d %f %c", &c, &a, &ch) ;
- Note that we are sending addresses of variables (addresses are obtained by using '&' the 'address of' operator) to scanf( ) function.
- This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses.

sprintf( ) and sscanf( ) Functions

© The sprintf( ) function works similar to the printf( ) function except for one small difference.

© Instead of sending the output to the screen as printf( ) does, this function writes the output to an array of characters. The following program illustrates this:

```
# include <stdio.h>
int main( )
{
        int i = 10 ;
        char ch = 'A' ;
        float a = 3.14 ;
        char str[ 20 ] ;
        printf ( "%d %c %f\n", i, ch, a ) ;
        sprintf ( str, "%d %c %f", i, ch, a ) ;
        printf ( "%s\n", str ) ;
}
```

- In this program, the printf( ) prints out the values of i, ch and a on the screen, whereas sprintf() stores these values in the character array str.
- The counterpart of sprintf( ) is the sscanf( ) function.
- It allows us to read characters from a string and to convert and store them in C variables according to specified formats.
- The sscanf( ) function comes in handy for in-memory conversion of characters to values.

Unformatted Console I/O Functions

putch( ) and putchar( )

© They print a character on the screen.

© As far as the working of putch( ) putchar( ) and fputchar( ) is concerned, it’s exactly same.

© The following program illustrates this:

```
# include <stdio.h>
# include <conio.h>
int main( )
{
    char ch = 'A' ;
    putch ( ch ) ;
    putchar ( ch ) ;
    fputchar ( ch ) ;
    putch ( 'Z' ) ;
    putchar ( 'Z' ) ;
    fputchar ( 'Z' ) ;
}
```

gets() and puts()

© gets( ) receives a string from the keyboard.

© Why is it needed?

   ➢ Because scanf( ) function has some limitations while receiving string of characters.

© The puts( ) function works exactly opposite to gets( ) function.

   ➢ It outputs a string to the screen.

© The following example illustrates:

```
# include <stdio.h>
int main( )
{
        char footballer[ 40 ] ;
        puts ( "Enter name" ) ;
        gets ( footballer ) ; /* sends base address of array */
        puts ( "Happy footballing!" ) ;
        puts ( footballer ) ;
        return 0 ;
}
```

FILE INPUT / OUTPUT DATA ORGANIZATION

- © Before we start doing file input/output let us first find out how data is organized on the disk.
- © All data stored on the disk is in binary form. How this binary data is stored on the disk varies from one OS to another.
- © However, this does not affect the C programmer since he has to use only the library functions written for the particular OS to be able to perform input/output.
- © It is the compiler vendor's responsibility to correctly implement these library functions by taking the help of OS. This is illustrated in Figure

File Operations

There are different operations that can be carried out on a file.
These are:

(a) Creation of a new file
(b) Opening an existing file
(c) Reading from a file / Writing to a file
(d) Moving to a specific location in a file (seeking)
(e) Closing a file

Let us now write a program to read a file and display its contents on the screen.

```
/* Display contents of a file on screen. */ # include <stdio.h>
int main( )
{
        FILE *fp ;
        char ch ;
        fp = fopen ( "PR1.C", "r" ) ;
        while ( 1 )
        {
                ch = fgetc ( fp ) ;
                if ( ch == EOF )
                        break ;
                printf ( "%c", ch ) ;
        }
        printf ( "\n" ) ;
        fclose ( fp ) ;
        return 0 ;
}
```

On execution of this program it displays the contents of the file 'PR1.C' on the screen

Opening a File

- Before we can read / write information from / to a file on a disk we must open the file.
- To open the file we have called the function fopen( ).
- It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file.

Reading from a File
Once the file has been opened for reading using fopen( ), as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer.

Closing the File
When we have finished reading from the file, we need to close it.
This is done using the function fclose( ) through the statement,

```
fclose ( fp ) ;
```

Once we close the file, we can no longer read from it using getc( ) unless we reopen the file.

Counting Characters, Tabs, Spaces

```
/* Count chars, spaces, tabs and newlines in a file */
    # include <stdio.h>
    int main( )
    {
        FILE *fp ; char ch ;
        int nol = 0, not = 0, nob = 0, noc = 0 ;
        fp = fopen ( "PR1.C", "r" ) ;
        while ( 1 )
        {
            ch = fgetc ( fp ) ; if ( ch == EOF ) break ;
            noc++ ;
            if ( ch == ' ' )
                nob++ ;
            if ( ch == '\n' )
                nol++ ;
            if ( ch == '\t' )
                not++ ;
        }
        fclose ( fp ) ;
        printf ( "Number of characters = %d\n", noc ) ;
        printf ( "Number of blanks = %d\n", nob ) ;
        printf ( "Number of tabs = %d\n", not ) ;
        printf ( "Number of lines = %d\n", nol ) ;
    }
```

Here is a sample run...

```
Number of characters = 125
Number of blanks = 25
Number of tabs = 13
Number of lines = 22
```

A File-copy Program
This program takes the contents of a file and copies them into another file, character-by-character.
Example

```
# include <stdio.h>
# include <stdlib.h>
int main( )
{
        FILE *fs, *ft ; char ch ;
        fs = fopen ( "pr1.c", "r" ) ;
        if ( fs == NULL )
        {
                puts ( "Cannot open source file" ) ;
                exit ( 1 ) ;
        }
        ft = fopen ( "pr2.c", "w" ) ;
        if ( ft == NULL )
        {
                puts ( "Cannot open target file" ) ;
                fclose ( fs ) ;
                exit ( 2 ) ;
        }
        while ( 1 )
        {
                ch = fgetc ( fs ) ;
                if ( ch == EOF )
                        break ;
                else
                        fputc ( ch, ft ) ;
        }
        fclose ( fs ) ; fclose ( ft ) ;
}
```

File Opening Modes

The tasks performed by fopen(), when a file is opened in each of these modes, also mentioned
"r" – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened, fopen() returns NULL.
Operations possible – reading from the file.
"w" – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
Operations possible – writing to the file.
"a" – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
Operations possible - adding new contents at the end of file.
Other Opening modes are
"r+" – Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.

Operations possible - reading existing, writing new, modifying existing contents of the file.
"w+" – Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file. Operations possible - writing new contents, reading them back and modifying existing contents of the file.
"a+" – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

Record I/O in Files
So far, we have dealt with reading and writing only characters and strings.
What if we want to read or write numbers from/to file?
what if we desire to read/write a combination of characters, strings and numbers?
For this first we would organize this dissimilar data together in a structure and then use fprintf() and fscanf() library functions to read/write data from/to file.
Following program illustrates the use of structures for writing records of employees:
/* Writes records to a file using structure */

```
# include <stdio.h>
# include <conio.h>
int main( )
{
	FILE *fp ;
	char another = 'Y' ;
	struct emp
	{
		char name[ 40 ] ;
		int age ;
		float bs ;
	} ;
	struct emp e ;
	fp = fopen ( "EMPLOYEE.DAT", "w" ) ;
	if ( fp == NULL )
	{
		puts ( "Cannot open file" ) ;
		exit ( 1 ) ;
	}
	while ( another == 'Y' )
	{
		printf ( "\nEnter name, age and basic salary: " ) ;
		scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
		fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;
		printf ( "Add another record (Y/N) " ) ;
		fflush ( stdin ) ; another = getche( ) ;
	}
	fclose ( fp ) ;
}
```

And here is the output of the program...

```
Enter name, age and basic salary: Sunil 34 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Sameer 21 1300.50
Add another record (Y/N) Y
Enter name, age and basic salary: Rahul 34 1400.55
Add another record (Y/N) N
```

- In this program we are just reading the data into a structure variable using scanf( ), and then dumping it into a disk file using fprintf( ).
- The user can input as many records as he/she desires.
- The procedure ends when the user supplies 'N' for the question 'Add another record (Y/N)'.

/* Read records from a file using structure */

```
# include <stdio.h>
# include <stdlib.h> int main( )
{
    FILE *fp ;
    struct emp
    {
        char name[ 40 ] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;
    fp = fopen ( "EMPLOYEE.DAT", "r" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit ( 1 ) ;
    }
    while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
        printf ( "%s %d %f\n", e.name, e.age, e.bs ) ;
    fclose ( fp ) ; return 0 ;
}
```

And here is the output of the program...

```
Sunil 34 1250.500000
Sameer 21 1300.500000
Rahul 34 1400.500000
```