C - Basic Introduction

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the UNIX operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972.

The UNIX operating system and virtually all UNIX applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- By 1973 UNIX OS was almost totally written in C.
- Today C is the most widely used System Programming Language.
- Most of the state of the art software have been implemented using C

Why to use C?

C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C Compilers

When you write any program in C language then to run that program you need to compile that program using a C Compiler which converts your program into a language understandable by a computer. This is called machine language (i.e. binary format). So before proceeding, make sure you have C Compiler available at your computer. Some examples of C compilers are Turbo C and Borland C.

C - Program Structure

A C program basically has the following form:

Preprocessor Commands
Functions
Variables
Statements & Expressions
Comments

Preprocessor Commands: This command tells the compiler to do preprocessing before doing actual compilation. Like;
#include <stdio.h> is a preprocessor command which tells a C compiler to include stdio.h file before going to actual compilation. You will learn more about C Preprocessors in C Preprocessors session.

Functions: are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. This function is prefixed with keyword *int* which means this function returns an integer value when it exits. This integer value is retured using *return* statement.
The C Programming language provides a set of built-in functions. *printf()* is a C built-in function which is used to print anything on the screen.

Variables: Variables are used to hold numbers, strings and complex data for manipulation.

Statements & Expressions: Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments: are used to give additional useful information inside a C Program. All the comments will be put inside /*...*/ as given in the example above. A comment can span through multiple lines.

Note the followings

- C is a case sensitive programming language. It means in C *printf* and *Printf* will have different meanings.
- C has a free-form line structure. End of each C statement must be marked with a semicolon.
- Multiple statements can be one the same line.
- White Spaces (ie tab space and space bar) are ignored.
- Statements can continue over multiple lines.

Data Types in C

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement in the program he is going to do.

Primary data type

All C Compilers accept the following fundamental data types

1.	Integer	int
2.	Character	char
3.	Floating Point	float
4.	Double precision floating point	double
5.	Void	void

Integer Type

Integers are whole numbers with a machine dependent range of values. A good programming language as to support the programmer by giving a control on a range of numbers and storage space. C has 3 classes of integer storage namely short int, int and long int. All of these data types have signed and unsigned forms. A short int requires half the space than normal integer values. Unsigned numbers are always positive and consume all the bits for the magnitude of the number. The long and unsigned integers are used to declare a longer range of values.

Floating Point Types

Floating point number represents a real number with 6 digits precision. Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision. To extend the precision further we can use long double which consumes 80 bits of memory space.

Void Type

Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

Character Type

A single character can be defined as a defined as a character type of data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitly applied to char. While unsigned characters have values between 0 and 255, signed characters have values from –128 to 127.

Size and Range of Data Types on 16 bit machine;

TYPE	**SIZE (Bits)**	**Range**
Char or Signed Char	8	-128 to 127
Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

Declaration of Variables
Every variable used in the program should be declared to the compiler. The declaration does two things.

1. Tells the compiler the variables name.
2. Specifies what type of data the variable will hold.

The general format of any declaration

datatype v1, v2, v3, ……..,vn;

Where v1, v2, v3 are variable names. Variables are separated by commas. A declaration statement must end with a semicolon.

Example:

int sum;
int number, salary;
double average, mean;

Datatype	**Keyword Equivalent**
Character	char
Unsigned Character	unsigned char
Signed Character	signed char
Signed Integer	signed int (or) int
Signed Short Integer	signed short int (or) short int (or) short
Signed Long Integer	signed long int (or) long int (or) long
UnSigned Integer	unsigned int (or) unsigned
UnSigned Short Integer	unsigned short int (or) unsigned short
UnSigned Long Integer	unsigned long int (or) unsigned long
Floating Point	float
Double Precision Floating Point	double
Extended Double Precision Floating Point	long double

User defined type declaration
In C language a user can define an identifier that represents an existing data type. The user defined datatype identifier can later be used to declare variables. The general syntax is

typedef type identifier;

here type represents existing data type and ‘identifier’ refers to the ‘row’ name given to the data type.

Example:

typedef int salary;

typedef float average;

Here salary symbolizes int and average symbolizes float. They can be later used to declare variables as follows:

Units dept1, dept2;
Average section1, section2;

Therefore dept1 and dept2 are indirectly declared as integer datatype and section1 and section2 are indirectly float data type.

The second type of user defined datatype is enumerated data type which is defined as follows.

Enum identifier {value1, value2 …. Value n};

The identifier is a user defined enumerated datatype which can be used to declare variables that have one of the values enclosed within the braces. After the definition we can declare variables to be of this 'new' type as below.

enum identifier V1, V2, V3, …… Vn

The enumerated variables V1, V2, ….., Vn can have only one of the values value1, value2 ….. Value n

Example 1:

```
enum day {Monday, Tuesday, …. Sunday};
enum day week_st, week end;
week_st = Monday;
week_end = Friday;
if (wk_st == Tuesday)
week_en = Saturday;
```

Example 2:

```
#include <stdio.h>

int main() {
  enum {RED=5, YELLOW, GREEN=4, BLUE};

  printf("RED = %d\n", RED);
  printf("YELLOW = %d\n", YELLOW);
  printf("GREEN = %d\n", GREEN);
  printf("BLUE = %d\n", BLUE);
  return 0;
}
```

This will produce following results

```
RED = 5
YELLOW = 6
GREEN = 4
BLUE = 5
```

C Programming - Constants and Variables

Instructions in C language are formed using syntax and keywords. It is necessary to strictly follow C language Syntax rules. Any instruction that mis-matches with C language Syntax generates an error while compiling the program. All programs must confirm to rules pre-defined in C Language. Keywords as special words which are exclusively used by C language, each keyword has its own meaning and relevance hence, Keywords should not be used either as Variable or Constant names.

Character Set

The character set in C Language can be grouped into the following categories.

1. Letters
2. Digits
3. Special Characters
4. White Spaces

White Spaces are ignored by the compiler until they are a part of string constant. White Space may be used to separate words, but are strictly prohibited while using between characters of keywords or identifiers.

C Character-Set Table

Letters	Upper Case A to Z
Digits	0 to 9
Lower Case	a to z

.

Special Characters

,	Comma	**&**	Ampersand
.	Period	^	Caret
;	Semicolon	*****	Asterisk
:	Colon	**-**	Minus Sign
?	Question Mark	+	Plus Sign
'	Aphostrophe	<	Opening Angle (Less than sign)
"	Quotation Marks	>	Closing Angle (Greater than sign)
!	Exclaimation Mark	**(**	Left Parenthesis
|	Vertical Bar	**)**	Right Parenthesis
/	Slash	**[**	Left Bracket
\	Backslash	**]**	Right Bracket
~	Tilde	**{**	Left Brace
-	Underscore	**}**	Right Bracket
$	Dollar Sign	#	.Number Sign
%	Percentage Sign		

White Space

1. Blank Space
2. Horizontal Tab
3. Carriage Return
4. New Line
5. Form Feed

Keywords and Identifiers

Every word in C language is a keyword or an identifier. Keywords in C language cannot be used as a variable name. They are specifically used by the compiler for its own purpose and they serve as building blocks of a c program.

The following are the Keyword set of C language.

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

Some compilers may have additional keywords listed in C manual.

Identifier refers to the name of user-defined variables, array and functions. A variable should be essentially a sequence of letters and or digits and the variable name should begin with a character.

Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifiers.

The identifiers must conform to the following rules.

1. First character must be an alphabet (or underscore)
2. Identifier names must consists of only letters, digits and underscore.
3. An identifier name should have less than 31 characters.
4. Any standard C language keyword cannot be used as a variable name.
5. An identifier should not contain a space.

Constants

A constant value is the one which does not change during the execution of a program. C supports several types of constants.

1. Integer Constants
2. Real Constants
3. Single Character Constants
4. String Constants

Integer Constants

An integer constant is a sequence of digits. There are 3 types of integer's namely decimal integer, octal integers and hexadecimal integer.

Decimal Integers: consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non digit characters are not permitted between digits. Examples for valid decimal integer constants are:

123
-31
0
562321
+ 78

Some examples for invalid integer constants are:

15 750
20,000
Rs. 1000

Octal Integers: constant consists of any combination of digits from 0 through 7 with an O at the beginning. Some examples of octal integers are:

O26
O
O347
O676

Hexadecimal integer: constant is preceded by OX or Ox, they may contain alphabets from A to F or a to f. The alphabets A to F refer to 10 to 15 in decimal digits. Examples of valid hexadecimal integers are:

OX2
OX8C
OXbcd
Ox

Real Constants

Real Constants consists of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously. These quantities are represented by numbers containing fractional parts like 26.082. Examples of real constants are:

0.0026
-0.97
435.29
+487.0

Real Numbers can also be represented by exponential notation. The general form for exponential notation is mantissa exponent. The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign.

A Single Character constant represent a single character which is enclosed in a pair of quotation symbols.

Example for character constants are:

'5'
'x'
';'
' '

All character constants have an equivalent integer value which is called ASCII Values.

String Constants

A string constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be an alphabet, number, special character and blank space. Example of string constants are

"BSCCSIT"
"1234"
"God Bless"
"!.....?"

Backslash Character Constants [Escape Sequences]

Backslash character constants are special characters used in output functions. Although they contain two characters they represent only one character. Given below is the table of escape sequence and their meanings.

Constant	Meaning
'\a'	Audible Alert (Bell)
'\b'	Backspace
'\f'	Formfeed
'\n'	New Line
'\r'	Carriage Return
'\t'	Horizontal tab
'\v'	Vertical Tab
'\''	Single Quote
'\"'	Double Quote
'\?'	Question Mark
'\\'	Back Slash
'\0'	Null

Variables

A variable is a value that can change any time. It is a memory location used to store a data value. A variable name should be carefully chosen by the programmer so that its use is reflected in a useful way in the entire program. Variable names are case sensitive. Examples of variable names are

Sun
number
Salary
Emp_name
average1

Any variable declared in a program should confirm to the following:

1. They must always begin with a letter, although some systems permit underscore as the first character.
2. The length of a variable must not be more than 8 characters.
3. White space is not allowed and
4. A variable should not be a Keyword
5. It should not contain any special characters.

Examples of Invalid Variable names are:

123
(area)
6th
%abc

Operators

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C has a rich set of operators which can be classified as

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Unary Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

1. Arithmetic Operators

All the basic arithmetic operations can be carried out in C. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C language. Unary operations operate on a singe operand, therefore the number 5 when operated by unary – will have the value –5.

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Examples of arithmetic operators are:

x + y
x - y
-x + y
a * b + c
-a * b
etc.,

Here a, b, c, x, y are known as operands. The modulus operator is a special operator in C language which evaluates the remainder of the operands after division.

Example

```
#include(stdio.h)                    //include header file stdio.h
void main()                          //tell the compiler the start of the program
{
    int numb1, num2, sum, sub, mul, div, mod;   //declaration of variables
    scanf ("%d %d", &num1, &num2);              //inputs the operands
    sum = num1+num2;                            //addition of numbers and storing in sum.
    printf("\n Thu sum is = %d", sum);          //display the output
    sub = num1-num2;                            //subtraction of numbers and storing in sub.
    printf("\n Thu difference is = %d", sub);   //display the output
    mul = num1*num2;                            //multiplication of numbers and storing in mul.
    printf("\n Thu product is = %d", mul);      //display the output
    div = num1/num2;                            //division of numbers and storing in div.
    printf("\n Thu division is = %d", div);     //display the output
```

```
        mod = num1%num2;                    //modulus of numbers and storing in mod.
        printf("\n Thu modulus is = %d", mod);    //display the output
}
```

Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let x = 27 and y = 5 be 2 integer numbers. Then the integer operation leads to the following results.

x + y = 32
x – y = 22
x * y = 115
x % y = 2
x / y = 5

In integer division the fractional part is truncated.

Floating point arithmetic

When an arithmetic operation is preformed on two real numbers or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The remainder operator is not applicable for floating point arithmetic operands.

Let x = 14.0 and y = 4.0 then

x + y = 18.0
x – y = 10.0
x * y = 56.0
x / y = 3.50

Mixed mode arithmetic
When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real thus 15/10.0 = 1.5

2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator comes into picture. C supports the following relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

It is required to compare the marks of 2 students, salary of 2 persons; we can compare those using relational operators.

A simple relational expression contains only one relational operator and takes the following form.

exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25 TRUE
-65 > 0 FALSE
10 < 7 + 5 TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

3. Logical Operators

C has the following logical operators; they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND (&&)

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Example: a > b && x = = 10

The expression to the left is a > b and that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if "a" is greater than "b" and "x" is equal to 10.

Logical OR (||)

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example: a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if "a" is less than either "m" or "n" and when "a" is less than both "m" and "n".

Logical NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

Example: ! (x >= y)
The NOT expression evaluates to true only if the value of "x" is neither greater than or equal to "y".

4. Assignment Operators

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

Example x = a + b

Here the value of a + b is evaluated and substituted to the variable x. In addition, C has a set of shorthand assignment operators of the form.

var oper = exp;

Here var is a variable, exp is an expression and oper is a C binary arithmetic operator. The operator oper = is known as shorthand assignment operator.

Example x + = 1 is same as x = x + 1

The commonly used shorthand assignment operators are as follows Shorthand assignment operators

Statement with simple assignment operator	**Statement with shorthand operator**
a = a + 1	a += 1
a = a – 1	a -= 1
a = a * (n+1)	a *= (n+1)
a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

Example for using shorthand assignment operator:

```
#define N 100                    //creates a variable N with constant value 100
#define A 2                      //creates a variable A with constant value 2
main()                           //start of the program
{
        int a;                   //variable a declaration
        a = A;                   //assigns value 2 to a
        while (a < N)            //while value of a is less than N
        {                        //evaluate or do the following
                printf("%d \n",a);   //print the current value of a
                a *= a;          //shorthand form of a = a * a
          }                      //end of the loop
}                                //end of the program
```

Output
2
4
16

5. Unary Operators

The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below:
1. ++ variable name
2. variable name++

3. – –variable name
4. variable name– –

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator – – subtracts the value 1 from the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following :

m = 5;
y = ++m; (prefix)
In this case the value of y and m would be 6

Suppose if we rewrite the above statement as
m = 5;
y = m++; (post fix)

Then the value of y will be 5 and that of m will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

6. Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark (?) and the colon (:) The syntax for a ternary operator is as follows:

exp1 ? exp2 : exp3

The ternary operator works as follows:
exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

Note that only one of the expressions is evaluated.

For example
```
a = 10;
b = 15;
x = (a > b)? a: b
```

Here "x" will be assigned to the value of "b". The condition follows that the expression is false therefore "b" is assigned to "x".

```
/* Example : to find the maximum value using conditional operator*/
#include<stdio.h>
void main()                          //start of the program
{
        int i,j,larger;                      //declaration of variables
        printf ("Input 2 integers : ");      //ask the user to input 2 numbers
        scanf("%d %d",&i, &j);               //take the number from standard input and store it
        larger = i > j ? i : j;              //evaluation using ternary operator
        printf("The largest of two numbers is %d \n", larger);          // print the largest number
}                                                                       // end of the program
```

Output
Input 2 integers: 34

45
The largest of two numbers is 45

7. Bitwise Operators

C has a distinction of supporting special operators known as bitwise operators for manipulation data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right on left. Bitwise operators may not be applied to a float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

8. Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->). The size of and the comma operators are discussed here. The remaining operators are discussed in forth coming chapters.

The Comma Operator

The comma operator can be used to link related expressions together. Comma-linked lists of expressions are evaluated left to right and value of right most expression is the value of the combined expression.
For example the statement: value = (x = 10, y = 5, x + y);

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are:

In for loops:
for (n=1, m=10; n <=m; n++,m++)

In while loops
While (c=getchar(), c != '10')

Exchanging values
t = x, x = y, y = t;

The sizeof Operator

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.
Example:
```
m = sizeof (sum);
n = sizeof (long int);
k = sizeof (235L);
```

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.
Example program that employs different kinds of operators. The results of their evaluation are also shown in comparison:

```
#include<stdio.h>
main()                                          //start of program
{
        int a, b, c, d;                         //declaration of variables
        a = 15; b = 10; c = ++a-b;              //assign values to variables
        printf ("a = %d, b = %d, c = %d\n", a,b,c);   //print the values
        d=b++ + a;
        printf ("a = %d, b = %d, d = %d\n, a,b,d);
        printf ("a / b = %d\n, a / b);
        printf ("a %% b = %d\n, a % b);
        printf ("a *= b = %d\n, a *= b);
        printf ("%d\n, (c > d) ? 1 : 0 );
        printf ("%d\n, (c < d) ? 1 : 0 );
}
```

Notice the way the increment operator ++ works when used in an expression. In the statement:
c = ++a – b;
New value a = 16 is used thus giving value 6 to C. That is "a" is incremented by 1 before using in expression.
However in the statement d = b++ + a; The old value b = 10 is used in the expression. Here "b" is incremented after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement.

printf("a %% b = %d\n", a%b);

This program also illustrates that the expression

c > d ? 1 : 0

Assumes the value 0 when c is less than d and 1 when c is greater than d.

C Expressions

Arithmetic Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2 + 2x + 1$	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example of evaluation statements are

```
x = a * b – c
y = b / c * a
z = a – b / c + d;
```

The following program illustrates the effect of presence of parenthesis in expressions.

```
main ()
{
float a, b, c x, y, z;
a = 9;
b = 12;
c = 3;
x = a – b / 3 + c * 2 – 1;
y = a – b / (3 + c) * (2 – 1);
z = a – ( b / (3 + c) * 2) – 1;
printf (“x = %fn”,x);
printf (“y = %fn”,y);
printf (“z = %fn”,z);
}
```

output

```
x = 10.00
y = 7.00
z = 4.00
```

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority * / %
Low priority + -

Rules for evaluation of expression

- First parenthesized sub expression left to right are evaluated.
- If parenthesis are nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When Parenthesis are used, the expressions within parenthesis assume highest priority.

Operator precedence and associativity

Each operator in C has a precedence associated with it. The precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators of higher precedence are evaluated first.

The operators of same precedence are evaluated from right to left or from left to right depending on the level. This is known as associativity property of an operator.

The table given below gives the precedence of each operator.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Input Output Operations

Introduction

One of the essential operations performed in a C language programs is to provide input values to the program and output the data produced by the program to a standard output device. We can assign values to variable through assignment statements such as x = 5 a = 0 ; and so on. Another method is to use the Input then scanf() which can be used to read data from a key board. For outputting results we have used extensively the function printf() which sends results out to a terminal. There exists several functions in 'C' language that can carry out input output operations. These functions are collectively known as standard Input/Output Library. Each program that uses standard input / out put function must contain the statement.

include < stdio.h >

at the beginning.

Single character input output:

The basic operation done in input output is to read characters from the standard input device such as the keyboard and to output or writing it to the output unit usually the screen. The getchar function can be used to read a character from the standard input device. The scanf can also be used to achieve the function. The getchar has the following form.

Variable name = getchar():

Variable name is a valid 'C' variable, that has been declared already and that possess the type char.

Example program:

```
# include < stdio.h >                    // assigns stdio-h header file to your program
void main ( )                            // Indicates the starting point of the program.
{
        char C,                                  // variable declaration
        printf ("Type one character:") ;         // message to user
        C = getchar () ;                 // get a character from key board and Stores it in variable C.
        Printf (" The character you typed is = %c", C) ;        // output Statement
}
```

The putchar function which in analogous to getchar function can be used for writing characters one at a time to the output terminal. The general form is:

putchar (variable name); Where variable is a valid C type variable that has already been declared
Example: putchar ( );

Displays the value stored in variable C to the standard screen.

Program shows the use of getchar function in an interactive environment.

```
#include < stdio.h >                          // Inserts stdio.h header file into the Pgm
void main ( )                                // Beginning of main function.
{
        char in;                             // character declaration of variable in.
        printf (" please enter one character");   // message to user
```

```
        in = getchar ( ) ;                    // assign the keyboard input value to in.
        putchar (in);                         // output 'in' value to standard screen.
}
```

String input and output:

The gets function relieves the string from standard input device while put S outputs the string to the standard output device. A strong is an array or set of characters.

The function gets accepts the name of the string as a parameter, and fills the string with characters that are input from the keyboard till newline character is encountered. (That is till we press the enter key). All the end function gets appends a null terminator as must be done to any string and returns.

The puts function displays the contents stored in its parameter on the standard screen.

The standard form of the gets function is

gets (str)

Here str is a string variable.

The standard form for the puts character is

puts (str)

Where str is a string variable.

Example program (Involving both gets and puts)

```
# include < stdio.h >
Void main ( )
{
        char s [80];
        printf ("Type a string less than 80 characters:");
        gets (s);
        printf ("The string types is:");
        puts(s);
}
```

Formatted Input For Scanf:

The formatted input refers to input data that has been arranged in a particular format. Input values are generally taken by using the scanf function. The scanf function has the general form.

Scanf ("control string", arg1, arg2, arg3 ……argn);

The format field is specified by the control string and the arguments
arg1, arg2, …….argn specifies the address of location where address is to be stored.

The control string specifies the field format which includes format specifications and optional number specifying field width and the conversion character % and also blanks tabs and newlines.

The Blanks tabs and newlines are ignored by compiler. The conversion character % is followed by the type of data that is to be assigned to variable of the assignment. The field width specifier is optional.

The general format for reading a integer number is

% x d

Here percent sign (%) denotes that a specifier for conversion follows and x is an integer number which specifies the width of the field of the number that is being read. The data type character d indicates that the number should be read in integer mode.

Example:

scanf (“%d %d”, &sum1, &sum2);

If the values input are 175 and 1342 here value 175 is assigned to sum1 and 1342 to sum 2. Suppose the input data was follows 1342 and 175.

The number 134 will be assigned to sum1 and sum2 has the value 2 because of %3d the number 1342 will be cut to 134 and the remaining part is assigned to second variable sum2. If floating point numbers are assigned then the decimal or fractional part is skipped by the computer.

To read the long integer data type we can use conversion specifier % ld & % hd for short integer.

Input specifications for real number:

Field specifications are not to be use while representing a real number therefore real numbers are specified in a straight forward manner using % f specifier.

The general format of specifying a real number input is

Scanf (% f “, &variable);

Example: Scanf (“%f %f % f”, &a, &b, &c);

with the input data

321.76, 4.321, 678 The values 321.76 is assigned to a , 4.321 to b & 678 to C.

If the number input is a double data type then the format specifier should be % lf instead of %f.

Input specifications for a character:

Single character or strings can be input by using the character specifiers. The general format is

% xc or %xs

Where c and s represents character and string respectively and x represents the field width. The address operator need not be specified while we input strings.

Example:

Scanf (“%C %15C”, &ch, name):

Here suppose the input given is “a, Robert” then “a” is assigned to “ch” and “name” will be assigned to “Robert”.

Printing One Line:

printf();

The most simple output statement can be produced in C’ Language by using printf statement. It allows you to display information required to the user and also prints the variables we can also format the output and provide text labels. The simple statement such as

Printf (“Enter 2 numbers”);

Prompts the message enclosed in the quotation to be displayed.

A simple program to illustrate the use of printf statement:-

```
#include < stdio.h >
main ( )
{
        printf (“Hello!”);
        printf (“Welcome to the world of Engineering!”);
}
```

Output:

Hello! Welcome to the world of Engineering.

Both the messages appear in the output as if a single statement. If you wish to print the second message to the beginning of next line, a new line character must be placed inside the quotation marks.

For Example: printf (“Hello!\n);
OR
printf (“\n Welcome to the world of Engineering”);

Conversion Strings and Specifiers:

The printf ( ) function is quite flexible. It allows a variable number of arguments, labels and sophisticated formatting of output. The general form of the printf ( ) function is

Syntax: Printf (“conversion string”, variable list);

The conversion string includes all the text labels, escape character and conversion specifiers required for the desired output. The variable includes the entire variable to be printed in order they are to be printed. There must be a conversion specifies after each variable.

Specifier Meaning

%c – Print a character
%d – Print an Integer
%i – Print an Integer
%e – Print float value in exponential form.
%f – Print float value
%g – Print using %e or %f whichever is smaller
%o – Print octal value
%s – Print a string
%x – Print a hexadecimal integer (Unsigned) using lower case a – f
%X – Print a hexadecimal integer (Unsigned) using upper case A – F
%a – Print a unsigned integer.
%p – Print a pointer value
%hx – Print hex short
%lo – Print an octal long
%ld – Print a long integer

C Programming: Decision Making

Branching

The C language programs presented until now follows a sequential form of execution of statements. Many times it is required to alter the flow of the sequence of instructions. C language provides statements that can alter the flow of a sequence of instructions. These statements are called control statements. These statements help to jump from one part of the program to another. The control transfer may be conditional or unconditional.

if Statement:

The simplest form of the control statement is the If statement. It is very frequently used in decision making and allowing the flow of program execution.
The if structure has the following syntax

```
if (condition)
        statement;
```

The statement is any valid C' language statement and the condition is any valid C' language expression, frequently logical operators are used in the condition statement. The condition part should not end with a semicolon, since the condition and statement should be put together as a single statement. The command says if the condition is true then performs the following statement or If the condition is fake the computer skips the statement and moves on to the next instruction in the program.

Example program

```
# include <stdio.h>                    //Include the stdio.h file
void main ()                           // start of the program
{
    int numbers;                       // declare the variables
    printf ("Type a number:")          // message to the user
    scanf ("%d", &number);             // read the number from standard input
    if (number < 0)                    // check whether the number is a negative number
        number = -number;              // if it is negative then convert it into positive
    printf ("The absolute value is %d \n", number);          // print the value
}
```

The above program checks the value of the input number to see if it is less than zero. If it is then the following program statement which negates the value of the number is executed. If the value of the number is not less than zero, we do not want to negate it then this statement is automatically skipped. The absolute number is then displayed by the program, and program execution ends.

The If else construct:

The syntax of the If else construct is as follows:-

The if else is actually just on extension of the general format of if statement. If the result of the condition is true, then program statement 1 is executed, otherwise program statement 2 will be executed. If any case either program statement 1 is executed or program statement 2 is executed but not both when writing programs this else statement is so frequently required that almost all programming languages provide a special construct to handle this situation.

```
#include <stdio.h>                     //include the stdio.h header file in your program
```

```
void main ()                                    // start of the main
{
        int num;                                // declare variable num as integer
        printf ("Enter the number");            // message to the user
        scanf ("%d", &num);                     // read the input number from keyboard
        if (num < 0)                            // check whether number is less than zero
                printf ("The number is negative");   // if it is less than zero then it is negative
        else                                    // else statement
                printf ("The number is positive")  // if it is more than zero then it is positive
}
```

In the above program the If statement checks whether the given number is less than 0. If it is less than zero then it is negative therefore the condition becomes true then the statement The number is negative is executed. If the number is not less than zero the If else construct skips the first statement and prints the second statement declaring that the number is positive.

Compound Relational tests:

C language provides the mechanisms necessary to perform compound relational tests. A compound relational test is simple one or more simple relational tests joined together by either the logical AND or the logical OR operators. These operators are represented by the character pairs && // respectively. The compound operators can be used to form complex expressions in C.

Syntax

a) if (condition1 && condition2 && condition3)
b) if (condition1 || condition2 || condition3)

The syntax in the statement 'a' represents a complex if statement which combines different conditions using the and operator in this case if all the conditions are true only then the whole statement is considered to be true. Even if one condition is false the whole if statement is considered to be false.

The statement 'b' uses the logical operator or (||) to group different expression to be checked. In this case if any one of the expression if found to be true the whole expression considered to be true, we can also uses the mixed expressions using logical operators and and or together.

Nested if Statement:

The if statement may itself contain another if statement is known as nested if statement.

***Syntax*:**

```
if (condition1)
        if (condition2)
                statement-1;
  else
                statement-2;
  else
                 statement-3;
```

if statement may be nested as deeply as you need to nest it. One block of code will only be executed if two conditions are true. Condition 1 is tested first and then condition 2 is tested. The second if condition is nested in the first. The second if condition is tested only when the first condition is true else the program flow will skip to the corresponding else statement.

```
#include <stdio.h>                              //Includes stdio.h file to your program
```

```
Void main ()                                        // start of the program
{
     int year, rem_4, rem_100, rem_400              // variable declaration
     printf ("Enter the year to be tested")         // message for user
     scanf ("%d", &year);                           // Read the year from standard input.
      rem_4 = year % 4                              //find the remainder of year by 4
     rem_100 = year % 100                           //find the remainder of year by 100
      rem_400 = year % 400                          //find the remainder of year by 400
      if ((rem_4 == 0 && rem_100 != 0) rem_400 == 0)        // check whether remainder is zero
              printf ("It is a leap year. \n");              // print true condition
      else
              printf ("No. It is not a leap year. \n");      //print the false condition
 }
```

The above program checks whether the given year is a leap year or not. The year given is divided by 4,100 and 400 respectively and its remainder is collected in the variables rem_4, rem_100 and rem_400. A if condition statements checks whether the remainders are zero. If remainder is zero then the year is a leap year. Here either the year – y 400 is to be zero or both the year – 4 and year – by 100 has to be zero, then the year is a leap year.

The ELSE If Ladder:

When a series of many conditions have to be checked we may use the ladder else if statement which takes the following general form.

```
if (condition1)
        statement – 1;
else if (condition2)
        statement2;
else if (condition3)
        statement3;
else if (condition n)
        statement n;
else
         default statement;
statement-x;
```

This construct is known as if else construct or ladder. The conditions are evaluated from the top of the ladder to downwards. As soon on the true condition is found, the statement associated with it is executed and the control is transferred to the statement – x (skipping the rest of the ladder. When all the condition becomes false, the final else containing the default statement will be executed.

/* Example program using If else ladder to grade the student according to the following rules.

Marks	**Grade**
70 to 100	DISTINCTION
60 to 69	IST CLASS
50 to 59	IIND CLASS
40 to 49	PASS CLASS
0 to 39	FAIL

```
#include <stdio.h>              //include the standard stdio.h header file
```

```
void main ()                    //start the function main
{
        int marks;                          //variable declaration
        printf ("Enter marks\n");           //message to the user
        scanf ("%d", &marks);               //read and store the input marks.

        if (marks <= 100 && marks >= 70) //check whether marks is less than 100 or greater than 70
                printf ("\n Distinction");          //print Distinction if condition is True
        else if (marks >= 60)                       //else if the previous condition fails Check
                printf("\n First class");           //whether marks is > 60 if true print Statement
        else if (marks >= 50)                       //else if marks is greater than 50 print
                printf ("\n second class");         //Second class
        else if (marks >= 35)                       //else if marks is greater than 35 print
                printf ("\n pass class");           //pass class
        else
                printf ("Fail");                    //If all condition fail apply default condition print Fail
}
```

The above program checks a series of conditions. The program begins from the first if statement and then checks the series of conditions it stops the execution of remaining if statements whenever a condition becomes true.

In the first If condition statement it checks whether the input value is lesser than 100 and greater than 70. If both conditions are true it prints distinction. Instead if the condition fails then the program control is transferred to the next if statement through the else statement and now it checks whether the next condition given is whether the marks value is greater than 60 If the condition is true it prints first class and comes out of the If else chain to the end of the program on the other hand if this condition also fails the control is transferred to next if statements program execution continues till the end of the loop and executes the default else statement fails and stops the program.

The Switch Statement

Unlike the If statement which allows a selection of two alternatives the switch statement allows a program to select one statement for execution out of a set of alternatives. During the execution of the switch statement only one of the possible statements will be executed the remaining statements will be skipped. The usage of multiple If else statement increases the complexity of the program since when the number of If else statements increase it affects the readability of the program and makes it difficult to follow the program.
The switch statement removes these disadvantages by using a simple and straight forward approach.

The general format of the Switch Statement is

```
Switch (expression)
{
        Case case-label-1;
        Case case-label-2;
        Case case-label-n;
        ……
        Case default
}
```

When the switch statement is executed the control expression is evaluated first and the value is compared with the case label values in the given order. If the label matches with the value of the expression then the control is transferred directly to the group of statements which follow the label. If none of the statements matches then the statement against the default is executed. The default statement is optional in switch statement in case if any default statement is not given and if none of the condition

matches then no action takes place in this case the control transfers to the next statement of the if else statement.

Example

```
#include <stdio.h>
void main ()
{
  int num1, num2, result;
  char operator;

  printf ("Enter two numbers");
  scanf ("%d %d", &num1, &num2);
  printf ("Enter an operator");
  scanf ("%c", &operator);

  switch (operator)
        {
              case '+':
                          result = num1 + num2;
                          break;
               case '-':
                          result = num1 - num2;
                           break;
              case '*':
                           result = num1 * num2;
                          break;
               case '/':
                           if (num2 != 0)
                                 result = num1 / num2;
                           else
                          {
                                 printf ("warning : division by zero \n");
                                 result = 0;
                           }
                           break;
              default:
                          printf ("\n unknown operator");
                          result = 0;
                          break;
        }
  printf ("%d", result);
}
```

In the above program the break statement is need after the case statement to break out of the loop and prevent the program from executing other cases.

The GOTO statement

The goto statement is simple statement used to transfer the program control unconditionally from one statement to another statement. Although it might not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto is desirable.

Syntax

a>	b>
goto label;	label:
……	……
……	……
……	……
Label:	goto label;
Statement;	

The goto requires a label in order to identify the place where the branch is to be made. A label is a valid variable name followed by a colon.

The label is placed immediately before the statement where the control is to be transformed. A program may contain several goto statements that transferred to the same place when a program. The label must be unique. Control can be transferred out of or within a compound statement, and control can be transferred to the beginning of a compound statement. However the control cannot be transferred into a compound statement. The goto statement is discouraged in C, because it alters the sequential flow of logic that is the characteristic of C language.

Sample Code

```
1.  #include <stdio.h>          //include stdio.h header file to your program
2.  main ()                     //start of main
3.  {
4.      int n, sum = 0, i = 0;              // variable declaration
5.      printf ("Enter a number");                  // message to the user
6.      scanf ("%d", &n);                   //Read and store the number
7.      loop: i++;                           //Label of goto statement
8.      sum += i;                           //the sum value in stored and I is added to sum
9.      if (i < n) goto loop                //If value of I is less than n pass control to loop
10.     printf ("\n sum of %d natural numbers = %d", n, sum);           //print the sum
11. }
```

Looping

During looping a set of statements are executed until some conditions for termination of the loop is encountered. A program loop therefore consists of two segments one known as body of the loop and other is the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

In looping process in general would include the following four steps

1. Setting and initialization of a counter
2. Exertion of the statements in the loop
3. Test for a specified conditions for the execution of the loop
4. Incrementing the counter

The test may be either to determine whether the loop has repeated the specified number of times or to determine whether the particular condition has been met.

The While Statement:

The simplest of all looping structure in C is the while statement. The general format of the while statement is:

```
while (test condition)
{
        body of the loop
}
```

Here the given test condition is evaluated and if the condition is true then the body of the loop is executed. After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statements immediately after the body of the loop. The body of the loop may have one or more statements. The braces are needed only if the body contained two are more statements

Example program for generating 'N' Natural numbers using while loop:

```
# include < stdio.h >              //include the stdio.h file
void main()                        // Start of your program
{
        int n, i=0;                //Declare and initialize the variables
        printf("Enter the upper limit number");        //Message to the user
        scanf("%d", &n);                               //read and store the number
        while(i < = n)                                 // While statement with condition
        {                                              // Body of the loop
                printf("\t%d",i);                      // print the value of i
                i++;                                   increment i to the next natural number.
        }
}
```

In the above program the looping concept is used to generate n natural numbers. Here n and I are declared as integer variables and I is initialized to value zero. A message is given to the user to enter the natural number till where he wants to generate the numbers. The entered number is read and stored by the scanf statement. The while loop then checks whether the value of I is less than n i.e., the user entered number if it is true then the control enters the loop body and prints the value of I using the printf statement and increments the value of I to the next natural number this process repeats till the value of I becomes equal to or greater than the number given by the user.

The Do while statement

The do while loop is also a kind of loop, which is similar to the while loop in contrast to while loop, the do while loop tests at the bottom of the loop after executing the body of the loop. Since the body of the loop is executed first and then the loop condition is checked we can be assured that the body of the loop is executed at least once.

The syntax of the do while loop is:

```
do
{
        statement;
}
while(expression);
```

Here the statement is executed, then expression is evaluated. If the condition expression is true then the body is executed again and this process continues till the conditional expression becomes false. When the expression becomes false, the loop terminates.

To realize the usefulness of the do while construct consider the following problem. The user must be prompted to press Y or N. In reality the user can press any key other than y or n. IN such case the

message must be shown again and the user should be allowed to enter one of the two keys, clearly this is a loop construct. Also it has to be executed at least once. The following program illustrates the solution.

```
/* Program to illustrate the do while loop*/
#include < stdio.h >                          //include stdio.h file to your program
void main()                                   // start of your program
{
        char inchar;                           // declaration of the character

        do                                    // start of the do loop
        {
                printf("Input Y or N");               //message for the user
                scanf("%c", &inchar);                 // read and store the character
         }
        while(inchar!='y' && inchar != 'n');          //while loop ends
        if(inchar=='y')                               // checks whther entered character is y
                printf("you pressed u\n");            // message for the user
        else
                printf("You pressed n\n");
}                                                     //end of for loop
```

The Break Statement

Sometimes while executing a loop it becomes desirable to skip a part of the loop or quit the loop as soon as certain condition occurs, for example consider searching a particular number in a set of 100 numbers as soon as the search number is found it is desirable to terminate the loop. C language permits a jump from one statement to another within a loop as well as to jump out of the loop. The break statement allows us to accomplish this task. A break statement provides an early exit from for, while, do and switch constructs. A break causes the innermost enclosing loop or switch to be exited immediately.

Example program to illustrate the use of break statement is

```
/* A program to find the average of the marks*/
#include < stdio.h >                    //include the stdio.h file to your program
void main()                             // Start of the program
{
        int i, num=0;                   //declare the variables and initialize
        float sum=0,average;            //declare the variables and initialize
        printf("Input the marks, 1 to end\n");        // Message to the user
        while(1)                                      // While loop starts
        {
                scanf("%d",&i);                       // read and store the input number
                if(i=1)                               // check whether input number is -1
                        break;                        //if number –1 is input skip the loop
                sum+=i;                               //else add the value of I to sum
                num++                                 // increment num value by 1
        }
}                                                     //end of the program
```

Continue statement:

During loop operations it may be necessary to skip a part of the body of the loop under certain conditions. Like the break statement C supports similar statement called continue statement. The

continue statement causes the loop to be continued with the next iteration after skipping any statement in between. The continue with the next iteration the format of the continue statement is simply:

Continue;

Consider the following program that finds the sum of five positive integers. If a negative number is entered, the sum is not performed since the remaining part of the loop is skipped using continue statement.

```
#include < stdio.h >                    //Include stdio.h file
void main()                             //start of the program
{
        int i=1, num, sum=0;            // declare and initialize the variables
        for (i = 0; i < 5; i++)         // for loop
        {
                printf("Enter the integer");            //Message to the user
                scanf("%i", &num);                      //read and store the number
                if(num < 0)                     //check whether the number is less than zero
                {
                        printf("You have entered a negative number");   // message to the user
                }                                                       // end of for loop
                sum+=num;                                       // add and store sum to num
        }
        printf("The sum of positive numbers entered = %d",sum);         // print thte sum.
}                                                                       // end of the program.
```

For Loop

The for loop provides a more concise loop control structure. The general form of the for loop is:

```
for (initialization; test condition; increment)
{
        body of the loop
 }
```

When the control enters for loop the variables used in for loop is initialized with the starting value such as I=0, count=0. The value which was initialized is then checked with the given test condition. The test condition is a relational expression, such as I < 5 that checks whether the given condition is satisfied or not if the given condition is satisfied the control enters the body of the loop or else it will exit the loop. The body of the loop is entered only if the test condition is satisfied and after the completion of the execution of the loop the control is transferred back to the increment part of the loop. The control variable is incremented using an assignment statement such as I=I+1 or simply I++ and the new value of the control variable is again tested to check whether it satisfies the loop condition. If the value of the control variable satisfies then the body of the loop is again executed. The process goes on till the control variable fails to satisfy the condition.

Additional features of the for loop:

We can include multiple expressions in any of the fields of for loop provided that we separate such expressions by commas. For example in the for statement that begins

for (i = 0; j = 0; i < 10, j=j-10)

Sets up two index variables i and j the former initialized to zero and the latter to 100 before the loop begins. Each time after the body of the loop is executed, the value of i will be incremented by 1 while the value of j is decremented by 10.

Just as the need may arise to include more than one expression in a particular field of the for statement, so too may the need arise to omit on or more fields from the for statement. This can be done simply by omitting the desired filed, but by marking its place with a semicolon. The init_expression field can simply be "left blank" in such a case as long as the semicolon is still included:

for (j!=100;++j)

The above statement might be used if j were already set to some initial value before the loop was entered. A for loop that has its looping condition field omitted effectively sets up an infinite loop, that is a loop that theoretically will be executed for ever.

For loop example program:

```
/* The following is an example that finds the sum of the first fifteen positive natural numbers*/
#include < stdio.h >                          //Include stdio.h file
void main()                                  //start main program
{
        int i;                               //declare variable
        int sum=0,sum_of_squares=0;          //declare and initialize variable.
        for(i=0;i < = 30; i+=2)              //for loop
        {
                sum+=i;                      //add the value of i and store it to sum
                sum_of_squares+=i*i;         //find the square value and add it to sum_of_squares
        }                                    //end of for loop
        printf("Sum of first 15 positive even numbers=%d\n",sum);        //Print sum
        printf("Sum of their squares=%d\n",sum_of_squares);              //print sum_of_square
}
```

Arrays

The C language provides a capability that enables the user to define a set of ordered data items known as an array.
Suppose we had a set of grades that we wished to read into the computer and suppose we wished to perform some operations on these grades, we will quickly realize that we cannot perform such an operation until each and every grade has been entered since it would be quite a tedious task to declare each and every student grade as a variable especially since there may be a very large number.
In C we can define a variable called grade, which represents not a single value of grade but a entire set of grades. Each element of the set can then be referenced by means of a number called as index number or subscript.

Declaration of arrays

Like any other variable arrays must be declared before they are used. The general form of declaration is:

type variable-name[50];

The type specifies the type of the elements that will be contained in the array, such as int float or char and the size indicates the maximum number of elements that can be stored inside the array for ex:

float height[50];

Declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. In C the array elements index or subscript begins with number zero. So height [0] refers to the first element of the array. (For this reason, it is easier to think of it as referring to element number zero, rather than as referring to the first element)
.
As individual array element can be used anywhere that a normal variable with a statement such as

G = grade [50];

The statement assigns the value stored in the 50th index of the array to the variable g. More generally if i is declared to be an integer variable, then the statement g=grades [i]; Will take the value contained in the element number i of the grades array to assign it to g. so if I were equal to 7 when the above statement is executed, then the value of grades [7] would get assigned to g.
A value stored into an element in the array simply by specifying the array element on the left hand side of the equals sign. In the statement

grades [100]=95;

The value 95 is stored into the element number 100 of the grades array.
The ability to represent a collection of related data items by a single array enables us to develop concise and efficient programs. For example we can very easily sequence through the elements in the array by varying the value of the variable that is used as a subscript into the array. So the for loop

```
for(i=0;i < 100;++i);
sum = sum + grades [i];
```

Will sequence through the first 100 elements of the array grades (elements 0 to 99) and will add the values of each grade into sum. When the for loop is finished, the variable sum will then contain the total of first 100 values of the grades array (Assuming sum were set to zero before the loop was entered)
Just as variables arrays must also be declared before they are used. The declaration of an array involves the type of the element that will be contained in the array such as int, float, char as well as maximum

number of elements that will be stored inside the array. The C system needs this latter information in order to determine how much memory space to reserve for the particular array.

The declaration int values[10]; would reserve enough space for an array called values that could hold up to 10 integers. Refer to the below given picture to conceptualize the reserved storage space.

values[0]	
values[1]	
values[2]	
values[3]	
values[4]	
values[5]	
values[6]	
values[7]	
values[8]	
values[9]	

Initialization of arrays

We can initialize the elements in the array in the same way as the ordinary variables when they are declared. The general form of initialization off arrays is:

type array_name[size]={list of values};

The values in the list care separated by commas, for example the statement

int number[3]={0,0,0};

Will declare the array size as a array of size 3 and will assign zero to each element if the number of values in the list is less than the number of elements, then only that many elements are initialized. The remaining elements will be set to zero automatically.

In the declaration of an array the size may be omitted, in such cases the compiler allocates enough space for all initialized elements. For example the statement

int counter[]={1,1,1,1};

Will declare the array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Complete the given points

Accessing the array elements:
(i) Assigning values to array elements: .....
(ii) Displaying the array elements: .....

```
/* Program to count the no of positive and negative numbers*/
#include< stdio.h >
void main( )
{
        int a[50],n,count_neg=0,count_pos=0,i;
        printf("Enter the size of the array\n");
        scanf("%d",&n);
        printf("Enter the elements of the array\n");
        for (i=0;i < n;i++)
```

```
        scanf("%d",&a[I]);
        for(i=0;i < n;i++)
        {
                if(a[i] < 0)
                        count_neg++;
                else
                        count_pos++;

        }
        printf("There are %d negative numbers in the array\n",count_neg);
        printf("There are %d positive numbers in the array\n",count_pos);
}
```

Multi-dimensional Arrays

Often there is a need to store and manipulate two dimensional data structure such as matrices & tables. Here the array has two subscripts. One subscript denotes the row & the other the column. The declaration of two dimension arrays is as follows:

data_type array_name[row_size][column_size];
int m[10][20]

Here m is declared as a matrix having 10 rows (numbered from 0 to 9) and 20 columns (numbered 0 through 19). The first element of the matrix is m[0][0] and the last row last column is m[9][19]

Elements of multi dimension arrays

A 2 dimensional array marks [4][3] is shown below figure. The first element is given by marks [0][0] contains 35.5 & second element is marks [0][1] and contains 40.5 and so on.

marks [0][0] 35.5	Marks [0][1] 40.5	Marks [0][2] 45.5
marks [1][0] 50.5	Marks [1][1] 55.5	Marks [1][2] 60.5
marks [2][0]	Marks [2][1]	Marks [2][2]
marks [3][0]	Marks [3][1]	Marks [3][2]

Initialization of multidimensional arrays

Like the one dimension arrays, two dimension arrays may be initialized by following their declaration with a list of initial values enclosed in braces

Example: int table[2][3]={0,0,01,1,1};

Initializes the elements of first row to zero and second row to 1. The initialization is done row by row. The above statement can be equivalently written as

int table[2][3]={{0,0,0},{1,1,1}}

By surrounding the elements of each row by braces.

C allows arrays of three or more dimensions. The compiler determines the maximum number of dimension. The general form of a multidimensional array declaration is:

date_type array_name[s1][s2][s3].....[sn];

Where s is the size of the i^{th} dimension.

Some examples are:

int survey[3][5][12];
float table[5][4][5][3];

Survey is a 3 dimensional array declared to contain 180 integer elements. Similarly table is a four dimensional array containing 300 elements of floating point type.

Assignment:

Give an example program to add two matrices & store the results in the 3rd matrix.

Structure

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types we can use a structure. Structure is a method of grouping data of different types. A structure is a convenient method of handling a group of related data items of different data types.

Structure decleration:
General format:

```
struct tag_name
{
        data type member1;
        data type member2;
        …
        …
};
```

Example:

```
struct lib_books
{
        char title[20];
        char author[15];
        int pages;
        float price;
};
```

the keyword struct declares a structure to holds the details of four fields namely title, author pages and price. These are members of the structures. Each member may belong to different or same data type. The tag name can be used to define objects that have the tag names structure. The structure we just declared is not a variable by itself but a template for the structure.

We can declare structure variables using the tag name any where in the program. For example the statement,

struct lib_books book1,book2,book3;

declares book1,book2,book3 as variables of type struct lib_books each declaration has four elements of the structure lib_books. The complete structure declaration might look like this

```
struct lib_books
{
        char title[20];
        char author[15];
        int pages;
        float price;
};

struct lib_books, book1, book2, book3;
```

structures do not occupy any memory until it is associated with the structure variable such as book1. the template is terminated with a semicolon. While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template. The tag name such as lib_books can be used to declare structure variables of its data type later in the program.

We can also combine both template declaration and variables declaration in one statement, the declaration

```
struct lib_books
{
        char title[20];
        char author[15];
        int pages;
        float price;
} book1,book2,book3;
```

is valid. The use of tag name is optional for example

```
struct
{
        …
        …
        …
} book1, book2, book3;
```

declares book1,book2,book3 as structure variables representing 3 books but does not include a tag name for use in the declaration.

A structure is usually defines before main along with macro definitions. In such cases the structure assumes global status and all the functions can access the structure.

Giving values to members

As mentioned earlier the members themselves are not variables they should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' which is known as dot operator or period operator.

For example:

```
Book1.price
```

Is the variable representing the price of book1 and can be treated like any other ordinary variable. We can use scanf statement to assign values like

```
scanf("%s",book1.file);
scanf("%d",& book1.pages);
```

Or we can assign variables to the members of book1

```
strcpy(book1.title,"basic");
strcpy(book1.author,"Balagurusamy");
book1.pages=250;
book1.price=28.50;
```

Assignment to Students

Explain how to access array elements and to display those elements under two topics : (i) Accessing array elements (ii) displaying array elements.

Functions and structures

We can pass structures as arguments to functions. Unlike array names however, which always point to the start of the array, structure names are not pointers. As a result, when we change structure parameter inside a function, we don't affect its corresponding argument.

Passing structure to elements to functions

A structure may be passed into a function as individual member or a separate variable.
A program example to display the contents of a structure passing the individual elements to a function is shown below.

```
#include<stdio.h>
#include<conio.h>
void fun(float,int);
void main()
{
        struct student
        {
                float marks;
                int id;
        };
        struct student s1={67.5,14};
        fun(s1.marks,s1.id);
        getch();
}
void fun(float marks,int id)
{
         printf("\nMarks:%f",marks);
        printf("\nID:%d",id);
}
```

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

Passing entire structure to functions

In case of structures having to having numerous structure elements passing these individual elements would be a tedious task. In such cases we may pass whole structure to a function as shown below:

```
# include stdio.h>
{
        int emp_id;
        char name[25];
```

```
        char department[10];
        float salary;
};
void main()
{
        static struct employee emp1= { 12, "sadanand", "computer", 7500.00 };

        /*sending entire employee structure*/
        display(emp1);
}

/*function to pass entire structure variable*/
display(empf)
struct employee empf
{
        printf("%d%s,%s,%f", empf.empid,empf.name,empf.department,empf.salary);
}
```

Arrays of structure

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
        int id_no;
        char name[20]; char address[20];
        char combination[3]; int age;
}
student[100];
```

An array of structures can be assigned initial values just as any other array can. Remember that each element is a structure that must be assigned corresponding initial values as illustrated below.

```
#include< stdio.h >
void main()
{
        struct info
        {
                int id_no;
                char name[20];
                char address[20];
                char combination[3];
                int age;
        };
        struct info std[100];
        int i,n;
        printf("Enter the number of students");
```

```
        scanf("%d",&n);
        for(i=0;i < n;i++)
        {
                printf(" Enter Id_no,name address combination age\m");
                scanf("%d%s%s%s%d",&std[I].id_no,std[I].name,std[I].address,std[I].
                combination,&std[I].age);
        }
        printf("\n Student information");
        for (I=0;I< n;I++)
        {
                printf("%d%s%s%s%d\n",
                std[I].id_no,std[I].name,std[I].address,std[I].combination,std[I].age);
        }
}
```

Structure within a structure

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
structure date def;
structure date doa;
}oldstudent, newstudent;
```

the structure student contains another structure date as its one of its members.

Union:

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to observe memory. They are useful for application involving multiple members, where values need not be assigned to all the members at any one time. Like structures union can be declared using the keyword union as follows:

```
union item
{
        int m;
        float p;
        char c;
}
code;
```

This declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

```
code.m
code.p
code.c
```

are all valid member variables. During accessing we should make sure that we are accessing the member whose value is currently stored.

For example a statement such as

```
code.m=456;
code.p=456.78;
printf("%d",code.m);
```

More on Structure

typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example:

```
typedef struct
{
        char firstName[20];
         char lastName[20];
        char SSN[10];
         float gpa;
}student;
```

Now you can use student directly to define variables of student type without using struct keyword. Following is the example:

```
student student_a;
```

You can use typedef for non-structs:

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y and z are all pointers to long ints

Functions

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

There are many advantages in using functions in a program they are:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.

2. The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigation.

4. A function may be used by many other programs this means that a c programmer can build on what others have already done, instead of starting over from scratch.

5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.

6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

We already know that C support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.

Function definition

```
[ data type] function name (argument list)
argument declaration;
{
        local variable declarations;
        statements;
        [return expression]
}
```

Example :

```
mul(a,b)
int a,b;
{
        int y;
        y=a+b;
        return y;
}
```

When the value of y which is the addition of the values of a and b. the last two statements ie,

y=a+b; can be combined as
return(y)
return(a+b);

Types of functions

A function may belong to any one of the following categories:
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

Functions with no arguments and no return values

Let us consider the following program

```
/* Program to illustrate a function with no argument and no return values*/
#include
main()
{
        staetemtn1();
        starline();
        statement2();
        starline();
}
/*function to print a message*/
statement1()
{
        printf("\n Sample subprogram output");
}
statement2()
{
        printf("\n Sample subprogram output two");
}
starline()
{
        int a;
        for (a=1;a<60;a++)
        printf("%c",'*');
        printf("\n");
}
```

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. Similarly when it does not return value the calling function does not receive any data from the called function. A function that does not return any value cannot be used in an expression it can be used only as independent statement.

Functions with arguments but no return values

The nature of data communication between the calling function and the arguments to the called function and the called function does not return any values to the calling function this shown in example below:

Consider the following:

Function calls containing appropriate arguments. For example the function call value (500,0.12,5)

Would send the values 500,0.12 and 5 to the function value (p, r, n) and assign values 500 to p, 0.12 to r and 5 to n. the values 500,0.12 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

Both the arguments actual and formal should match in number type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below:

```
main()
{
        function1(a1,a2,a3……an)
}

function1(f1,f2,f3….fn);
{
        function body;
}
```

Here a1, a2, a3 are actual arguments and f1, f2, f3 are formal arguments.

The no of formal arguments and actual arguments must be matching to each other suppose if actual arguments are more than the formal arguments, the extra actual arguments are discarded. If the numbers of actual arguments are less than the formal arguments then the unmatched formal arguments are initialized to some garbage values. In both cases no error message will be generated.

The formal arguments may be valid variable names; the actual arguments may be variable names expressions or constants. The values used in actual arguments must be assigned values before the function call is made.

When a function call is made only a copy of the values actual arguments is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.

Let us consider the following program

```
/*Program to find the largest of two numbers using function*/
#include<stdio.h>
main()
{
        int a,b;
        printf("Enter the two numbers");
        scanf("%d%d",&a,&b);
        largest(a,b);
}
/*Function to find the largest of two numbers*/
largest(int a, int b)
{
        if(a>b)
                printf("Largest element=%d",a);
        else
                printf("Largest element=%d",b);
}
```

in the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function does not return any value.

Functions with arguments and return values

The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.
To assure a high degree of portability between programs a function should generally be coded without involving any input output operations. For example different programs may require different output formats for displaying the results. Theses shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

Recursion

Recursive function is a function that calls itself. When a function calls another function and that second function calls the third function then this kind of a function is called nesting of functions. But a recursive function is the function that calls itself repeatedly.

An example program to find out factorial of a given number using Recursion:

```
int fact(int);
void main()
{
        int n, fac;
        clrscr();
        printf("Enter the value of n");
        scanf("%d",&n);
        fac=fact(n);
        printf("The factorial of %d is %d",n,fac);
        getch();
}

int fact(int n)
{
        if(n= =1)
                return(1);
        else
                return fact(n-1)*n;
}
```

Functions and arrays

We can pass an entire array of values into a function just as we pass indiviual variables. In this task it is essential to list the name of the array along with functions arguments without any subscripts and the size of the array as arguments

For example: Largest(a,n);

will pass all the elements contained in the array a of size n. the called function expecting this call must be appropriately defined. The largest function header might look like:

```
float smallest(array,size);
float array[];
int size;
```

The function smallest is defined to take two arguments, the name of the array and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[ ];
```

The above declaration indicates to compiler that the arguments array is an array of numbers. It is not necessary to declare size of the array here. While dealing with array arguments we should remember one major distinction. If a function changes the value the value of an array elements then these changes will be made to the original array that passed to the function. When the entire array is passed as an argument, the contents of the array are not copied into the formal parameter array instead information about the address of the array elements are passed on to the function. Therefore any changes introduced to array elements are truly reflected in the original array in the calling function.

Global and local variables

A local variable is one that is declared inside a function and can only be used by that function. If you declare a variable outside all functions then it is a global variable and can be used by all functions in a program.

```
#include<stdio.h>

// Global variables
int a;
int b;
 int Add()
{
        return a + b;
}
 int main()
{
         int answer; // Local variable
         a = 5;
         b = 7;
         answer = Add();
         printf("%d\n",answer);
         return 0;
}
```

Call by Value and Call by reference

The arguments passed to function can be of two types namely
1. Values passed / Call by Value
2. Address passed / Call by reference
The first type refers to call by value and the second type refers to call by reference.

*For instance consider **program 1:***

```
main()
{
        int x=50, y=70;
        interchange(x,y);
        printf("x=%d y=%d",x,y);
}
interchange(x1,y1)
int x1,y1;
{
        int z1;
        z1=x1;
        x1=y1;
        y1=z1;
        printf("x1=%d y1=%d",x1,y1);
}
```

Here the value to function interchange is passed by value.

*Consider **program2***

```
main()
{
        int x=50, y=70;
        interchange(&x,&y);
        printf("x=%d y=%d",x,y);
}
interchange(x1,y1)
int *x1,*y1;
{
        int z1;
        z1=*x1;
        *x1=*y1;
        *y1=z1;
        printf("*x=%d *y=%d",x1,y1);
}
```

Here the function is called by reference. In other words address is passed by using symbol & and the value is accessed by using symbol *.
The main difference between them can be seen by analyzing the output of program1 and program2.

The output of program1 that is call by value is
x1=70 y1=50
x=50 y=70

But the output of program2 that is call by reference is

*x=70 *y=50
x=70 y=50

This is because in case of call by value the value is passed to function named as interchange and there the value got interchanged and got printed as

x1=70 y1=50

and again since no values are returned back and therefore original values of x and y as in main function namely

x=50 y=70 got printed.

Pointer

In c a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

Pointer declaration

A pointer is a variable that contains the memory location of another variable. The syntax is as shown below. You start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of the variable.
type * variable name

Example: int *ptr;
float *string;

Address operator

Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

ptr=#

This places the address where num is stores into the variable ptr. If num is stored in memory 21260 address then the variable ptr has the value 21260.

```
/* A program to illustrate pointer declaration*/
main()
{
        int *ptr;
        int sum;
        sum=45;
        ptr=∑
        printf ("\n Sum is %d\n", sum);
        printf ("\n The sum pointer is %d", ptr);
}
```

We will get the same result by assigning the address of num to a regular(non pointer) variable. The benefit is that we can also refer to the pointer variable as *ptr the asterisk tells to the computer that we are not interested in the value 21260 but in the value stored in that memory location. While the value of pointer is 21260 the value of sum is 45 however we can assign a value to the pointer * ptr as in *ptr=45. This means place the value 45 in the memory address pointer by the variable ptr.

Since the pointer contains the address 21260 the value 45 is placed in that memory location. And since this is the location of the variable num the value also becomes 45. this shows how we can change the value of pointer directly using a pointer and the indirection pointer.

/* Program to display the contents of the variable their address using pointer variable*/

```
include< stdio.h >
void main( )
{
        int num, *intptr;
        float x, *floptr;
        char ch, *cptr;
        num=123;
        x=12.34;
        ch='a';
        intptr=&x;
        cptr=&ch;
        floptr=&x;
        printf("Num %d stored at address %u\n",*intptr, intptr);
        printf("Value %f stored at address %u\n",*floptr, floptr);
        printf("Character %c stored at address %u\n",*cptr, cptr);
}
```

Pointer expressions & pointer arithmetic

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z= 5* - *p2/p1;
*p2= *p2 + 10;
```

C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc.

We can also compare pointers by using relational operators the expressions such as p1 >p2, p1==p2 and p1!=p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/

```
#include< stdio.h >
main()
{
        int ptr1,ptr2;
        int a,b,x,y,z;
        a=30;b=6;
        ptr1=&a;
        ptr2=&b;

        x=*ptr1+ *ptr2 –6;
        y=6*- *ptr1/ *ptr2 +30;

        printf("\nAddress of a +%u",ptr1);
        printf("\nAddress of b %u",ptr2);
        printf("\na=%d, b=%d",a,b);
        printf("\nx=%d,y=%d",x,y);
        ptr1=ptr1 + 70;
```

```
        ptr2= ptr2;
        printf("\na=%d, b=%d",a,b);
}
```

Pointers and function

The pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.
1. Call by reference
2. Call by value.

Call by value

We have seen that a function is invoked there will be a link established between the formal and actual parameters. A temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between actual and formal parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter represents a local variable in the called function. The current value of corresponding actual parameter becomes the initial value of formal parameter. The value of formal parameter may be changed in the body of the actual parameter. The value of formal parameter may be changed in the body of the subprogram by assignment or input statements. This will not change the value of actual parameters.

```
/* Include< stdio.h >
void main()
{
        int x,y;
        x=20;
        y=30;
        printf("\n Value of a and b before function call =%d %d",a,b);
        fncn(x,y);
        printf("\n Value of a and b after function call =%d %d",a,b);
}

fncn(p,q)
int p,q;
{
        p=p+p;
        q=q+q;
}
```

Call by Reference

When we pass address to a function the parameters receiving the address should be pointers. The process of calling a function by using pointers to pass the address of the variable is known as call by reference. The function which is called by reference can change the values of the variable used in the call.

```
/* example of call by reference*/

include< stdio.h >
void main()
{
```

```
        int x,y;
        x=20;
        y=30;
        printf("\n Value of a and b before function call =%d %d",a,b);
        fncn(&x,&y);
        printf("\n Value of a and b after function call =%d %d",a,b);
}

fncn(p,q)
int p,q;
{
        *p=*p+*p;
        *q=*q+*q;
}
```

Pointer to arrays

An array is actually very much like pointer. We can declare the arrays first element as a[0] or as int *a because a[0] is an address and *a is also an address the form of declaration is equivalent. The difference is pointer is a variable and can appear on the left of the assignment operator that is lvalue. The array name is constant and cannot appear as the left side of assignment operator.

```
/* A program to display the contents of array using pointer*/
void main()
{
        int a[100];
        int i,j,n;

        printf("\nEnter the elements of the array\n");
        scanf("%d",&n);
        printf("Enter the array elements");

        for(I=0;I< n;I++)
        {
                scanf("%d",&a[I]);
                printf("Array element are");
        }

        for(ptr=a, ptr< (a+n); ptr++)
        printf("Value of a[%d]=%d stored at address %u",j+=,*ptr,ptr);
}
```

Strings are characters arrays and here last element is \0 arrays and pointers to char arrays can be used to perform a number of string functions.

Pointers and structures

We know the name of an array stands for the address of its zeroth element the same concept applies for names of arrays of structures. Suppose item is an array variable of struct type. Consider the following declaration:

```
        struct products
        {
                char name[30];
                int manufac;
```

```
        float net;
        item[2],*ptr;
}
```

this statement declares item as array of two elements, each type struct products and ptr as a pointer data objects of type struct products, the

assignment ptr=item;

would assign the address of zeroth element to product[0]. Its members can be accessed by using the following notation.

```
ptr- >name;
ptr- >manufac;
ptr- >net;
```

The symbol - > is called arrow pointer and is made up of minus sign and greater than sign. Note that ptr- > is simple another way of writing product[0].

When the pointer is incremented by one it is made to pint to next record ie item[1]. The following statement will print the values of members of all the elements of the product array.

```
for(ptr=item; ptr< item+2;ptr++)
printf("%s%d%f\n",ptr- >name,ptr- >manufac,ptr- >net);
```

We could also use the notation

(*ptr).number

to access the member number. The parenthesis around ptr are necessary because the member operator '.' Has a higher precedence than the operator *.

Dynamic memory allocation

In programming we may come across situations where we may have to deal with data, which is dynamic in nature. The number of data items may change during the executions of a program. The number of customers in a queue can increase or decrease during the process at any time. When the list grows we need to allocate more memory space to accommodate additional data items. Such situations can be handled move easily by using dynamic techniques. Dynamic data items at run time, thus optimizing file usage of storage space.

The process of allocating memory at run time is known as dynamic memory allocation. Although c does not inherently have this facility there are four library routines which allow this function.

Many languages permit a programmer to specify an array size at run time. Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program. But c inherently does not have this facility but supports with memory management functions, which can be used to allocate and free memory during the program execution. The following functions are used in c for purpose of memory management.

Function	**Task**
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space.

Memory allocations process

According to the conceptual view the program instructions and global and static variable in a permanent storage area and local area variables are stored in stacks. The memory space that is located between these two regions in available for dynamic allocation during the execution of the program. The free memory region is called the heap. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a null pointer.

Allocating a block of memory

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast_type*)malloc(byte_size);
```

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

Example

```
x=(int*)malloc(100*sizeof(int));
```

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int

Allocating multiple blocks of memory

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of calloc is:

```
ptr=(cast_ype*) calloc(n,elem_size);
```

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

Releasing the used space

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

```
free(ptr);
```

ptr is a pointer that has been created by using malloc or calloc.

To alter the size of allocated memory

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is :

```
ptr=realloc(ptr,newsize);
```

This function allocates new memory space of size newsize to the pointer variable ptr ans returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.

/*Example program for reallocation*/

```
#include< stdio.h >
#include< stdlib.h >
define NULL 0

main()
{
        char *buffer;

        /*Allocating memory*/

        if((buffer=(char *) malloc(10))==NULL)
        {
                printf("Malloc failed\n");
                exit(1);
        }
        printf("Buffer of size %d created \n,_msize(buffer));
```

```
        strcpy(buffer,"Kathmandu");
        printf("\nBuffer contains:%s\n",buffer);

        /*Reallocation*/
        if((buffer=(char *)realloc(buffer,15))==NULL)
        {
                printf("Reallocation failed\n");
                exit(1);
        }
printf("\nBuffer size modified.\n");
printf("\nBuffer still contains: %s\n",buffer);
strcpy(buffer,"Hastinapur");
printf("\nBuffer now contains:%s\n",buffer);

/*freeing memory*/
free(buffer);
}
```

File Handling in C

C supports a number of functions that have the ability to perform basic file operations, which include:
1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

File operation functions in C

Function Name	Operation
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the beginning of the file

Defining and opening a file

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

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

```
FILE *fp;
fp=fopen("filename","mode");
```

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

w	open for writing (file need not exist)
a	open for appending (file need not exist)
r+	open for reading and writing, start at beginning
w+	open for reading and writing (overwrite file)
a+	open for reading and writing (append if file exists)

Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

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

Closing a file

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

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

Observe the following program.

```
….
FILE *p1 *p2;
p1=fopen (“Input”,”w”);
p2=fopen (“Output”,”r”);
….
…
fclose(p1);
fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include< stdio.h >
main()
{
        file *f1;
        printf(“Data input output”);
        f1=fopen(“Input”,”w”);                  /*Open the file Input*/
        while((c=getchar())!=EOF)               /*get a character from key board*/
        putc(c,f1);                             /*write a character to input*/
        fclose(f1);                            /*close the file input*/
        printf(“\nData output\n”);
        f1=fopen(“INPUT”,”r”);                  /*Reopen the file input*/
        while((c=getc(f1))!=EOF)
        printf(“%c”,c);
        fclose(f1);
}
```

The getw and putw functions

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

putw(integer,fp);
getw(fp);

```
/*Example program for using getw and putw functions*/
#include< stdio.h >
main()
{
        FILE *f1,*f2,*f3;
        int number I;
        printf("Contents of the data file\n\n");
        f1=fopen("DATA","W");
        for(I=1;I< 30;I++)
        {
                scanf("%d",&number);
                if(number==-1)
                break;
                putw(number,f1);
        }

        fclose(f1);
        f1=fopen("DATA","r");
        f2=fopen("ODD","w");
        f3=fopen("EVEN","w");
        while((number=getw(f1))!=EOF)                   /* Read from data file*/
        {
                if(number%2==0)
                putw(number,f3);                                /*Write to even file*/
                else
                putw(number,f2);                                /*write to odd file*/
        }

        fclose(f1);
        fclose(f2);
        fclose(f3);
        f2=fopen("ODD","r");
        f3=fopen("EVEN","r");
        printf("\n\nContents of the odd file\n\n");
        while(number=getw(f2))!=EOF)
        printf("%d%d",number);
        printf("\n\nContents of the even file");
        while(number=getw(f3))!=EOF)
        printf("%d",number);
        fclose(f2);
        fclose(f3);
}
```

The fprintf & fscanf functions

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

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f",name,age,7.5);

Here name is an array variable of type char and age is an int variable. The general format of fscanf is

fscanf(fp,"controlstring",list);

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

Example

fscanf(f2,"5s%d",item,&quantity");

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

```
/*Program to handle mixed data types*/
#include< stdio.h >
main()
{
        FILE *fp;
        int num,qty,I;
        float price,value;
        char item[10],filename[10];
        printf("Input filename");
        scanf("%s",filename);
        fp=fopen(filename,"w");
        printf("Input inventory data\n\n"0;
        printf("Item namem number price quantity\n");
        for I=1;I< =3;I++)
        {
                fscanf(stdin,"%s%d%f%d",item,&number,&price,&quality);
                fprintf(fp,"%s%d%f%d",itemnumber,price,quality);
        }
        fclose (fp);
        fprintf(stdout,"\n\n");
        fp=fopen(filename,"r");
        printf("Item name number price quantity value");

        for(I=1;I< =3;I++)
        {
                fscanf(fp,"%s%d%f%d",item,&number,&prince,&quality);
                value=price*quantity");
                fprintf("stdout,"%s%d%f%d%d\n",item,number,price,quantity,value);
        }
        fclose(fp);
}
```

C Graphics

If all pictures are built by concept of pixel then wondering how each picture differ that is how some picture appear more brighter while some other have a shady effect. All this is by the concept or technically terminology called as resolution.

So let's have an insight on this important terminology

.

Resolution is the number of rows that appear from top to bottom of a screen and in turn the number of pixels or pixel elements that appear from left to right on each scan line. Based on this resolution only the effect of picture appears on screen. In other words greater the resolution greater will be the clarity of picture. This is because greater the number of dots greater will be sharpness of picture. That is resolution value is directly proportional to clarity of picture

.

There are generally two modes available namely text and graphics. In a graphics mode we have generally the following adapters namely CGA called as Color Graphics Adapter, EGA and VGA. Each adapter differs in the way of generating colors and also in the number of colors produced by each adapter. Pixel being a picture element when we consider the graphics mode each pixel has a color associated with it. But the way these colors are used depends on adapters because each adapter differs in the way they handle colors and also in the number of colors supported.

Having known about adapters now let us start knowing on how to start switching to graphics mode from text mode in other words how to start using pixel and resolution concepts.

This is done by a function called intigraph ( ).This intigraph ( ) takes in it 2 main arguments as input namely gd and gm.

In this gd has the number of mode which has the best resolution. This is very vital for graphics since the best resolution only gives a sharper picture as we have seen before. This value is obtained by using the function called as getgraphmode ( ) in C graphics. The other argument gm gives insight about the monitor used, the corresponding resolution of that, the colors that are available since this varies based on adapters supported. This value is obtained by using the function named as getmodename ( ) in C graphics.

Graphics function in C and Pixel concept

There are numerous graphics functions available in c. But let us see some to have an understanding of how and where a pixel is placed in each when each of the graphics function gets invoked.

Function:

putpixel(x, y, color)

Purpose:

The functionality of this function is it put a pixel or in other words a dot at position x, y given in inputted argument. Here one must understand that the whole screen is imagined as a graph. In other words the pixel at the top left hand corner of the screen represents the value (0, 0).

Here the color is the integer value associated with colors and when specified the picture element or the dot is placed with the appropriate color associated with that integer value.

Function:

Line(x1, y1, x2, y2)

Purpose: The functionality of this function is to draw a line from (x1,y1) to (x2,y2).Here also the coordinates are passed taking the pixel (0,0) at the top left hand corner of the screen as the origin. And also one must note that the line formed is by using number of pixels placed near each other.

Function:

getpixel(x, y)

Purpose:
This function when invoked gets the color of the pixel specified. The color got will be the integer value associated with that color and hence the function gets an integer value as return value.

So the smallest element on the graphics display screen is a pixel or a dot and the pixels are used in this way to place images in graphics screen in C language.

Some Examples of C graphics:

```
// Drawing a Circle
        #include<graphics.h>
        void main()
        {
                int gd= DETECT, gm;
                initgraph(&gd,&gm,"c:\\tc\\bgi");
                circle(200,100,10);
                setcolor(WHITE);
                 getch();
                closegraph();

         }

//Drawing Lines
        #include<graphics.h>
        void main()
        {
                int gd= DETECT, gm;
                initgraph(&gd,&gm,"c:\\tc\\bgi");
                line(90,70,60,100);
                line(200,100,150,300);
                setcolor(WHITE);
                getch();
                closegraph();

         }

// Drawing rectangle
        #include<graphics.h>
        void main()
        {
                int gd= DETECT, gm;
                initgraph(&gd,&gm,"c:\\tc\\bgi");
                rectangle(200,100,150,300);
                setcolor(WHITE);
                getch();
                closegraph();
         }
```

```
//Constructing Triangle
        #include<graphics.h>
        void main()
        {
                int gd= DETECT, gm;
                initgraph(&gd,&gm,"c:\\tc\\bgi");
                line(200,100,10,20);line(10,20,50,60);line(50,60,200,100);
                setcolor(WHITE);
                getch();
                 closegraph();

         }
```