LECTURE NOTES
ON

Computer Programming Using C
21CS01101

B.Tech. 1st Semester
(Computer Science and Engineering)

Prepared By

Dixita B. Kagathara

Assistant Professor

dixita.kagathara@darshan.ac.in

Darshan UNIVERSITY

योगः कर्मसु कौशलम्

Table of Content

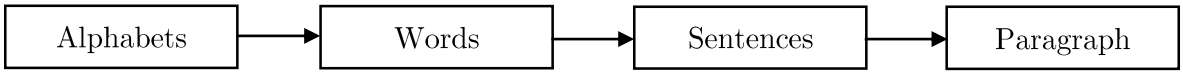
Unit-1

Overview of C

1.1. Getting started with C

- English language and C language learning are closely related.

Steps in learning English language:



Steps in learning C language:

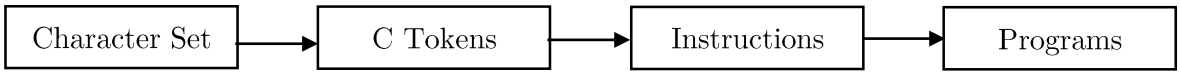


Figure 1.1.1 Steps in learning a language

- Like English, C language too has a set of rules (syntax rules) that one must follow while writing C programs.

1.2. C Character set

- The characters can be used to form words, numbers and expressions.
- The characters in C are grouped into the following categories:
  - Alphabets (Letters)
  - Digits
  - Special characters
  - White spaces

Table 1.2.1 C character set

Categories of Character	
Alphabets	A, B,……,Z a, b,……,z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ! # $? % ^ & * ( ) _ + | \ ` - = { } [ ] : ” ; < > ? , . /
White spaces	Blank space Horizontal tab Carriage return New line Form feed

1.3. C Tokens

- The smallest part of C program is called Token.
- C has mainly six types of tokens as shown below:

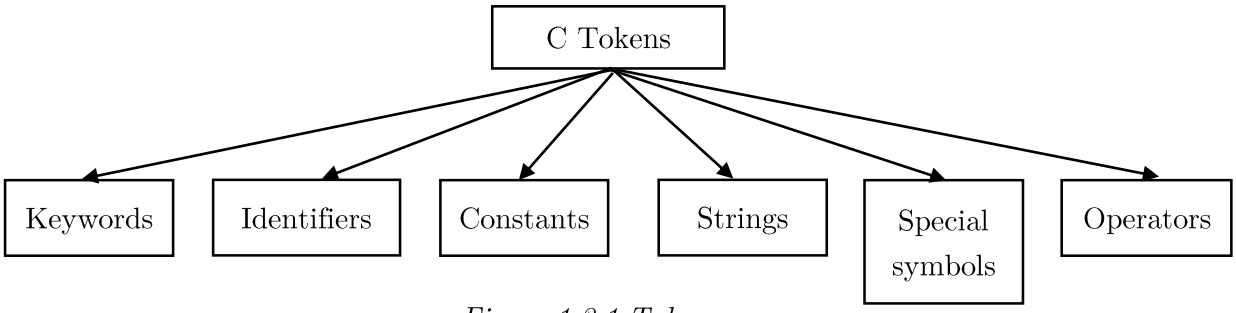


Figure 1.3.1 Tokens

1.4. Keywords

- Keywords are reserved words.
- Meaning of these words cannot be changed.
- Keywords serves as a basic building block for program statement.
- Keywords cannot be used as an identifier.
- List of keywords used in C are given below:

Table 1.4.1 Keywords

ANSI C Keywords			
Auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
continue	for	signed	void
Default	goto	sizeof	volatile
Do	if	static	while

1.5. Identifiers

- The identifiers are user defined names used in a program for providing names to variables, arrays and functions.
- The identifiers consist of letters (uppercase, lowercase), digits and underscore.
- Rules to define identifiers:
  - First character must be an alphabet or an underscore.
  - It must consist of only alphabets (a to z & A to Z), digits (0 to 9) & underscore (_).
  - Only first 31 characters are significant.
  - Cannot use C keyword.

- Must not contain white space.

1.6. Constants

- Constants in C refer to fixed values that do not change during the execution of a program.
- C supports following types of constants:

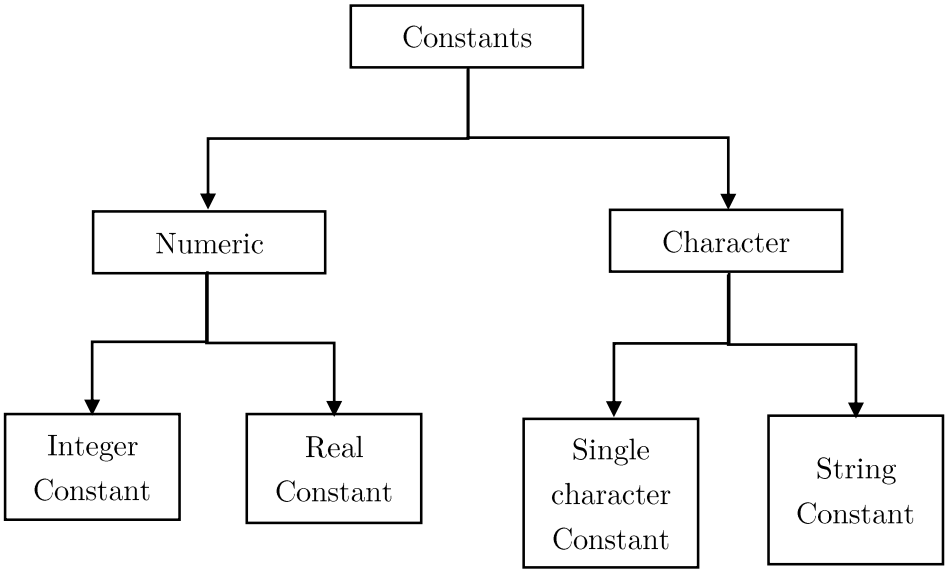


Figure 1.6.1 Types of Constants

Integer constant:

- Integer constant is a number without decimal point and fractional part.
- There are three types of integers constant.

Decimal integer

- Decimal integer consists of a set of digits, 0 to 9 having optional – or + sign. No other characters are allowed like space, commas, and non-digit characters.
- Example: 123, -321, 0, +78

Octal integer

- Octal integer consists of any combination of digits from the set 0 to 7. Octal numbers are always preceded by 0.
- Example: 037, 0, 0551

Hexadecimal integer

- Hexadecimal integer consists of any combination of digits from the set 0 to 9 and A to F alphabets. It always starts with 0x or 0X. A represents 10, B represents 11… F represents 15.
- Example: 0X2A, 0x95, 0xA47C.

Real constant:

- The number containing the fractional part is called real number. Ex: 0.0083, -0.75, +247.0, -0.75.
- A real number may also be expressed in exponential notation.
- The general form is: ***mantissa e exponent***

- Example: 215.65 can be written as 2.1565e2.

Single character constant:

- It contains single character enclosed within a pair of single quote mark.
- Example: '5', 'A', ';', ' '

String constant:

- A string constant is a sequence of characters enclosed in double quotes.
- The characters may be letters, numbers, special characters and blank space.
- Example: "Darshan University", "Hello world", "5+3", etc…
- A character constant 'X' and string constant "X" is not equivalent.

1.7. Variables

- Variable is a symbolic name given to some value which can be changed.
- A variable may take different values at different times during execution.
- A variable name may consist of letters, digits and the underscore (_) character.

Table 1.7.1 Variables

Variable name	Valid?	Remarks
First_tag	Valid	
Int	Invalid	int is a keyword
Price$	Invalid	$ sign is not permitted
CSE Branch	Invalid	Blank space is not permitted
Int_type	Valid	Keyword may be part of variable name

1.8. Data types

- C language is rich in its data types.
- It determines the type and size of data associated with variables.
- ANSI C supports following data types:

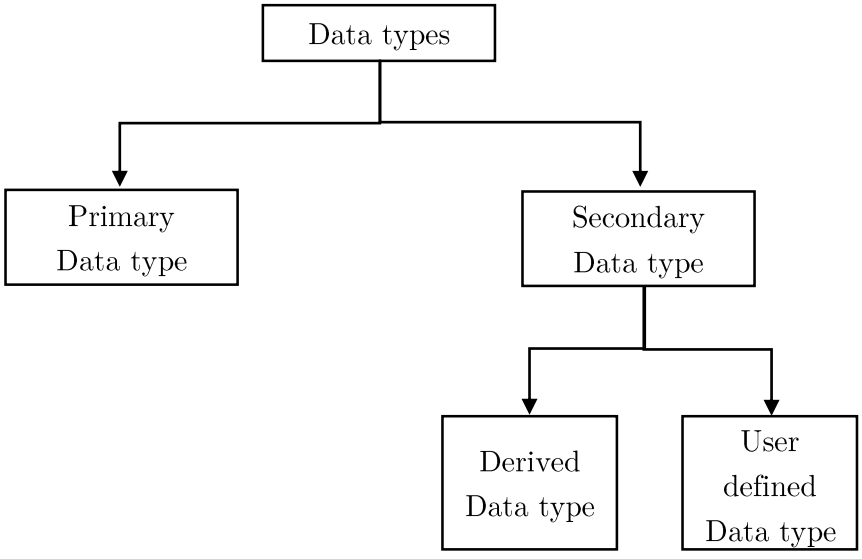


Figure 1.8.1 Data types

Primary data types

- Primary data types are built in data types which are directly supported by machine.
- They are also known as fundamental data types.
- The fundamental data types are:
    - int
    - float
    - char
    - void

Integer Types:

- int datatype can store integer number which is whole number without fraction part such as 10, 20, 30, etc.
- C language has three classes of integer storage namely ***short int, int and long int.***
- All of these data types have signed and unsigned forms.

Floating point Types:

- float data type can store floating point number which represents a real number with decimal point and fractional part with 6 digits of precision such as 10.50, 155.25 etc.
- floating point numbers are defined in C 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 up to 14 digits.

Character Types:

- Char data type can store single character of alphabet or digit or special symbol such as 'a', '5','$' etc.
- Each character is assigned some integer value which is known as ASCII values.

Void Types:

- The void type has no value therefore we cannot declare it as variable as we did in case of int or float or char.
- The void data type is used to indicate that function is not returning anything.

Table 1.8.1 Data types and it's range

Data Type	Storage Space	Value Range	Example
Char	1 bytes	-128 to 127	'a', '$' ,'1' , etc…
Int	2 bytes	-32768 to 32767	1, 2, 5, 9, 3, 0, etc..
Float	4 bytes	3.4E-38 to 3.4E+38	10.20, 20.30, 40.30, etc…
Double	8 bytes	3.4E-4932 to 1.1E+4932	10, 10.50, 100058, etc..

Secondary data type

- Secondary data types are not directly supported by the machine.
- It is combination of primary data types to handle real life data in more convenient way.
- There are two types of secondary data types:
    1. Derived data types

2. User defined data types

Derived data types: Derived data type is extension of primary data type. It is built-in system and its structure cannot be changed. Examples: Array and Pointer.

- **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
- **Pointer:** Pointer is a special variable which contains memory address of another variable.

User defined data types: User defined data type can be created by programmer using combination of primary data type and/or derived data type. Examples: Structure, Union, Enum.

- **Structure:** Structure is a collection of logically related data items of different data types grouped together under a single name.
- **Union**: Union is like a structure, except that each element shares the common memory.
- **Enum**: Enum is used to assign names to integral constants, the names make a program easy to read and maintain.

1.9. Operators

- C supports a rich set of built-in operators.
- An operator is a symbol that tells the computer to perform certain mathematical or logical operation.
- C operators can be classified in following categories:
  - Arithmetic operator
  - Relational operator
  - Logical operator
  - Assignment operator
  - Increment/Decrement operator
  - Conditional operator
  - Bitwise operator
  - Special operator

Arithmetic operators (+, -, *, /, %)

- Arithmetic operators are used for mathematical calculation.
- It can operate on any built-in data type allowed in C.
- A modulo division operator (%) cannot be used on floating data type.
- C does not have an operator for exponentiation.

Table 1.9.1 Arithmetic operators

Operator	Meaning	Written as	Description	Integer Arithmetic (Ex: a=14,b=4)
+	Addition	a + b	Addition of a and b	a + b = 18
-	Subtraction	a – b	Subtraction of b from a	a – b = 10

*	Multiplication	a * b	Multiplication of a and b	a * b = 56
/	Division	a / b	Division of a by b	a / b = 3 (Decimal part truncated)
%	Modulo division (remainder)	a % b	Modulo of a by b	a % b = 2

- If X,Y, and Z are floats, then:
  X = 6.0/7.0 = 0.857143
  Y = 1.0/3.0 = 0.333333
  Z = -2.0/3.0 = -0.666667

Relational operators (<, <=, >, >=, ==, !=)

- Relational operators are used to check relationship between two operands.
- If the relation is true, it returns 1; if the relation is false, it returns value 0.
- An expression containing a relational operator is termed as a relational expression.
- Relational expressions are used in decision statements such as if, for, while, etc…

Table 1.9.2 Relational operators

Operator	Meaning	Written as	Description	Example
<	is less than	a < b	a is less than b	6 < 4 is evaluated to 0 (false)
<=	is less than or equal to	a <= b	a is less than or equal to b	6 <= 4 is evaluated to 0 (false)
>	is greater than	a > b	a is greater than b	6 > 4 is evaluated to 1 (true)
>=	is greater than or equal to	a >= b	a is greater than or equal to b	6 >= 4 is evaluated to 1 (true)
==	is equal to	a == b	a is equal to b	6 == 4 is evaluated to 0 (false)
!=	is not equal to	a!=b	a is not equal to b	6!= 4 is evaluated to 1 (true)

Logical operators (&&, ||, !)

- An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false.

Table 1.9.3 Truth table

a	b	a&&b	a||b
0	0	0	0

0	1	0	1
1	0	0	1
1	1	1	1

- Logical operators are used to test more than one condition and make decisions.

Table 1.9.4 Logical operators

Operator	Meaning	Description	Example
&&	logical AND	Both non zero then true, either is zero then false	a>b && x==10
||	logical OR	Both zero then false, either is non zero then true	Number<0 || number>100
!	Logical NOT	True only if the operand is 0	!(a)

Assignment operators (+=, -=, *=, /=)

- Assignment operators (=) is used to assign the result of an expression to a variable.
- Assignment operator stores a value in memory.
- C also supports shorthand assignment operators which simplify operation with assignment.

Table 1.9.5 Assignment operators

Operator	Meaning	Same as
=	Assigns value of right side to left side	
+=	a += 1	a = a + 1
-=	a -= 1	a = a - 1
*=	a *= 1	a = a * 1
/=	a /= 1	a = a / 1
%=	a %= 1	a = a % 1

- The advantages of using shorthand operators are:
  - What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
  - The statement is more concise.
  - The statement is more efficient.

Increment and decrement operators (++, --)

- Increment (++) operator used to increase the value of the variable by one.
- Decrement (--) operator used to decrease the value of the variable by one.
- These two operators are unary operators, meaning they only operate on a single operand.

Table 1.9.6 Increment/Decrement operators

Operator	Example	Explanation	Output

Pre increment operator (++x)	x=10; p=++x;	First increment value of x by one then assign.	x will be 11 p will be 11
Post increment operator (x++)	x=10; p=x++;	First assign value of x then increment value.	x will be 11 p will be 10

Conditional operators (?:)

- A conditional operator is known as ternary operator.

Syntax: exp1 ? exp2 : exp3

- where exp1, exp2 and exp3 are expressions.

Working of the ?: Operator

- exp1 is evaluated first.
- if exp1 is true (nonzero), then
  - exp2 is evaluated and its value becomes the value of the expression
- If exp1 is false (zero), then
  - exp3 is evaluated and its value becomes the value of the expression
- **Example:**

m=2, n=3;

1. r=(m>n) ? m : n; Value of r will be 3
2. r=(m<n) ? m : n; Value of r will be 2

Bitwise operators (&, |, ^, <<, >>)

- Bitwise operators are used to perform operation bit by bit.
- Bitwise operators may not be applied to float or double.

Table 1.9.7 Bitwise operators

Operator	Description	Example

&	Binary AND Operator copies a bit to the result if it exists in both operands.	int a=8, b=6, c;
		c = a & b;
		printf("Output = %d", c);
		Output: 0

|	Binary OR Operator copies a bit if it exists in either operand.	int a=8, b=6, c;
		c = a | b;
		printf("Output = %d", c);
		Output: 14

^	Binary XOR Operator copies the bit if it is set in one operand but not both.	int a=8,b=6,c;
		c=a^b;
		printf("%d",c);

		Output: 14

<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	int a=8, b;
		b = a << 1;
		printf("Output = %d", b);
		Output: 16 (multiplying a by a power of two)

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	int a=8, b;
		b = a >> 1;
		printf("Output = %d", b);
		Output: 4 (dividing a by a power of two)

Special operators

Table 1.9.8 Special operators

Operator	Description
&	Address operator, it is used to determine address of the variable.
*	Pointer operator, it is used to declare pointer variable and to get value from it.
,	Comma operator. It is used to link the related expressions together.
Sizeof	It returns the number of bytes the operand occupies.
.	Member selection operator, used in structure.
→	Member selection operator, used in pointer to structure.

1.10. Input/Output Operations

- The ***stdio.h*** header file provides built in functions for reading (scanf()) data from input devices (keyboard) and writing (printf()) formatted data to output devices (monitor).

scanf():

- scanf() is a library function that reads data with specified format from standard input (keyboard).
- Syntax*:* scanf("control string" ,&variable1, &variable2...);
- Example*:* scanf("%d", &number);
- First argument is specification of format and other arguments are pointer variables to store data.
- The function returns the total number of items successfully read.
- The value is assigned to variable name.

printf():

- printf() is a library function that prints formatted data to standard output, generally monitor.
- Syntax*:* printf("control string" ,variable1, variable2….);
- Example*:* printf("Your marks are %d", mark);
- The control string indicated how many arguments follow and what their types are.

1.11. Basic structure of C program

Table 1.11.1 Basic structure of C program

Basic structure of C Program	Example
Documentation Section	// This program is to find area of circle
Link Section	#include<stdio.h>
Definition Section	#define PI 3.14
Global Declaration Section	int i; float areaofcircle(float);

main()
{
Declaration Part
Executable Part
}
	void main()
	{
	//Declaration Part
	float r,area;
	
	//Executable Part
	scanf("%f",&r);
	area=areaofcircle(r);
	printf("Area of Circle is :- %f",area);
	}

Subprogram Section (User Defined Section)
Function1()
Function2()
	float areaofcircle(float r)
	{
	return PI * r * r;
	}

1.11.1. Comments

- Comments makes code more readable and also used to explain the code.
- Comments are not part of the program.
- There are two types of comments:
  - **Single line Comments:** it starts with two forward slashes (//).
  - Any line between // and the end of the line is considered as comment and thus ignored by the compiler (will not be executed).
  - Example: // This is a comment

- **Multi-line Comments:** it starts with /* and ends with */.
- Any line between /* and */ is considered as multi-line comment and thus ignored by the compiler.
- Example: /* This is
  a multi-line comment */

1.11.2. Header files

- Header files contain the set of predefined standard library functions.
- The ***"#include"*** pre-processing directive is used to include the header files with ".h" extension in the program.
- Whenever we require the definition of a function, then we simply include that header file in which function is declared.
- There are two types of header files defined in a program:
  1. **System defined header file:**
     - The header file which is predefined is known as a system defined header file.
  2. **User-defined header file:**
     - The header file which is defined by the user is known as a user-defined header file.
- Some system headers files are given below:

Table 1.11.2.1 Header files

Name	Description
stdio.h	Input/Output Functions
conio.h	Console input/output
ctype.h	Character Handling Functions
math.h	Mathematics Functions
stdlib.h	General Utility Functions
string.h	String Functions
time.h	Date and Time Functions
errno.h	Testing error codes

1.12. Features of C language

- Modularity
- Extensibility
- Elegant syntax
- Case sensitive
- Less memory required
- The standard library concepts
- The portability of the compiler
- A powerful and varied range of operators

- Ready access to the hardware when needed

1.13. Arithmetic expression and evaluation

- An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language.
- C can handle any complex mathematical expression.

Table 1.13.1 Expressions

Arithmetic expression	C expression
a x b - c	a*b-c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{\mathbf{ab}}{\mathbf{c}}$	(a*b)/c
$\mathbf{3x^2+2x+1}$	(3*x*x)+(2*x)+1

Evaluation of expression

- Expressions are evaluated using an assignment statement of the form:

Variable=expression;

- The expression is evaluated first and result then assign to left hand side variable.

Examples:

```
X=a*b-c;
Y=b/c*a;
Z=a-b/c+d;
```

1.14. Type conversion

- C permits mixing of constants and variables of different types in an expression.
- C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance.
- This automatic conversion is known as ***implicit type conversion.***
- The lower type is automatically converted to the higher type before the operation proceeds. The result is higher type.

Example: Implicit Type Conversion

```
int a = 20;
double b = 20.5;
printf("%lf", a + b);
Output: 40.500000
```

- C uses the rule that, in all expressions except assignments, any implicit type conversions are made from lower size type to a higher size type as shown below:

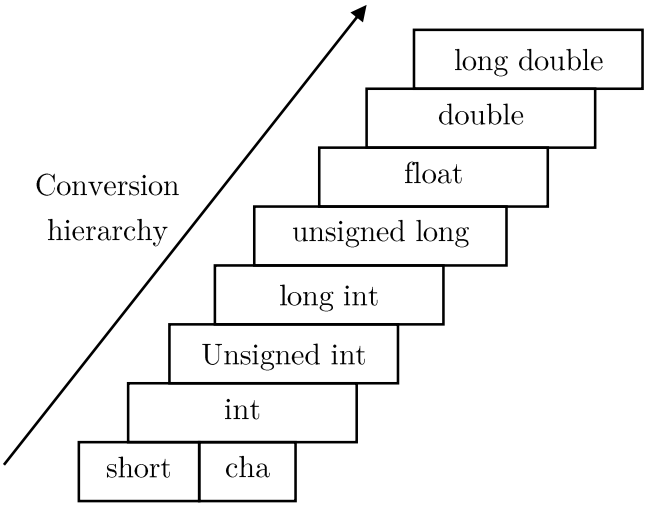


Figure 1.14.1 Conversion Hierarchy

- The final result of an expression is converted to the type of variable on the left of the assignment sign before assigning value to it.
- Following changes are introduced during final assignment.
  - float to int causes truncation of fractional part.
  - double to float causes rounding of digits.
  - long int to int causes dropping of the excess higher order bits.

Explicit Conversion

- Sometimes we want to force a type conversion in a way that is different from automatic conversion.
- The process of such a local conversion or casting is known as ***explicit casting***.
- The general form of cast is: ***(type-name) expression***
- Where type-name is one of the standard C data types. The expression may be constant, variable, or an expression.
- Some examples of casts and their actions shown below:

Table 1.14.1 Use of casts

Example	Action
X=(int)7.5	7.5 is converted to integer.
A=(int)21.3/(int)4.5	Evaluated as 21/5 and result would be 5.
B=(double)sum/n	Division is done in floating point mode.
Y=(int)(a+b)	The result of a+b is converted to integer.
Z=(int)a+b	a is converted to integer and then added to b.
P=cos((double)x)	Converts x to double before using it.

Example: Explicit Type Conversion

```
double a = 4.5, b = 4.6, c = 4.9;
int result = (int)a + (int)b + (int)c;
printf("result = %d", result);
```

Output: 12

1.15. Operator precedence and Associativity

Precedence

- Precedence of an operator is its priority in an expression for evaluation.
- There is distinct level of precedence.
- The operator at the higher level of precedence is evaluated first.
- Precedence rule decides the order in which different operators are applied.
- 5 + 3 * 2 is calculated as 5 + (3 * 2), giving 11, and not as (5 + 3) * 2, giving 16.
- We say that the multiplication operator (*) has higher "precedence" or "priority" than the addition operator (+), so the multiplication must be performed first.

Associativity

- The operator at the same precedence level is evaluated either from 'left' to 'right' or 'right' to 'left' depending on the associativity of an operator.
- Associativity rule decides the order in which multiple occurrences of the same level operators are applied.
- Operator associativity is why the expression 8 - 3 - 2 is calculated as (8 - 3) - 2, giving 3, and not as 8 - (3 - 2), giving 7.
- We say that the subtraction operator (-) is "left associative", so the left subtraction must be performed first. When we can't decide by operator precedence alone in which order to calculate an expression, we must use associativity.
- Following table provides a complete list of operators, their precedence level, and their rule of association.
- Rank 1 indicates highest precedence level and 15 is the lowest.

Table 1.15.1 Summary of C Operators

Operator	Description	Associativity	Rank
()	Function Call	Left to Right	1
[]	Array Element Reference		
+, -	Unary Plus, Unary minus	Right to Left	2
++, --	Increment, Decrement		
!	Logical Negation		
~	Ones Complement		
*	Pointer Reference (Indirection)		
&	Address		
sizeof	Size of an object		
*	multiplication	Left to Right	3
/	Division		
%	Modulus		

+ -	Addition Subtraction	Left to Right	4
<< >>	Left Shift Right Shift	Left to Right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right	6
== !=	Equality Inequality	Left to Right	7
&	Bitwise AND	Left to Right	8
^	Bitwise XOR	Left to Right	9
|	Bitwise OR	Left to Right	10
&&	Logical AND	Left to Right	11
||	Logical OR	Left to Right	12
?:	Conditional Expression	Right to Left	13
==, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=	Assignment Operators	Right to Left	14
,	Comma Operator	Left to Right	15

Precedence of Arithmetic operators

- An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operator.

Example: x=a-b/3+c*2-1

When a=9, b=12 and c=3 the statement becomes,

x=9-12/3+3*2-1

First Pass

Step-1: x=9-**12/3**+3*2-1

Step-2: x=9-4+***3*2***-1

Step-3: x=***9-4***+6-1

Second Pass

Step-4: x=***5+6***-1

Step-5: x=***11-1***

Step-6: x=10

1.16. Maths function

- Most of the C compiler supports basic math functions.
- To use math functions in a program, we should include ***math.h*** header file in the beginning of the program.
- List of some standard math functions given below:

Table 1.16.1 Math functions

Function	Meaning
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Reminder of x/y
log(x)	Natural log of x, x>0
log10(x)	Base 10 log of x, x>0
pow(x,y)	x to the power of y (x^y)
sqrt(x)	Square root of x, x>=0

1.17. Algorithm and Flowchart

Algorithm

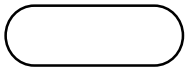
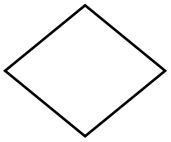
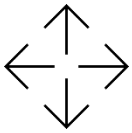
- Algorithm is a finite sequence of well-defined steps for solving a mathematical or computational problem.
- An algorithm takes some inputs, execute some finite number of steps and gives an output.
- It is written in the natural language like English.

- It is difficult to debug and to show branching and looping.
- **Example:** write an algorithm to check whether number is Positive or Negative.
  Step 1: Read no.
  Step 2: If no is greater than equal zero, go to step 4.
  Step 3: Print no is a negative number, go to step 5.
  Step 4: Print no is a positive number.
  Step 5: Stop.

Flowchart

- Flowchart is a pictorial or graphical representation of a problem.
- It is drawn using various symbols given below:

Table 1.12.1 Flowchart symbols

Notation	Description
	Start / Stop
	Input / Output (Read / Print)
	Process
	Decision making
	Subroutine
	Flow / Directions

- It is easy to understand and to show branching and looping.
- The flowchart symbols are linked together with arrows showing the process flow direction.

- **Example:** Draw a flowchart to check whether number is Positive or Negative.

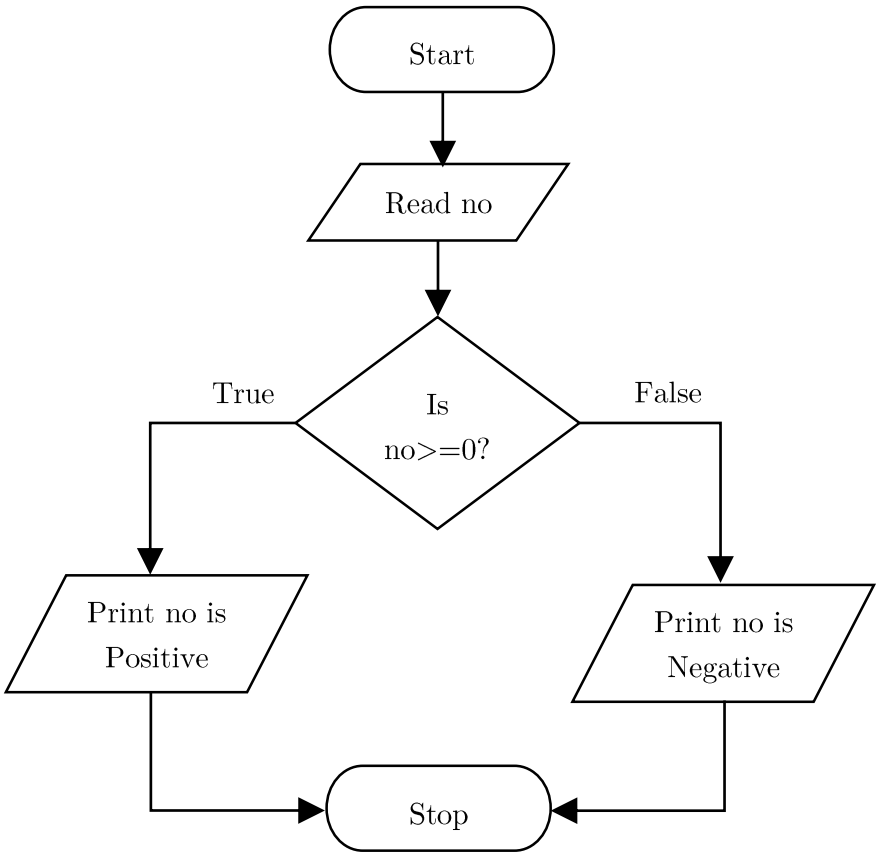


Figure 1.17.1 flowchart to check positive or negative number

Difference between Flowchart and Algorithm

Table 1.13.2 Difference between Flowchart & Algorithm

Flowchart	Algorithm
Flowchart is a pictorial or graphical representation of a program.	Algorithm is a finite sequence of well-defined steps for solving a problem.
It is drawn using various symbols.	It is written in the natural language like English.
Easy to understand.	Difficult to understand.
Easy to show branching and looping.	Difficult to show branching and looping.
Flowchart for big problem is impractical.	Algorithm can be written for any problem.

Write an Algorithm and draw a Flowchart to find largest number from given 3 no.

Flowchart

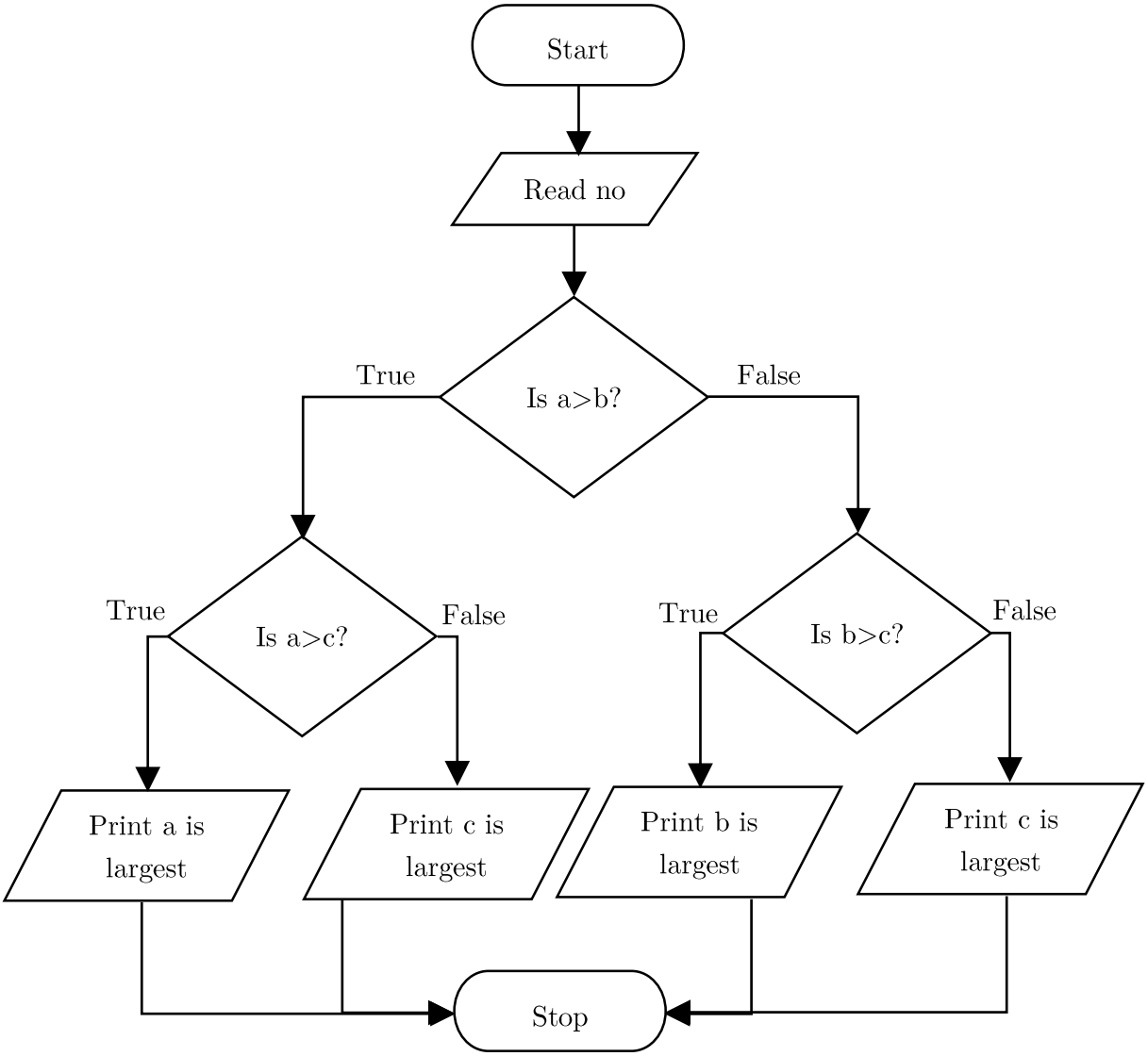


Figure 1.17.2 flowchart to find largest number from three number

Algorithm

Step 1: Read a, b, c.

Step 2: If a>b, go to step 5.

Step 3: If b>c, go to step 8.

Step 4: Print c is largest number, go to step 9.

Step 5: If a>c, go to step 7.

Step 6: Print c is largest number, go to step 9.

Step 7: Print a is largest number, go to step 9.

Step 8: Print b is largest number.

Step 9: Stop.

Write an Algorithm and draw a flowchart to print 1 to 10.

Flowchart

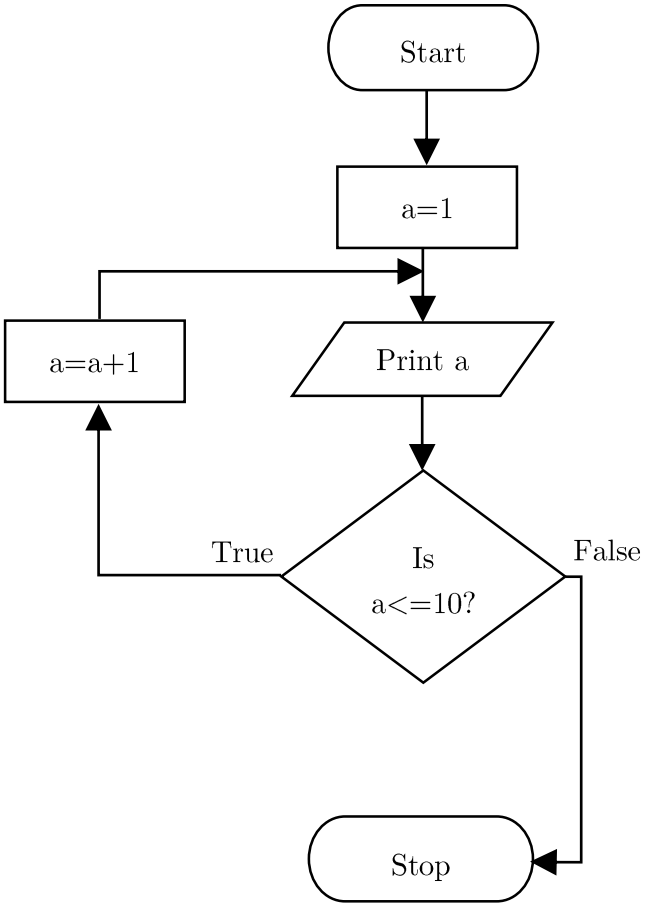


Figure 1.17.3 flowchart to print 1 to 10

Algorithm

Step 1: Initialize a to 1.

Step 2: Print a.

Step 3: Repeat step 2 until a<=10.

Step 3.1: a=a+1.

Step 4: Stop.

Darshan UNIVERSITY

Unit-2

Decision Making & Looping

2.1 Introduction

- C program is a set of statements which are normally executed sequentially.
- However, in practice, we have a number of situations where we have to change the order of execution of statement based on certain conditions.
- Decision Making statements (control statements) are used to control the flow of program.
- It evaluates condition or logical expression first and based on its result (either true or false), the control is transferred to particular statement.
- If result is true then it takes one path otherwise it takes another path.
- Decision making statements in C language are:
  1. If statement
  2. Switch statement
  3. Conditional operator statement
  4. goto statement

2.2 Decision making with if statement

- The if statement is powerful decision-making statement and is used to control the flow of execution of statements.
- It is two-way decision-making statement.

Syntax:

if(test expression)

- It allows computer to evaluate first and then, depending on whether the value of the expression (relation or condition) is 'true' or 'false', it transfers the control to a particular statement.
- It has two paths, one for true condition and other for the false condition.
- The if statement may be implemented in following different forms:
  1. Simple if statement
  2. if...else statement
  3. Nested if...else statement
  4. else if ladder statement

2.3 Simple if statement

- The general form of simple if statement is:

```
if (test expression)
{
        statement-block;
}
statement-x;
```

- It evaluates test expression first and based on its result, the control is transferred to the particular statement.
- If the test expression is true, the statement-block will be executed; otherwise, the statement-block will be skipped and the execution will jump to the statement-x.
- When the test expression is true, both the statement-block and the statement-x executed in sequence.

Flowchart:

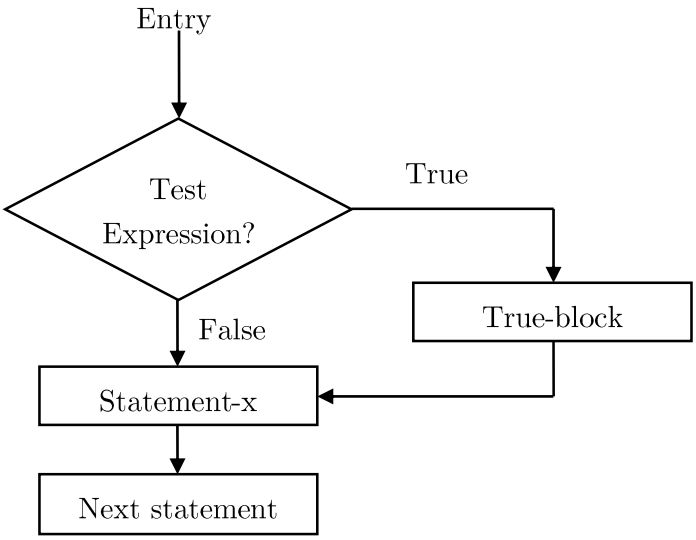


Figure 2.3.1 simple if statement flowchart

Example: Check whether given number is positive or negative using simple if statement.

```
#include<stdio.h>
void main()
{
    int a;
    printf("Enter Number:");
    scanf("%d",&a);
    if(a >= 0)
    {
        printf("Positive Number");
    }
    if(a < 0)
    {
        printf("Negative Number");
    }
}
```

2.4 if-else statement

- If supports statements only for true part, while if..else support statements for true part as well false part.
- if…else is two branch decision making statement.
- The general form of if..else statement is:

```
if (test expression)
{
        True – block statement(s);
}
else
{
        False – block statement(s);
}
statement-x;
```

- If the test expression is true, then the True-block statements(s) are executed; otherwise, false-block statement(s) are executed.
- Either true-block or false-block will be executed, not both.

Flowchart:

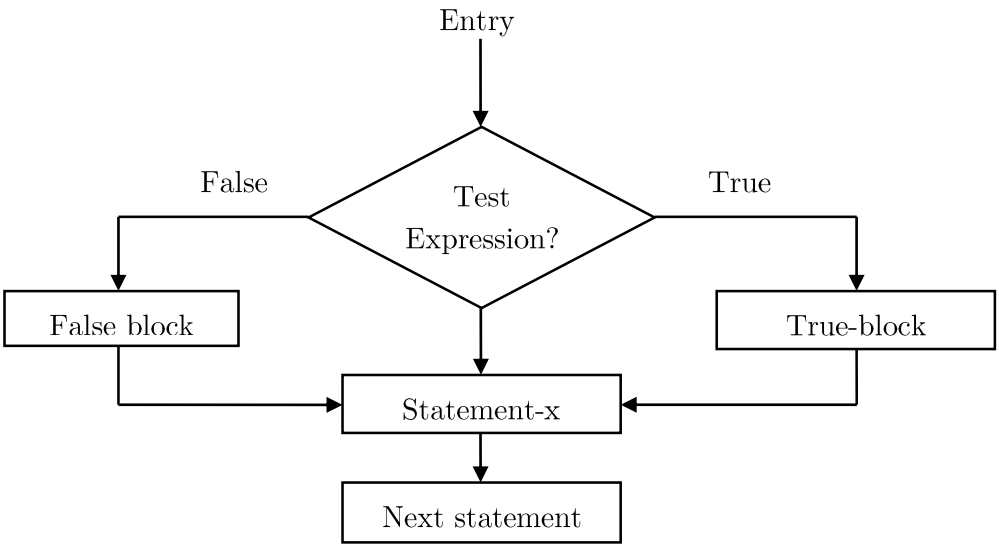


Figure 2.4.1 if-else statement flowchart

Example: Check whether given number is positive or negative using if-else statement.

```
#include<stdio.h>
void main()
{
    int a;
    printf("Enter Number:");
    scanf("%d",&a);
    if(a >= 0)
```

```
    {
        printf("Positive Number");
    }
    else
    {
        printf("Negative Number");
    }
}
```

2.5 Nested if-else

- When a series of decisions are involved, we may have to use more than one if…else statement in nested form.

```
if(test-condition-1)
{
        if(test-condition-2)
        {
                Statement-1;
        }
        else
        {
                Statement-2;
        }
}
else
{
         Statement-3;
}
Statement-x;
```

- If test-condition-1 is false then Statement-3 will be executed and then control will be transferred to statement-x.
- If test-condition-1 is true then test-condition-2 is evaluated. If it is true then Statement-1 will be executed, if it is false then Statement-2 will be executed and then control will be transferred to statement-x.

Flowchart:

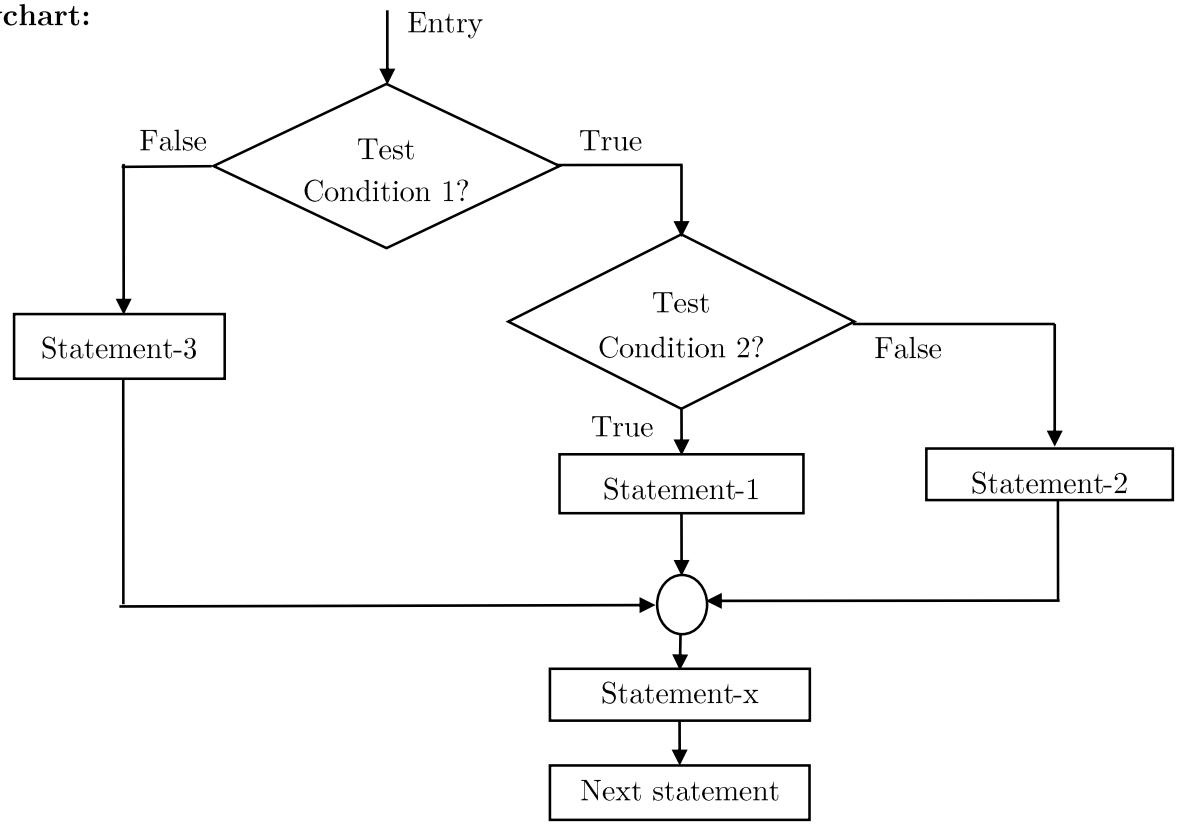


Figure 2.5.1. Nested if-else statement flowchart

Example: Find maximum number from given three numbers.

```
#include<stdio.h>
void main()
{
  int a,b,c;
  printf("Enter value of a, b, c:");
  scanf("%d%d%d", &a, &b, &c);
  if(a>b)
  {
        if(a>c)
              printf("a is max");
        else
              printf("c is max");
  }
  else
  {
        if(b>c)
              printf("b is max");
        else
```

```
            printf("c is max");
        }
    }
```

2.6 if-else if ladder

- It's another way of putting ifs together when multipath decisions are involved.
- It takes following general form:

```
if(condition-1)
        statement-1;
else if(condition-2)
        statement-2;
else if(condition-N)
        statement-N;
else
        default-statement;
statement-x;
```

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

Flowchart:

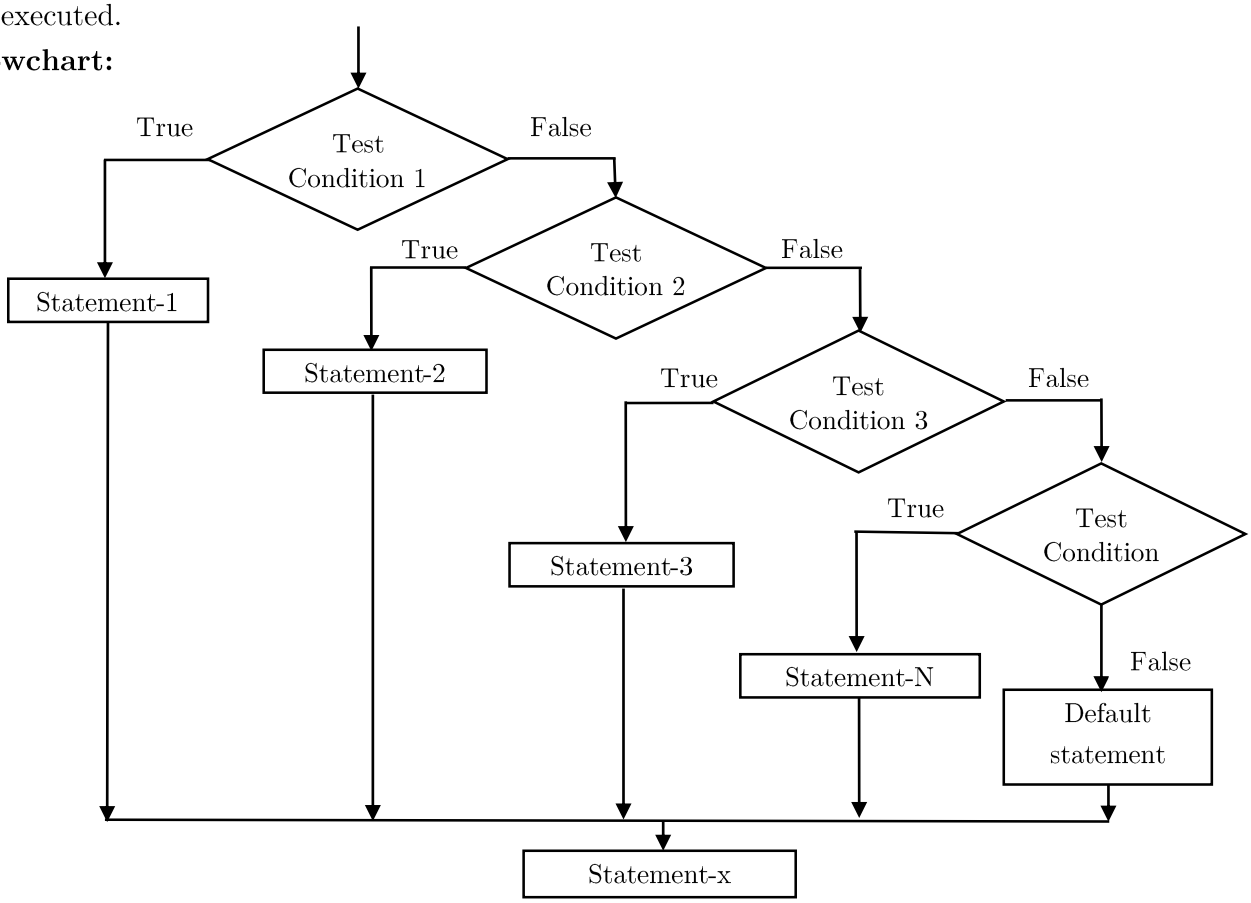


Figure 2.6.1 else-if ladder flowchart

Example: Check whether given number is positive, negative or zero using else if ladder.

```
#include<stdio.h>
void main()
{
    int a;
    printf("Enter Number:");
    scanf("%d",&a);
    if(a > 0)
        printf("Positive Number");
    else if(a==0)
        printf("Zero");
    else
        printf("Negative Number");
}
```

2.7 Switch case statement

- The switch statement allows to execute one code block among many alternatives.
- Switch is built-in multi-way decision statement.
- It works similar to if...else..if ladder.
- The expression is evaluated once and compared with the values of each case.
- If there is a match, the corresponding statements after the matching case are executed.
- If there is no match, the default statements are executed.
- If we do not use break, all statements after the matching label are executed.
- The default clause inside the switch statement is optional.

```
switch (expression)
{
    case constant1:
      // statements
      break;
    case constant2:
      // statements
      break;
    .
    .
    default:
   // default statements
}
```

Flowchart:

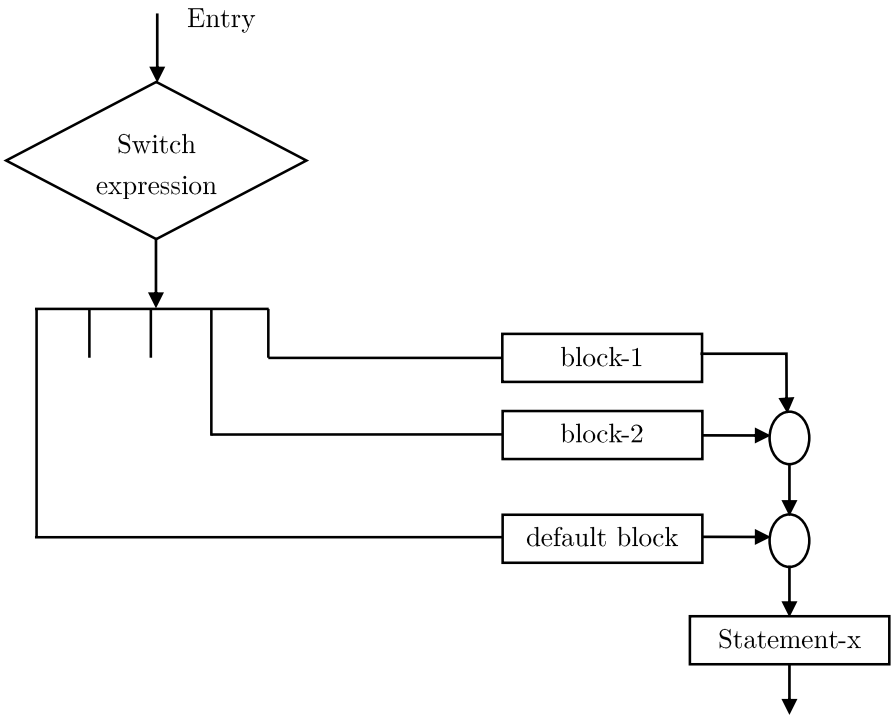


Figure 2.7.1 Switch-case statement flowchart

Example: Print Day name based on day number.

```
void main()
{
    int day;
    printf("Enter day number(1-7):");
    scanf("%d",&day);
    switch(day)
    {
        case 1:
                printf("Sunday");
                break;
        case 2:
                printf("Monday");
                break;
        case 3:
                printf("Tuesday");
                break;
        case 4:
                printf("Wednesday");
                break;
        case 5:
                printf("Thursday");
```

```
            break;
        case 6:
            printf("Friday");
            break;
        case 7:
            printf("Saturday");
            break;
        default:
            printf("Invalid input");
            break;
    }
}
```

Rules for switch statement

- The switch expression must be integral type. Float or other data types are not allowed.
- Case labels must be constant or constant expression.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colons.
- The break statement is optional.
- The default case statement is optional. There can be at most one default label.
- It is permitted to nest switch statement.

2.8 Conditional operator

- A conditional operator is known as ternary operator.
- The conditional operator works similar to the if-else.

Syntax: *exp1 ? exp2 : exp3*

- where exp1, exp2 and exp3 are expressions.

Working of the ?: Operator

- exp1 is evaluated first
- if exp1 is true (nonzero), then
  - exp2 is evaluated and its value becomes the value of the expression
- If exp1 is false (zero), then
  - exp3 is evaluated and its value becomes the value of the expression

Example:

consider m=2, n=3;

1. r=(m>n) ? m : n; Value of r will be 3
2. r=(m<n) ? m : n; Value of r will be 2

2.9 Differentiate switch and if-else

Table 2.9.1 Difference between if-else & switch-case

if-else	Switch-case
If statement is used to select among two alternatives.	The switch statement is used to select among multiple alternatives.
If can have values based on constraints.	Switch can have values based on user choice.
If implements Linear search.	Switch implements Binary search.
Float, double, char, int and other data types can be used in if condition.	Only int and char data types can be used in switch block.

2.10 Differentiate conditional operator and if-else

Table 2.10.1 Difference between conditional operator & if-else

Conditional operator	if-else
It is a programming statement.	It is a programming block.
It is not suitable for executing a function or a several statements	In that case, if else is more appropriate.
Nested ternary operator is not readable and cannot be debugged easily.	If else is more readable and easier to debug in case of issue.

Example:
Max=(a>b)?a:b;
	if(a>b)
	Max=a;
	else
	Max=b;

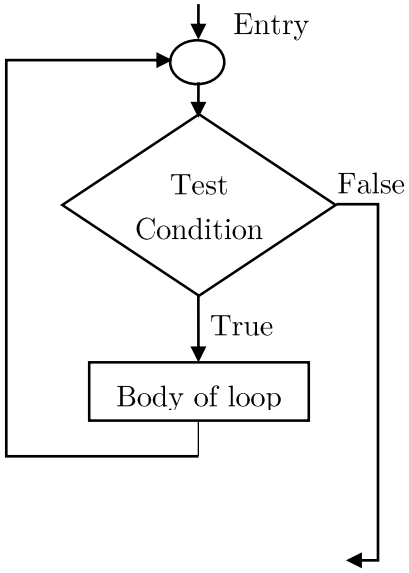
2.11 Loop and its types

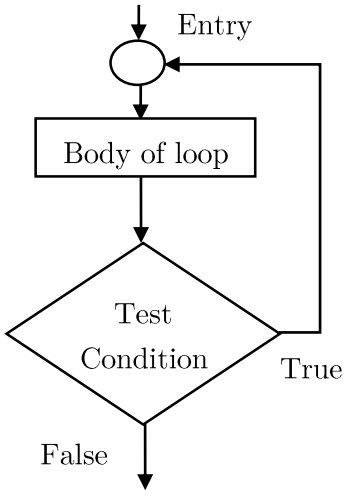
- Loop is used to execute the block of code several times according to the condition given in the loop.
- A loop mainly consists of two parts:
  - Body of the loop
  - Control statement
- The control statements test certain conditions and then directs the repeated execution of the statements.
- Depending on the position of the control statement in the body of loop, it is classified as the entry control loop or as an exit control loop.

Table 2.11.1. Difference between Entry & Exit control loop

Entry control loop	Exit control loop

Entry control loop checks condition before start of the loop execution.	Exit control loop first executes body of the loop and checks condition at last.
Body of loop may or may not be executed at all.	Body of loop will be executed at least once.
for, while are example of entry control loop.	do…while is example of exit control loop.





- A looping process, consist of a following steps:
  - Initialization
  - Test condition
  - Execution of the body of loop
  - Incrementing/ updating the condition variable

2.12 while loop

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

```
initialization;
while (test condition)
{
        body of the loop;
        increment or decrement;
}
```

- While loop is also known as entry control loop because first control-statement is executed and if it is true then only body of the loop will be 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 is repeated till the test condition is true. When it becomes false, 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 loop may have one or more statements.

Example: To print first 10 positive integer numbers

```
void main()
{
  int i;
  i = 1;                        \\ initialization of i
  while(i <= 10)                \\ condition checking
  {
        printf("\t%d",i);       \\ statement execution
        i++;                    \\ increment of control variable
  }
}
```

2.13 do-while loop

- Sometime, it might be necessary to execute the body of loop before the test is performed.
- Such situation can be handled with the help of the do statement.
- The general format of the do…while statement is:

```
initialization;
do
{
      body of the loop;
      increment or decrement;
}
while(test-condition);
```

- On reaching the do statement, the program evaluates the body of loop first time.
- At the end of the loop, the test condition in the while statement evaluated.
- If the condition is true, the program continues to evaluate the body of loop once again.
- This process continues as long as the condition is true.
- Since the test condition evaluated at the bottom of the loop, the do…while construct provides the exit-control loop and therefore the body of loop is always executed at least once.

Example: To print first 10 positive integer numbers

```
void main()
{
  int i;
  i = 1;                        \\ initialization of i
  do
  {
        printf("\t%d",i);       \\ statement execution
```

```
        i++;                    \\ increment of control variable
    } while(i <= 10);           \\ condition checking
}
```

2.14 for loop

- for loop is another entry-controlled loop that provides a more concise loop control structure.
- The general form of the for loop is:

```
for (initialization; test condition; increment/decrement)
{
        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. Initialization part will be executed exactly one time.
- Then it is checked with the given test condition. The test condition is relational expression. If the given condition is satisfied, the control enters into the body of the loop. If condition is false then it will exit from the loop.
- 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++.
- If new value of the control variable satisfies the loop condition, then the body of the loop is again executed. This process goes on till the control variable fails to satisfy the condition.

Example: find the sum of the first 10 positive integer numbers

```
void main()
{
        int i, sum=0;                   //declare variable int sum=0;
        for(i=1; i < = 10; i++)         // for loop
        {
                sum = sum + i ;         // add the value of i and store it to sum
        }
        printf("%d", sum);
}
```

2.15 Nested loop

- Nesting of loops allows the looping of statements inside another loop.
- The nesting may continue up to any desired level.
- It takes the following forms:

```
Outer loop
{
   Inner loop
   {
        // inner loop statements.
```

```
        }
            // outer loop statements.
    }
```

- Outer loop and inner loop can be for loop, while loop or do..while loop.

Example:

```
#include <stdio.h>
void main()
{
  int i, j, rows;
  printf("Enter the number of rows: ");
  scanf("%d", &rows);
  for (i = 1; i <= rows; ++i)
  {
        for (j = 1; j <= i; ++j)
        {
              printf("* ");
        }
        printf("\n");
   }
}
```

Output:

```
Enter the number of rows: 3
*
* *
* * *
```

2.16 Jump in to the loop

- Sometimes, when executing a loop, it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.
- C permits a jump from one statement to another statement with in a loop as well as a jump out of a loop.

Jumping out of the loop

- An early exit from a loop can be achieved using:
  1. break
  2. goto

2.16.1 break

- When, a break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.
- When the loop is nested, the break would only exit from the loop containing it. The break will exit only in a single loop.

Example: Read and sum numbers till -1 is entered

```
void main()
{
  int i, num=0;
  float sum=0,average;
  printf("Input the marks, -1 to end\n");
  while(1)
  {
       scanf("%d",&i);
       if(i==-1)
           break;
       sum+=i;
  }
printf("%d", sum);
}
```

2.16.2 goto

- A goto statement can transfer the control to any place in the program.
- It can be used to jump within the loop as well as exiting from the loop.
- The goto statement is marked by label statement. Label statement can be used anywhere in the function above or below the goto statement.
- Generally, goto should be avoided because its usage results in less efficient code, complicated logic and difficult to debug.

Example: Following program prints 10,9,8,7,6,5,4,3,2,1

```
void main ()
{
  int n=10;
  loop:
          printf("%d,",n);
          n--;
          if (n>0)
                  goto loop;
  }
```

2.16.3 Skipping a part of the loop

- The continue statement can be used to skip the rest of the body of an iterative loop.
- The continue statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION".
- Continue statement causes the control to go directly to the test condition and then continue to iteration process.

Example: Find sum of 5 positive integers. If a negative number is entered then skip it.

```
void main()
{
  int i=1, num, sum=0;
```

```
    for (i = 0; i < 5; i++)
    {
        scanf("%d", &num);
        if(num < 0)
            continue;    // starts with the beginning of the loop
        sum+=num;
    }
}
```

Darshan UNIVERSITY

Unit-3

Pointer, Arrays & Strings

3.1. Introduction

- An array is a fixed-size sequenced collection of elements of the same data type.
- It is derived data type.
- The individual element of an array is referred by their index or subscript value.
- An array provides a convenient structure for representing a data, it is classified as one of the data structures in C.
- Types of arrays are:
  - One dimensional array
  - Two-dimensional array
  - Multidimensional array

3.2. One dimensional array

- A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.

Syntax: data_type array_name[size];

Example: int Number[5];

- The data_type specifies the type of the elements that can be stored in an array, like int, float or char.
- The size indicates the maximum number of elements that can be stored inside the array.
- In above example, data type of an array is int and maximum elements that can be stored in an array are 5.
- The computer reserves five storage location as shown below:

Number[0]	Number[1]	Number[2]	Number[3]	Number[4]

- The value to the array element can be assigned as follows:

```
Number[0]  =  35;
Number[1]  =  40;
Number[2]  =  20;
Number[3]  =  57;
Number[4]  =  19;
```

- This would cause the array number to store the value shown below:

35	40	20	57	19
Number[0]	Number[1]	Number[2]	Number[3]	Number[4]

3.2.1 Array initialization

- After an array is declared, it's element must be initialized. Otherwise, they will contain "garbage" values.
- An array can be initialized at either of the following stages:
  1. At compile time
  2. At run time

Compile time initialization

- The general format of array initialization is:

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

- The value in the list is separated by commas.
- The array can be initialized with one of the following formats:

1.	int number[3]={0, 0, 0};	will declare the variable number as an array of size 3 and will assign zero to each element.
2.	int number[5]= {10, 50};	will initialize 0^{th} element of an array to 10, 1^{st} element to 50 and rest all elements will be initialized to 0.
3.	int number[ ]={1, 5, 6};	first of all, array size will be fixed to 3 then it will initialize 0^{th} element to 1, 1^{st} element to 5 and 2^{nd} element to 6.

Run time initialization

```
for(i=0;i<100;i++)
{
        if(i<50)
                sum[i]=0.0;
        else
                sum[i]=1.0;
}
```

- The first 50 elements of the array sum are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.
- We can also use a read function such as scanf() to initialize an array at runtime.

Example (one dimensional array): Read elements of an array and print it.

```
#include <stdio.h>
void  main()
{
   int arr[10];
   int i;
   printf("Read 10 elements in the array :\n");
   for(i=0; i<10; i++)
   {
     scanf("%d", &arr[i]);
   }
   printf("\nElements in array are: ");
   for(i=0; i<10; i++)
   {
```

```
        printf("%d  ", arr[i]);
    }
}
```

3.3. Two-dimensional Arrays

- Operators C allow us to define a table of items by using two-dimensional array.
- It is also called matrix.

Syntax: data_type array_name[row size][column_size];

Example: int matrix[4][3];

- First subscript denotes the number of rows and second subscript denotes the number of columns.
- Two dimensional arrays are stored in a memory as shown in below fig:

	Column 0	**Column 1**	**Column 2**
Row 0	[0][0]	[0][1]	[0][2]
	310	*275*	*365*
Row 1	[1][0]	[1][1]	[1][2]
	10	*190*	*325*
Row 2	[2][0]	[2][1]	[2][2]
	405	*235*	*240*
Row 3	[0][0]	[0][1]	[0][2]
	320	*270*	*380*

Initializing two-dimensional array

- The two-dimensional array can be initialized with one of the following formats:

1.	int table[2][3] = {0,0,0,1,1,1};	will initialize 1st row to zero, 2nd row to one.
2.	int table[2][3] = {{0,0,0},{1,1,1});	here, 1st group is for 1st row and 2nd group is for 2nd row.
3.	int table[2][3] ={{1,1},{2}};	initializes first two elements to the first row to one, the first element of second row to 2, and all other elements to zero.

Example (Two-dimensional array): Program to count number of Positive, Negative and Zeros in 3*3 matrix

```
#include<stdio.h>
void main()
{
    int a[3][3], i,j, p = 0, n = 0, z = 0;
    printf("Enter numbers\n");
```

```
    for(i=0; i<3; i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

for(i=0; i<3; i++)
    {
        for(j=0;j<3;j++)
        {
            if(a[i][j] > 0)
                p++;
            else if(a[i][j] < 0)
                n++;
            else
                z++;
        }
    }
     printf("\nPositive no: %d\nNegative no: %d\nZeros: %d\n", p, n, z);
}
```

3.4. Multi-dimensional Arrays

- C allows arrays of three or more dimensions.
- The exact limit is determined by the compiler.

Syntax: type array_name[s1][s2][s3].....[sm];

Example: int survey[3][5][12];

- Survey is three-dimensional array declared to contain 180 integer type elements.
- The array survey may represent a survey data of rainfall during last three years from January to December in five cities.

3.5. String it's Declaration & Initialization

- A string is a sequence of characters that is treated as a single data item.
- Any group of character define between double quotation mark is a string constant.
- Example: "Darshan University"
- C does not support string as a data type.
- However, it is allowed to represent string as character arrays.
- The general format of declaration of a string variable is:

  char string_name[size];
- The size determines the number of characters in the string_name.

Examples:

```
char city[10];
char name[30];
```

- when compiler assigns a characters string to a characters array, it automatically supplies a null characters ('\0') at the end of the string.
- Therefore, the size of array should be equal to the maximum number of characters in the string plus one.
- C permits a character array to be initialized in the either of the following two forms:

```
char city[9]= "NEW YORK";
char city[9]= {'N','E','W',' ','Y','O','R','K','\0'};
```

- C also permits us to initialize a character array without specifying the number of elements.

```
char city[ ]= {'N','E','W',' ','Y','O','R','K','\0'};
```

3.6. Reading and Writing a strings

3.6.2 Reading a string

Using scanf() function

- Input function scanf can be used with %s format specification to read in a string of characters.
- Example:

```
char address[10];
scanf("%s", address);
```

- The problem with the scanf() function is that it terminates its input on the first white space it finds.
- A white space includes blanks, tabs, carriage returns, form feeds and new lines.

Using gets() functions

- Another more convenient method of reading a string of text containing a whitespace is to use the library function gets() available in the <stdio.h> header file.
- It is function with one string parameter and called as under:

```
gets(str);
```

- it reads the characters from keyboard until a new-line character is encountered and then appends a null character to string.

3.6.2 Writing a string

Using printf() function

- printf() function with %s format specification is used to print a string.
- The format %s can be used to display an array of characters that is terminating by a null character.
- Example: printf("%s", name);
- Can be used to display the entire content of the array name.

Using puts() function

- Another more convenient method of printing a string value is to use the function puts declared in header file <stdio.h>.
- This is a one parameter function and invoked as under:

puts(str);

Where, str is a string variable containing a string value

3.7. Arithmetic Operations with Characters

- C allow us to manipulate character the way same we do with numbers.
- To write, a character in its integer representation, we may write it as an integer.
- if the machine uses the ASCII representation, then, following operations are valid.

1.	x='a'; printf("%d",x);	It prints 97
2.	x='z'-1;	In ASCII z is 122 therefore, it will assign 121 to x
3.	ch>='A'&& ch<='Z'	Would test whether character in variable ch is an upper-case letter.
4.	x=atoi("2022")	Converts string of digits to their integer value.

3.8. String Comparison

- C does not permit the comparison of two different string directly.
- That is, statement such as

  if(name == name2)

  if(name == "ABC") are not permitted.
- Therefore, it is necessary to compare two strings to be tested character by character.

Example: C program to Compare Two given Strings.

```
#include<stdio.h>
void main()
{
   char s1[100], s2[100];
   int i;
   printf("\nEnter two strings :");
   gets(s1);
   gets(s2);

   i = 0;
   // while s1 is equal to s2
   while (s1[i] == s2[i] && s1[i] != '\0')
      i++;
   if (s1[i] > s2[i])
      printf("s1 is greater than s2");
   else if (s1[i] < s2[i])
      printf("s1 is less than s2");
   else
      printf("s1 is equal to s2");
}
```

3.9. String Handling Functions

- C has several inbuilt functions to operate on string.
- These functions are known as string handling functions.
- For Example: char s1[]="Their", s2[]="There";

Table 3.9.1 String handling functions

Function	Meaning
strlen(s1)	Returns length of the string. l = strlen(s1); it returns 5
strcmp(s1,s2)	Compares two strings. It returns negative value if s1<s2, positive if s1>s2 and zero if s1=s2. printf("%d", strcmp(s1,s2)); Output : -9
strcpy(s1,s2)	Copies 2nd string to 1st string. strcpy(s1,s2) copies the string s2 in to string s1 so s1 is now "There". s2 remains unchanged
strcat(s1,s2)	Appends 2nd string at the end of 1st string. strcat(s1,s2); a copy of string s2 is appended at the end of string s1. Now s1 becomes "TheirThere"
strchr(s1,c)	Returns a pointer to the first occurrence of a given character in the string s1. printf("%s",strchr(s1,'i')); Output : ir
strstr(s1,s2)	Returns a pointer to the first occurrence of a given string s2 in string s1. printf("%s",strstr(s1,"he")); Output : heir
strrev(s1)	Reverses given string. strrev(s1); makes string s1 to "riehT"
strlwr(s1)	Converts string s1 to lower case. printf("%s", strlwr(s1)); Output : their
strupr(s1)	Converts string s1 to upper case. printf("%s", strupr(s1)); Output : THEIR

strncpy(s1,s2,n)	Copies first n character of string s2 to string s1 s1=""; s2="There";
	s1=""; s2="There";
	strncpy(s1,s2,2);
	printf("%s",s1);
	Output : Th

strncat(s1,s2,n)	Appends first n character of string s2 at the end of string s1.
	strncat(s1,s2,2);
	printf("%s", s1);
	Output : TheirTh

strncmp(s1,s2,n)	Compares first n character of string s1 and s2 and returns similar result as strcmp() function. printf("%d", strcmp(s1,s2,3)); Output : 0
strrchr(s1,c)	Returns the last occurrence of a given character in a string s1. printf("%s",strrchr(s2,'e')); Output : ere

3.10. Pointer

- A pointer is a variable that store address / reference of another variable.
- Pointer is derived data type in C language.
- A pointer contains the memory address of that variable as their value. Pointers are also called address variables because they contain the addresses of other variables.
- Syntax:

  datatype *ptr_variablename;

```
void main()
{
        int a=10, *p; // assign memory address of a to pointer variable p
        p = &a;
        printf("%d %d %d", a, *p, p);
}
```

  Output: 10, 10, 5000

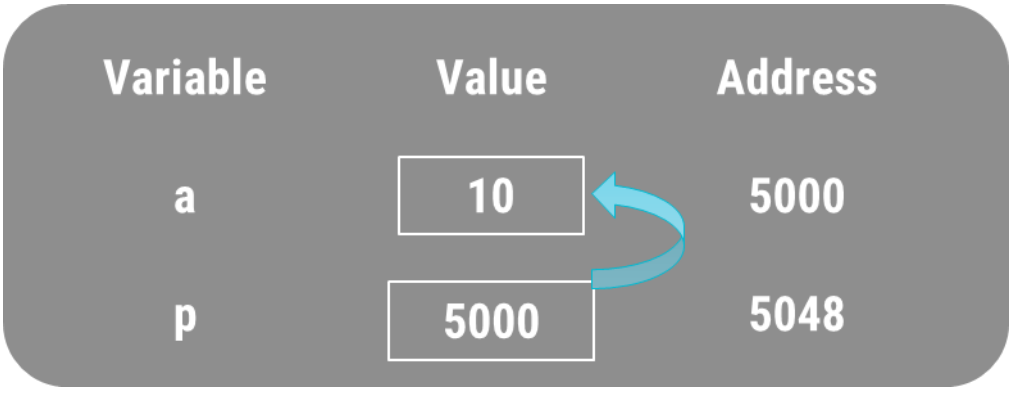


Figure 3.10 pointer

- p is integer pointer variable
- & is address of or referencing operator which returns memory address of variable.
- * is indirection or dereferencing operator which returns value stored at that memory address.

- & operator is the inverse of * operator
- x = a is same as x = *(&a)
- C uses pointers to create dynamic data structures, data structures built up from blocks of memory allocated from the heap at run-time. Example linked list, tree, etc.
- Pointers in C provide an alternative way to access information stored in arrays.
- Pointer use in system level programming where memory addresses are useful. For example, shared memory used by multiple threads.
- Pointers are used for file handling.

3.11. Pointer to Array

- When we declare an array, compiler allocates continuous blocks of memory so that all the elements of an array can be stored in that memory.
- The address of first allocated byte or the address of first element is assigned to an array name.
- Thus array name works as pointer variable.
- Example: int a[10], *p;
- a[0] is same as *(a+0), a[2] is same as *(a+2) and a[i] is same as *(a+i)

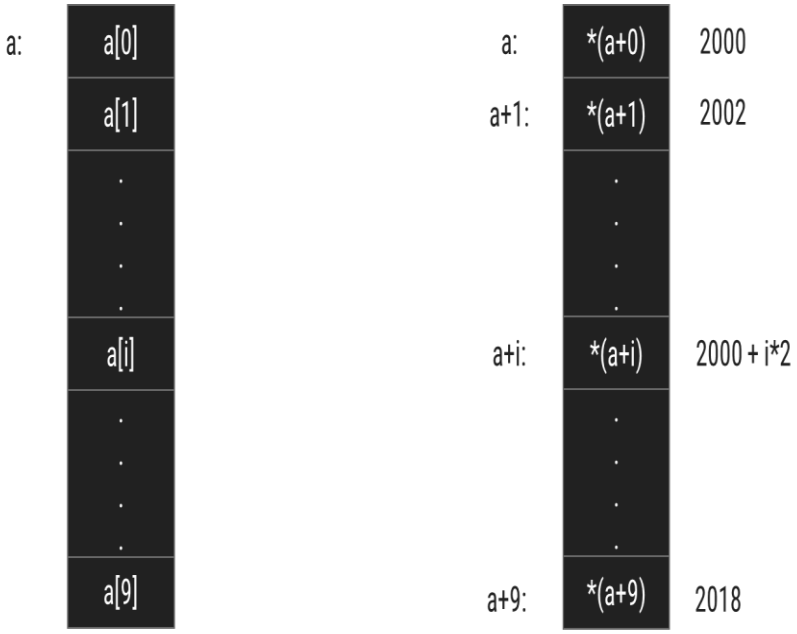


Figure 3.11 pointer to array

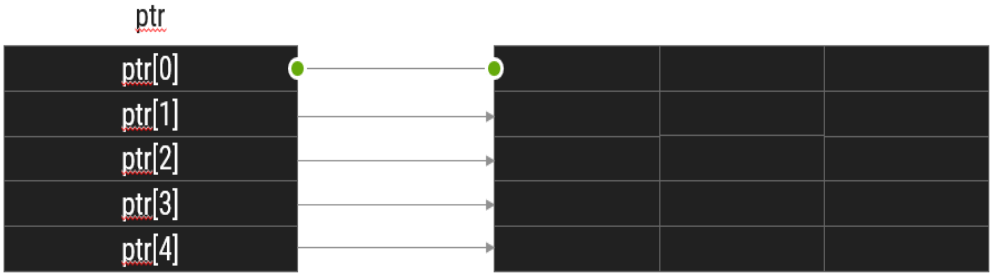
3.12. Array of pointers

- As we have an array of char, int, float etc, same way we can have an array of pointer.
- Individual elements of an array will store the address values.
- So, an array is a collection of values of similar type. It can also be a collection of references of similar type known by single name.
- Syntax: **datatype *name[size];**

- Example:
  int *ptr[5]; \\Declares an array of integer pointer of size 5
  int mat[5][3]; \\Declares a two dimensional array of 5 by 2

- Now, the array of pointers ptr can be used to point to different rows of matrix as follow

```
for(i=0;i<5;i++)
{
    ptr[i]=&mat[i][0]; \\ Can be re-written as ptr[i]=mat[i];
}
```



- By dynamic memory allocation, we do not require to declare two-dimensional array, it can be created dynamically using array of pointers.

Darshan UNIVERSITY

Unit-4
Functions

4.1. Function Definition

- A function is a group of statements that perform a specific task.
- It divides a large program into smaller parts.
- Program execution in C language starts from the main function.
- Syntax of function is described below

```
void main()
{
   // body part
}
```

- With the help of function, we can avoids rewriting the same code over and over.
- Using functions, it becomes easier to write programs and keep track of what they doing.

4.2. Library and User-defined function

- There are two types of functions 1) Library function 2) user defined function
- Library functions are those function whose definition is inside a header file
- Library functions are also known as predefined or inbuilt function.
- E.g. printf() function defined in stdio.h file, pow() function defined in math.h, etc.
- User Defined functions are those function whose definition is written (created) by user
- E.g. findSimpleInterest(), findtotal()

4.3. Program Structure for Function

- When we use a user-defined function program structure is divided into three parts.
- 1) function declaration (prototype) 2) function call 3) function definition

4.3.1 Function Declaration

- Like variables, all the functions must be declared before they are used.
- A function declaration tells the compiler about a function name and how to call the function.
- The function declaration is also known as function prototype or function signature.
- It consists of four parts,
  - Function type (return type).
  - Function name
  - Parameter List
  - Terminating semicolon

4.3.2 Function call

- Function is invoked from main function or other function that is known as function call.
- Function can be called by simply using a function name followed by a list of actual argument enclosed in parentheses.

4.3.3 Function Definition

- Function Definition is also called function implementation.
- It has mainly two parts.
  - Function header: It is same as function declaration but with argument name.
  - Function body: It is actual logic or coding of the function

4.4. Actual Parameter, Formal Parameter and return statement

- Values that are passed to the called function from the main function are known as Actual parameters.
- The variables declared in the function prototype or definition are known as Formal parameters.
- When a method is called, the formal parameter is temporarily "bound" to the actual parameter.
- If function is returning a value to calling function, it needs to use the keyword **return.**
- The called function can only return one value per call.

4.5. Categories of functions

- Functions can be classified in one of the following category based on whether arguments are present or not, whether a value is returned or not.
  - Functions with no arguments and no return value
  - Functions with no arguments and return a value
  - Functions with arguments and no return value
  - Functions with arguments and one return value

4.5.1 Function with no arguments and no return value

- When a function has no argument, it does not receive any data from calling function.
- When it does not return a value, the calling function does not receive any data from the called function.
- In fact, there is no data transfer between the calling function and called function.

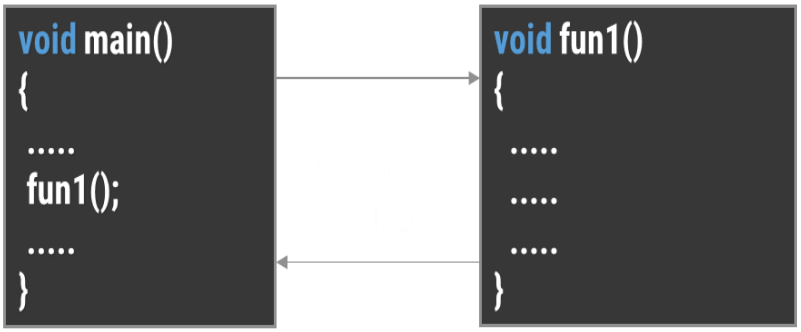


Figure 4.5.1 Function with no arguments and no return value

4.5.2 Function with no arguments and return value

- When a function has no argument, it does not receive data from calling function.
- When a function has return value, the calling function receives one data from the called function.

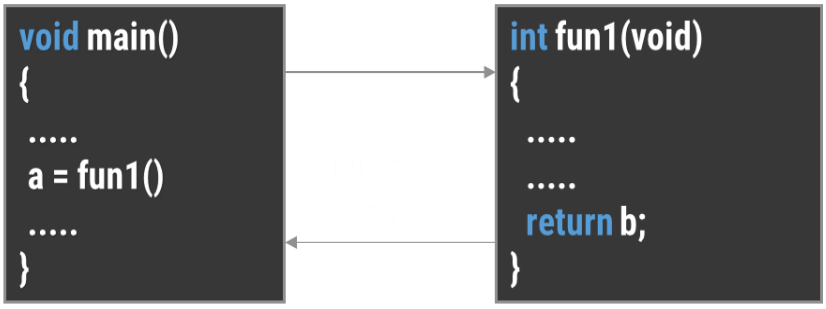


Figure 4.5.2 Function with no arguments and return value

4.5.3 Function with arguments and no return value

- When a function has argument, it receives data from calling function.
- When it does not return a value, the calling function does not receive any data from the called function.

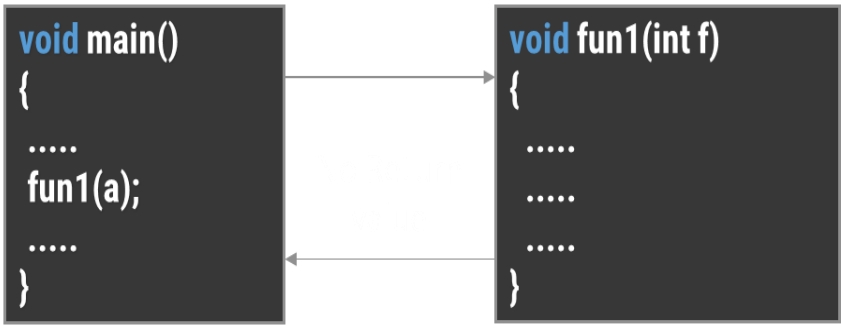


Figure 4.5.3 Function with arguments and no return value

4.5.4 Function with arguments and no return value

- When a function has argument, it receives data from calling function.
- When a function has return value, the calling function receives any data from the called function

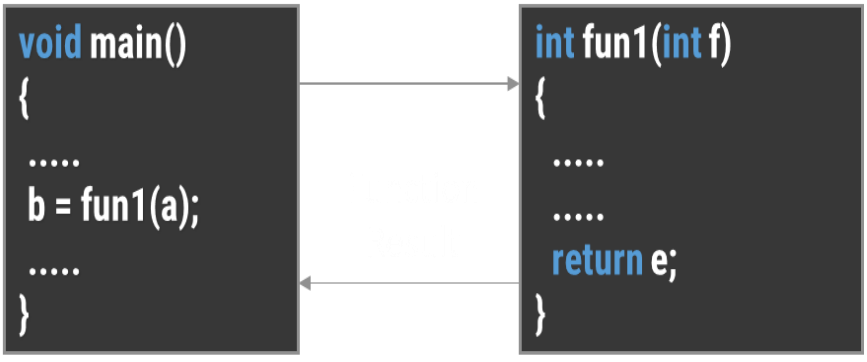


Figure 4.5.4 Function with arguments and return value

4.6. Recursion

- Any function which calls itself is called recursive function and such function calls are called recursive calls
- Recursion cannot be applied to all problems, but it is more useful for the tasks that can be defined in terms of a similar subtask.
- Any problem that can be solved recursively can be solved iteratively.
- When recursive function call itself, the memory for called function allocated and different copy of the local variable is created for each function call.
- A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have.
- Base class allows the recursion algorithm to stop.
- A base case is typically a problem that is small enough to solve directly.
- A recursive algorithm must change its state in such a way that it moves forward to the base case.

  E. g. $5! = 5 * 4!$
  $= 5 * 4 * 3!$
  $= 5 * 4 * 3 * 2!$
  $= 5 * 4 * 3 * 2 * 1!$
  $= 5 * 4 * 3 * 2 * 1$
  $= 120$

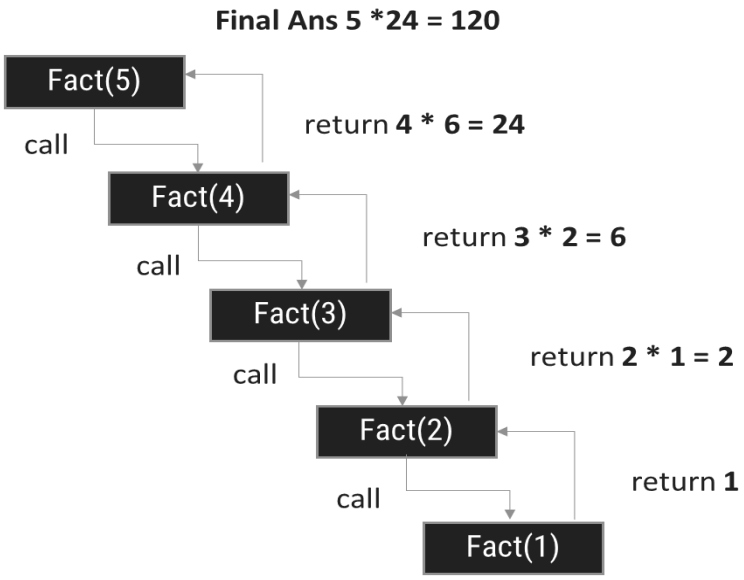


Figure 4.6 Recursion

4.7. Pointer and Function

- Like normal variable, pointer variable can be passed as function argument and function can return pointer as well.
- There are two approaches to passing argument to a function:

- Call by value
- Call by reference

4.7.1 Call by Value

- In call by value, the values of actual parameters are copied to their corresponding formal parameters.
- So the original values of the variables of calling function remain unchanged.
- Even if a function tries to change the value of passed parameter, those changes will occur in formal parameter, not in actual parameter.
- Example

```
#include<stdio.h>
void swap(int,int);
int main(){
        int x=10,y=20;
        swap(x,y);
        printf("\nValue inside main: %d %d",x,y);
        return 0;
}
void swap(int x,int y){
        int temp=x;
        x=y;
        y=temp;
        printf("Value inside swap function: %d %d",x,y);
}
```

Output:

```
Value inside swap function: 20 10
Value inside main: 10 20
```

4.7.2 Call by Reference

- In call by reference, the address of variable passed rather than its value hence modification done inside swap function can be reflected also in main.
- Example

```
#include<stdio.h>
void swap(int *,int *);
int main(){
        int x=10,y=20;
        swap(&x,&y);
        printf("\nValue inside main: %d %d",x,y);
        return 0;
}
void swap(int *x,int *y){
```

```
        int temp=*x;
        *x=*y;
        *y=temp;
        printf("Value inside swap function: %d %d",*x,*y);
}
```

Output:

Value inside swap function: 20 10

Value inside main: 20 10

Darshan UNIVERSITY

Unit-5

Structure & Unions

5.1. Structure

5.1.1 Introduction

- Structure is a collection of logically related data items of different data types grouped together under a single name.
- Structure is a user defined data type.
- Structure helps to organize complex data in a more meaningful way.
- Syntax of Structure:

```
struct structure_name
{
    data_type member1;
    data_type member2;
    ……….
};
```

- struct is a keyword.
- structure_name is a user define name of a structure.
- member1, member2 are members of structure.
- Example:
- struct student

```
{
      char name[30]; // Student Name
      int roll_no; // Student Roll No
      float CPI; // Student CPI
      int backlog; // Student Backlog
};
```

- Terminate structure definition with semicolon;.
- Cannot assign value to members inside the structure definition, it will cause compilation error.

```
struct student
{
   char name[30] = “ABC”;          // Compilation error
   . . .
};
```

5.1.2 Structure Variable

- *struct struct_name variable_name*
- In C programming, there are two ways to declare a structure variable.
  - Along with structure definition

- After structure definition.

5.1.2.1 Along with structure definition

```
struct structure_name
{
    member1_declaration;
    member2_declaration;
. . .
    memberN_declaration;
} structure_variable;
```

5.1.2.2 After structure definition

```
struct structure_name
{
    member1_declaration;
    member2_declaration;
. . .
    memberN_declaration;
} ;
struct structure_name structure_variable;
```

5.1.3 Access Structure member

- Structure is a complex data type, we cannot assign any value directly to it using assignment operator. Need to access an individual member of structure to assign a value. To assign a value to structure member two operators are used 1) Dot (.) operator 2) Arrow (->) operator.

5.1.3.1 Dot(.) operator

- It is known as member access operator. Use dot operator to access members of simple structure variable.
- Syntax:
  structure_variable.member_name;
- Example:
  student1.CPI = 7.46;

5.1.3.2 Arrow (->) operator

- In C language it is illegal to access a structure member from a pointer to structure variable using dot operator.
- Arrow operator is used to access structure member from pointer to structure.
- Syntax:
  pointer_to_structure->member_name;
- Example:
  // Student1 is a pointer to student type
  student1 -> CPI = 7.46

Write a program to read and display student information using structure.

```
#include <stdio.h>
struct student
{
    char name[40]; // Student name
    int roll; // Student enrollment
    float CPI; // Student mobile number
    int backlog;
};
int main()
{
    struct student student1; // Simple structure variable
    // Input data in structure members using dot operator
    printf("Enter Student Name:");
    scanf("%s", student1.name);
    printf("Enter Student Roll Number:");
    scanf("%d", &student1.roll);
    printf("Enter Student CPI:");
    scanf("%f", &student1.CPI);
    printf("Enter Student Backlog:");
    scanf("%d", &student1.backlog);
    // Display data in structure members using dot operator
    printf("\nStudent using simple structure variable.\n");
    printf("Student name: %s\n", student1.name);
    printf("Student Enrollment: %d\n", student1.roll);
    printf("Student CPI: %f\n", student1.CPI);
    printf("Student Backlog: %i\n", student1.backlog);
}
```

Output:

```
Enter Student Name:Raj
Enter Student Roll Number:111
Enter Student CPI:7.89
Enter Student Backlog:0

Student using simple structure variable.
Student name: Raj
Student Enrollment: 111
Student CPI: 7.890000
Student Backlog: 0
```

5.1.4 Array of Structure

- It can be defined as the collection of multiple structure variables where each variable contains information about different entities.

- The array of structures in C are used to store information about multiple entities of different data types.
- Syntax:

```
struct structure_name
{
        member1_declaration;
        member2_declaration;
        ...
        memberN_declaration;
} structure_variable[size];
```

WAP to read and display N student information using array of structure

```
#include<stdio.h>
struct student {
    char name[20];
    int rollno;
    float cpi;
};
int main( ) {
    int i,n;
    printf("Enter how many records u want to store : ");
    scanf("%d",&n);
    struct student sarr[n];
    for(i=0; i<n; i++)
    {
        printf("\nEnter %d record : \n",i+1);
        printf("Enter Name : ");
        scanf("%s",sarr[i].name);
        printf("Enter RollNo. : ");
        scanf("%d",&sarr[i].rollno);
        printf("Enter CPI : ");
        scanf("%f",&sarr[i].cpi);
    }
    printf("\n\tName\tRollNo\tMarks\t\n");
    for(i=0; i<n; i++) {
        printf("\t%s\t\t%d\t\t%.2f\t\n", sarr[i].name, sarr[i].rollno, sarr[i].cpi);
    }
return 0;
}
```

5.1.5 Nested Structure

- When a structure contains another structure, it is called nested structure.
- For example, we have two structures named Address and Student. To make Address nested to Student, we have to define Address structure before and outside Student structure and create an object of Address structure inside Student structure.
- Syntax:

```
struct structure_name1
{
    member1_declaration;
    member2_declaration;
    ...
    memberN_declaration;
};
struct structure_name2
{
    member1_declaration;
    member2_declaration;
    ...
    struct structure1 obj;
};
```

WAP to read and display student information using nested of structure.

```
#include<stdio.h>
struct Address
{
    char HouseNo[25];
    char City[25];
    char PinCode[25];
};
struct Student
{
    char name[25];
    int roll;
    float cpi;
    struct Address Add;
};
int main()
{
    int i;
    struct Student s;
```

```
        printf("\n\tEnter Student Name : ");
        scanf("%s",s.name);
        printf("\n\tEnter Student Roll Number : ");
        scanf("%d",&s.roll);
        printf("\n\tEnter Student CPI : ");
        scanf("%f",&s.cpi);
        printf("\n\tEnter Student House No : ");
        scanf("%s",s.Add.HouseNo);
        printf("\n\tEnter Student City : ");
        scanf("%s",s.Add.City);
        printf("\n\tEnter Student Pincode : ");
        scanf("%s",s.Add.PinCode);
        printf("\nDetails of Students");
        printf("\n\tStudent Name : %s",s.name);
        printf("\n\tStudent Roll Number : %d",s.roll);
        printf("\n\tStudent CPI : %f",s.cpi);
        printf("\n\tStudent House No : %s",s.Add.HouseNo);
        printf("\n\tStudent City : %s",s.Add.City);
        printf("\n\tStudent Pincode : %s",s.Add.PinCode);
        return 0;
    }
```

Output:

```
Details of Students
    Student Name : aaa
    Student Roll Number : 111
    Student CPI : 7.890000
    Student House No : 39
    Student City : rajkot
    Student Pincode : 360001
```

5.1.6 Structure using pointer

- Reference/address of structure object is passed as function argument to the definition of function.

  Example

```
#include <stdio.h>
struct student {
        char name[20];
        int rollno;
        float cpi;
};
int main()
```

```
{
    struct student *studPtr, stud1;
    studPtr = &stud1;
    printf("Enter Name: ");
    scanf("%s", studPtr->name);
    printf("Enter RollNo: ");
    scanf("%d", &studPtr->rollno);
    printf("Enter CPI: ");
    scanf("%f", &studPtr->cpi);
    printf("\nStudent Details:\n");
    printf("Name: %s\n", studPtr->name);
    printf("RollNo: %d", studPtr->rollno);
    printf("\nCPI: %f", studPtr->cpi);
    return 0;
}
```

5.1.7 Union

- Union is a user defined data type similar like Structure.
- It holds different data types in the same memory location.
- Can define a union with various members, but only one member can hold a value at any given time.
- Unions are used when all the members are not assigned value at the same time.
- Example

```
union student
{
    char name[30]; // Student Name
    int roll_no; // Student Roll No
    float CPI; // Student CPI
    int backlog; // Student Backlog
} student1;
```

- Union must be terminated with semicolon;.
- Cannot assign value to members inside the union definition, it will cause compilation error.
- Example

```
union student
{
   char name[30] = "ABC"; // Compilation error
  . . .
} student1;
```

5.1.8 Difference between structure and union

Table 5.1.8 Difference between structure and union

Comparison	Structure	Union
Basic	The separate memory location is allotted to each member of the structure.	All members of the 'union' share the same memory location.
Keyword	'struct'	'union'
Size	Size of Structure = sum of size of all the data members.	Size of Union = size of the largest member.
Store Value	Stores distinct values for all the members.	Stores same value for all the members.
At a Time	A structure stores multiple values, of the different members, of the structure.	A union stores a single value at a time for all members.

Declaration	struct ss
	{
	int a;
	float f;
	char c
	};
		union uu
		{
		int a;
		float f;
		char c
		};

5.2. File Management

- In real life, we want to store data permanently so that later we can retrieve it and reuse it.
- A file is a collection of characters stored on a secondary storage device like hard disk, or pen drive.
- There are two kinds of files that programmers deal with text files and binary files.
- Text file are human readable and it is a stream of plain English characters.
- Binary files are not human readable. It is a stream of processed characters and ASCII symbols.

5.2.1 File Modes

Table 5.2.1 File Modes

Mode	Description
R	Open the file for reading only. If it exists, then the file is opened with the current contents; otherwise an error occurs.
W	Open the file for writing only. A file with specified name is created if the file does not exist. The contents are deleted, if the file already exists.
A	Open the file for appending (or adding data at the end of file) data to it.

	The file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
r+	The existing file is opened to the beginning for both reading and writing.
w+	Same as w except both for reading and writing.
a+	Same as a except both for reading and writing.

5.2.2 File Handling Functions

Table 5.2.2 File handling functions

Syntax	Description
fp=fopen(file_name, mode);	This statement opens the file and assigns an identifier to the FILE type pointer fp. Example: fp = fopen("printfile.c","r");
fclose(filepointer);	Closes a file and release the pointer. Example: fclose(fp);
fprintf(fp, "control string", list);	Here fp is a file pointer associated with a file. The control string contains items to be printed. The list may include variables, constants and strings. Example: fprintf(fp, "%s %d %c", name, age, gender);
fscanf(fp, "control string", list);	Here fp is a file pointer associated with a file. The control string contains items to be printed. The list may include variables, constants and strings. Example: fscanf(fp, "%s %d", &item, &qty);
char getc(FILE *fp);	getc() returns the next character from a file referred by fp; it require the FILE pointer to tell from which file. It returns EOF for end of file or error. Example: c = getc(fp);
putc(int c, FILE *fp);	putc() writes or appends the character c to the FILE fp. If a putc function is successful, it returns the character written, EOF if an error occurs. Example: putc(c, fp);
int getw(FILE *pvar);	getw() reads an integer value from FILE pointer fp and returns an integer. Example: i = getw(fp);
putw(int, FILE *fp);	putw writes an integer value read from terminal and are written to the FILE using fp. Example: putw(i, fp);
EOF	EOF stands for "End of File". EOF is an integer defined in <stdio.h> Example: while(ch != EOF)

5.2.3 File Positioning Functions

- fseek, ftell, and rewind functions will set the file pointer to new location.
- A subsequent read or write will access data from the new position.

Table 5.2.3 File positioning functions

Syntax	Description
fseek(FILE *fp, long offset, int position);	fseek() function is used to move the file position to a desired location within the file. fp is a FILE pointer, offset is a value of datatype long, and position is an integer number. Example: /* Go to the end of the file, past the last character of the file */ fseek(fp, 0L, 2);
long ftell(FILE *fp);	ftell takes a file pointer and returns a number of datatype long, that corresponds to the current position. This function is useful in saving the current position of a file. Example: /* n would give the relative offset of the current position. */ n = ftell(fp);
rewind(fp);	rewind() takes a file pointer and resets the position to the start of the file. Example: /* The statement would assign 0 to n because the file position has been set to the start of the file by rewind. */ rewind(fp);

Write a C program to count lines, words, tabs, and characters.

```
#include <stdio.h>
void main()
{
    FILE *p;
    char ch;
    int ln=0,t=0,w=0,c=0;
    p = fopen("text1.txt","r");
    ch = getc(p);
    while (ch != EOF) {
        if (ch == '\n')
            ln++;
        else if(ch == '\t')
            t++;
        else if(ch == ' ')
            w++;
        else
            c++;
        ch = getc(p);
```

```
    }
    fclose(p);
    printf("Lines = %d, tabs = %d, words = %d, characters = %d\n",ln, t, w, c);
}
```

5.3. Dynamic Memory Allocation

- If memory is allocated at runtime (during execution of program) then it is called dynamic memory.
- It allocates memory from **heap** (*heap*: it is an empty area in memory).
- Memory can be accessed only through a pointer.
- This dynamic memory allocation is used when number of variables are not known in advance or large in size.
- Memory can be allocated at any time and can be released at any time during runtime.

5.3.1 Dynamic Memory allocation/deallocation functions

- There are four dynamic memory allocation/deallocation function in c.
  1)malloc () 2)calloc() 3)realloc () 4)free()
  First three are for memory allocation and last one is for memory deallocation.

5.3.1.1 malloc()

- malloc() is used to allocate a certain amount bytes of memory during the execution of a program.
- malloc() allocates size_in_bytes bytes of memory from heap, if the allocation succeeds, a pointer to the block of memory is returned else NULL is returned.
- malloc() returns an uninitialized memory for you to use.
- malloc() can be used to allocate space for complex data types such as structures.
- Allocated memory space may not be contiguous.
- The blocks are kept in ascending order of storage address, and the last block points to the first
- Syntax:
  *ptr_var = (cast_type *) malloc (size_in_bytes);*
  This statement returns a pointer to size_in_bytes of uninitialized storage, or NULL if the request cannot be satisfied.
- Example:
  *fp = (int *)malloc(sizeof(int) *20);*

  Write a C program to allocate memory using malloc.

```
#include <stdio.h>
void main()
{
    int *fp; //fp is a pointer variable
```

```
    fp = (int *)malloc(sizeof(int)); //returns a pointer to int size storage
    *fp = 25; //store 25 in the address pointed by fp
    printf("%d", *fp); //print the value of fp, i.e. 25
    free(fp); //free up the space pointed to by fp
}
```

5.3.1.2 calloc()

- calloc() is used to allocate a block of memory during the execution of a program.
- calloc() allocates a region of memory to hold no_of_blocks of size_of_block each, if the allocation succeeds then a pointer to the block of memory is returned else NULL is returned.
- The memory is initialized to ZERO.
- Syntax:
  *ptr_var = (cast_type *)* ***calloc*** *(no_of_blocks, size_of_block);*
  This statement returns a pointer to no_of_blocks of size size_of_blocks, it returns NULL if the request cannot be satisfied.
- Example:
  int n = 20;
  *fp = (int *)calloc(n, sizeof(int));*

  Write a C program to allocate memory using calloc.

```
#include <stdio.h>
void main()
{
    int i, n; //i, n are integer variables
    int *fp; //fp is a pointer variable
    printf("Enter how many numbers: ");
    scanf("%d", &n);
    fp = (int *)calloc(n, sizeof(int)); //calloc returns a pointer to n blocks
    for(i = 0; i < n; i++)  //loop through until all the blocks are read
    {
        scanf("%d",fp); //read and store into location where fp points
        fp++; //increment the pointer variable
    }
    free(fp); //frees the space pointed to by fp
}
```

5.3.1.3 realloc()

- realloc() changes the size of the object pointed to by pointer fp to specified size.
- The contents will be unchanged up to the minimum of the old and new sizes.
- If the new size is larger, the new space will be uninitialized.

- realloc() returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *fp is unchanged.
- Syntax:
  *ptr_var = (cast_type *) realloc (void *fp, size_t);*
  This statement returns a pointer to new space, or NULL if the request cannot be satisfied.
- Example:
  *fp = (int *)realloc(fp,sizeof(int)*20);*

Write a C program to allocate memory using realloc.

```
#include <stdio.h>
void main()
{
    int *fp; //fp is a file pointer
    fp = (int *)malloc(sizeof(int)); //malloc returns a pointer to int size storage
    *fp = 25; //store 25 in the address pointed by fp
    fp =(int *)realloc(fp, 2*sizeof(int)); //returns a pointer to new space
    printf("%d", *fp); //print the value of fp
    free(fp); //free up the space pointed to by fp
}
```

5.3.1.4 free()

- When the memory is not needed anymore, you must release it calling the function free.
- free deallocates the space pointed to by fp.
- It does nothing if fp is NULL.
- fp must be a pointer to space previously allocated by calloc, malloc or realloc
- Syntax:
  *void free(void *);*
  This statement free up the memory not needed anymore.
- Example:
  free(fp);

Write a C program to sort numbers using malloc.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int i,j,t,n;
    int *p;
    printf("Enter value of n: ");
    scanf("%d", &n);
```

```
    p=(int *) malloc(n * sizeof(int));
    printf("Enter values\n");
    for(i=0; i<n; i++)
        scanf("%d", &p[i]);
    for(i=0; i<n; i++)
    {
        for(j= i+1; j<n; j++)
        {  if(p[i] > p[j])
            {
                t = p[i];
                p[i] = p[j];
                p[j] = t;
            }
        }
    }
    printf("Ascending order\n");
    for(i=0; i<n; i++)
        printf("%d\n", p[i]);
    free(p);
}
```