COMPUTER I SEM

Programming Fundamentals — Unit I

Introduction :

Concept of Algorithm :

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in a finite amount of time.

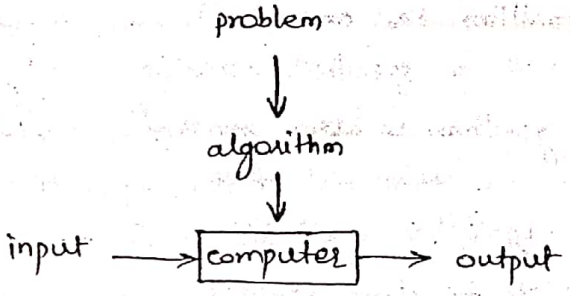


Figure: The notion of the algorithm.

Eg: Euclid's algorithm for computing gcd (m,n)

Step 1: If n=0, return the value of m as the answer and stop; otherwise, proceed to step 2.

Step 2: Divide m by n and assign the value of the remainder to r

Step 3: Assign the value of n to m and the value of r to n. Goto step 1.

pseudocode :

```
ALGORITHM  Euclid (m,n)
// Computes gcd (m,n) by Euclid's algorithm
// Input: Two non-negative, not both-zero integers m and n
// Output: Greatest common divisor of m and n
while n≠0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

refer: cs-fundamentals.com/tech-interview/dsa/what-is-an-algorithm.

A well defined algorithm has five important features:

* Finiteness: An algorithm must always terminate after a finite number of steps.
* Definiteness: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
* Input: An algorithm has zero or more inputs; i.e. quantities which are given to it initially before the algorithm begins.
* Output: An algorithm has one or more outputs; i.e. quantities which have a specified relation to the inputs.
* Effectiveness: An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

refer: https://www.smartdraw.com/flowchart/flowchart-symbols.htm

Flowcharts:

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process.

- Each step in the sequence is noted within a diagram shape.
- Steps are linked by connecting lines and directional arrows.

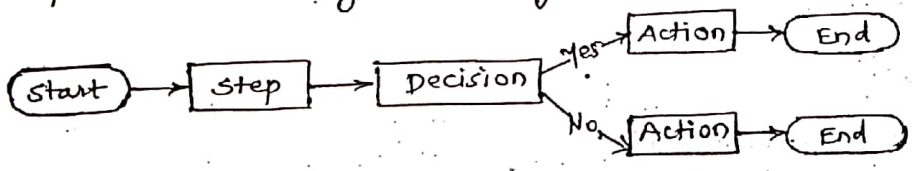
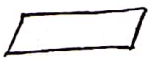
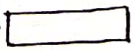


Fig. A basic flowchart.

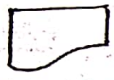
Symbols used in Flowcharting

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/output	A parallelogram represents input or output
	process	A rectangle represents a process.
	Decision	A diamond indicates a decision

Refer: asq.org/learn-about-quality/process-analysis-tools/overview/flowchart.html

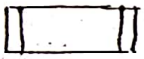
Delay or wait — To represent delay in process.

Link to another page or another flowchart. — shows linking to another page or another flowchart.



Document — represent document.

Database — represents database.



predefined process — predefined process

Refer: https://www.lucidcha com/pages/what-is-a-flowchart-tutorial

↓

Types of flowcharts:

- Document Flowcharts
- Data flowcharts
- System flowcharts
- Program flowcharts

} Sterneckert book "Critical Incident Management"

- System flowchart
- General flowchart
- Detailed flowchart

} Veronis book "Microprocessors: Design and Applications"

- System flowchart
- Program flowchart

} Bohl, book "A guide for programmers"

- Decision flowchart
- Logic flowchart
- Systems flowchart
- Product Flowchart
- Process flowchart

} Fryman, book "Quality and Process Improvement" for business perspective.

Example: ?

Introduction to different Programming Languages

Basics: Introduction to C

Variables, constants, arithmatic, control flow, functions, input and output.

First program:

pre-processor directive → `#include <stdio.h>`

```
main()
{
    printf("hello, world\n");
}
```

Turbo C/C++ — WINDOWS

→ How to compile a program
Alt+F9

→ How to Run a program.
Ctrl+F9

LINUX.

→ compile `gcc filename.c`

→ output `./a.out`

- stdio.h → Header file for standard input output library
- main() → parent function.
- { → start of a program } scope
- } → End of a program } (corresponding to first curly brace)
- printf → To print the statement/value within quotes (" ").
- \n → next line (new line)
- \t → tab.
- \b → backespace.

Comments: Comment may appear anywhere in program.
There can be a single line comment or multiple line comments.

→ Single line comment can be represented by // (double forward slash)

- Multiple line comments can be represented by /* .... */. whatever written in between /* and */ will be treated as comment.

Example:

```
int main (void)
{
  printf ("Hello world \n");
  return (0);
}
```

Braces enclose the main function body, which contains declarations and executable statements.

Reserved words:

- Identifiers from standard libraries.
- names for memory cells.
- All the reserved words appear in lowercase.

eg. int, void, double, return.

Identifiers:

There are two types of identifiers

1) Standard, 2) User defined

eg. printf, scanf

eg. KMS_PER_MILES

Rules for selecting identifiers:

1) An identifier must consist only of letters, digits and underscores.
2) An identifier cannot begin with a digit.
3) A C reserved word cannot be used as an identifier.
4) An identifier defined in a C standard library should not be redefined.

eg. letter_1, inches, CENT_PER_INCH, Hello, variable (valid identifiers)

eg. 1letter, double, int, TWO*FOUR, joe's (Invalid identifiers)

Uppercase and lowercase letters:

The names like Rate, rate, and RATE are viewed by C compiler as different identifiers.

- In C all the reserved words, ~~are~~ names of all standard library functions use only lowercase letters.
- Industry uses all uppercase letters in the names of constant macros.

Variable Declarations and Data types

* variable : a name associated with a memory cell whose value can change.

- variable declarations: statements that communicate to the compiler the names of variables in the program and the kind of information stored in each variable.

Syntax display for Declarations:

Syntax:
```
int variable_list ;
double variable_list;
char variable_list;
```

Examples:
```
int count,
    large;
double x, y, z;
char first_initial;
char ans;
```

* Data types:

A data type is a set of values and a set of operations on those values.

- A standard data type in C is a data type that is predefined, such as char, double, int
- double and int used for real numbers and integers resp.

int
- int must include at least the values -32760 through 32767.
- we can store integer in a type int variable,
- can perform common arithmatic operations (add, subtract, multiply, divide) and compare two integers.

eg. -10500, 435 +15 -25 32767

double
- a real number has an integral part and a fractional part that are separated by a decimal point.
- In C, the data type double is used to represent real numbers.
  eg. 3.14159, 0.0005, 150.0
- we can store a real numbers in a type double variable, perform the common arithmatic operations (add, subtract, multiply, divide) & compare them.

integer types in C

type	Range
short	-32,768 .. 32767
unsigned short	0 .. 65535
int	-2,147,483,647 .. 2,147,483,647
unsigned	0 .. 4,294,967,295
long	-2,147,483,647 .. 2,147,483,647
unsigned long	0 .. 4,294,967,295

char - Data type char represents individual character value.
- a letter, a digit or a special symbol.
eg. 'A' 'x' '2' '9' '*' ':' ',' ' '

floating point types in C

type	Approximate Range*	Significant digits*
float	$10^{-37} .. 10^{38}$	6
double	$10^{-307} .. 10^{308}$	15
long double	$10^{-4931} .. 10^{4932}$	19

* In a typical microprocessor-based C implementation

ASCII code : a particular code that specifies the integer representing each char value.
The digit characters '0' through '9' have code values of 48 through 57 (decimal).
In ASCII, uppercase letters have the decimal code values 65 through 90.
Lowercase letters have the consecutive decimal code values 97 through 122.

Assignment Statements: An assignment statement stores a value or a computational result in a variable.

```
kms = KMS_PER_MILE * miles;
```

In 'C' the symbol = is the assignment operator.

Form: variable = expression;

Example: x = y + z + 2.0 ;

Input/Output Operations and Functions.

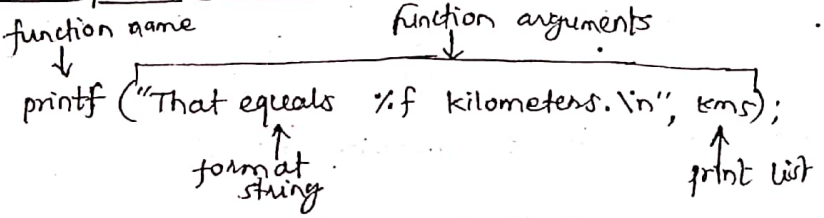
- The data transfer from the outside world into memory is called an input operation.
- Program results can be displayed to the program user by an output operation.

```
# include <stdio.h>
```

Function call:

- In 'C' a function call is used to call or activate a function.

The printf function



The function call consists of two parts: the function name and the function arguments.

A placeholder: always begins with the symbol %
%f marks the display position for a type double variable.

\n represents newline escape sequence.
we can use multiple placeholders.

Syntax: Display for printf function call.

Syntax: printf (format string, print list);
printf (format string);

Examples: printf("I am %d years old and my salary is %f : \n", age, sal

The scanf statement:

- The statement `scanf("%f", &miles);` calls function scanf to copy data into the variable miles.
- It copies data from standard input devices.
- In most cases the standard input devices is keyboard.
- The & (ampersand) character is the C address-of operator.

Syntax display of scanf function call

Syntax: `scanf (format string, input list);`

Example: `scanf ("%c%d", &first_initial, &age);`

The return statement:

`return (0);`

transfers control from your program to the operating system.

- The value in parentheses, 0, is considered the result of function main's execution, it indicates that your program executed without error.

Comments in a program

→ // used for a single line comment. in C++

→ /* .. */ can be used for single as well as multiple line comments.

Eg.
```
double miles;   // input - distance in miles      → in c++
double kms;     /* output - distance in kilometers */ → in 'c'
```

Arithmatic expression:

Arithmatic Operators:	Meaning	Examples
+	addition	5+2 is 7
-	subtraction	5.0 - 2.0 is 3.0
*	multiplication	5*2 is 10
/	division	5.0/2.0 is 2.5
%	remainder	5%2 is 1

Operator / and %

Result of integer division.

3/15 = 0	18/3 = 6
15/3 = 5	16/-3 varies.
16/3 = 5	0/4 = 0
17/3 = 5	4/0 is undefined.

Result of % operation

3%5 = 3	5%3 = 2
4%5 = 4	5%4 = 1
5%5 = 0	15%-7 varies
7%5 = 2	15%0 is undefined

Type conversion through Casts

C allows the programmer to convert the type of an expression by placing the desired type in parentheses before the expression, an operation called the **type cast**.

eg. `n= (int)(9* 0.5);`

Expressions with multiple operators:

- Expressions with multiple operators are common in C.
- Expressions can include both unary and binary operators.
- **Unary operators** can take only one operand.
- Unary negation (-) and plus (+) operators.

```
x= -y;
p= +x*y;
```

- **Binary operators** require two operands.

```
x= y+z;
z= y-x;
```

Rules for evaluating expressions:

a) **parentheses rule:** All expressions in parentheses must be evaluated separately.
- Nested parentheses must be evaluated inside out. with innermost expression evaluated first.

b) **Operator precedence rule:** Operators in the same expression are evaluated in the following ~~rule~~ order:

unary +,-	first
*, /, %	next
binary +,-	last

⑤ Associativity rule: Unary operators in the same sub expression and at the same precedence level (such as + and -) are evaluated right to left (right associativity).

- Binary operators in the same subexpression and the same precedence level (such as + and -) are evaluated left to right. (left associativity).

For example; $x * y * z + a / b - c * d$

with parentheses:

$$(x * y * z) + (a / b) - (c * d)$$

- The formula for area of circle.

$$a = \pi r^2$$

can be written as

area = PI * radius * radius;

The formula for the average velocity, v, of a particle traveling on a line between points p_1 & p_2 in time t_1 to t_2 is

$$v = \frac{p_2 - p_1}{t_2 - t_1}$$

$$v = (p_2 - p_1) / (t_2 - t_1)$$

Mathematical ~~expression~~ formulas as C expressions

Mathematical formula	C expression
1. $b^2 - 4ac$	b*b - 4*a*c
2. a+b-c	a+b-c
3. $\frac{a+b}{c+d}$	(a+b) / (c+d)
4. $\frac{1}{1+x^2}$	1 / (1+x*x)
5. a X -(b+c)	a * -(b+c)

Formatting Values of type int

printf("Results: %3d meters = %4d ft. %2d in.\n", meters, feet, inch);

⑪. o/p: Results: 21 meters = 68 ft. 11 in (if meters is 21, feet is 68 & inch 11

Formatting values of type double

Format specification for a type double value, we must indicate both the total field width needed and the number of decimal places desired.

- The total field width should be large enough to accomodate all digits before and after the decimal point.
- The form of the format string placeholder is %.n.mf where n is a number representing the total field width, and m is the desired number of decimal places.
- we must include the column for -(negative) sign also.

Eg.

Value	Format	Displayed output	value	Format	Displayed output.
3.14159	%.5.2f	-3.14	3.14159	%.4.2f	3.14
-.006	%.8.3f	--0.006	-.006	%.8.5f	-0.00600

* Interactive mode — the programs ~~that~~ user interacts with the program and types in data while it is running.

and

* Batch mode. — the program scans its data from a data file prepared beforehand instead of interacting with its user.

```
#include <stdio.h>
#include <conio.h>
#define KMS_PER_MILE 1.609
int main(void)
{
    double miles, kms  /* distance in miles and kilometers */
    printf("\n Enter the distance : ");
    scanf("%Lf", &miles);
    printf("The distance in miles is %.2f.\n", miles);
  /* convert the distance to kilometers */
    kms = KMS_PER_MILE * miles;
  /* display the distance in kilometers */
    printf("That equals %.2f kilometers.\n", kms);
    return(0);
}
```

(12) o/p: Enter the distance: 112
The distance in miles is 112.00.
That equals 180.21 kilometers.

Common programming errors:

Errors are so common that they have their own special name -bugs- and the process of correcting them is called debugging a program.

Three kinds of error occurs:
syntax errors, runtime errors & logic errors.

* Syntax errors: It occurs when your code violets one or more grammar rules of C and is detected by the compiler.
- The program contains the following syntax errors:
  - Missing semicolon at the end of the variable declarations.
  - Undeclared variable
  - comment is not closed. Like these may be arises.

* Run time errors: Else detected and displayed by the computer during the execution of a program.
- A runtime error occurs when the program directs the computer to perform an illegal operation, such as dividing a number by zero.

* Program with a runtime error:

```
#include <stdio.h>
int main (void)
{
    int first, second;
    double temp, ans;
    printf("Enter two integers>");
    scanf("%d %d", &first, &second);
    temp = second / first;
    ans = first / temp;
    printf("The result is %.3f \n", ans);
    return(0);
}
```

o/p: Enter two integers> 14 3
Arithmetic fault, divide by zero at line of routine main

- Logic errors: occurs when a program follows faulty algorithm.
- Because logic errors usually do not cause run-time errors and do not display error messages, they are very difficult to detect.
- The only sign of a logic error may be incorrect program output.
- One can detect logic errors by testing the program thoroughly.

Library functions:

Predefined functions and Code Reuse:

- Code reuse, reusing program fragments that have already been written and tested whenever possible, is one way to accomplish goal to write error-free code.
- 'C' promotes reuse by providing many predefined functions that can be used to perform mathematical computations.

eg.

```
y = sqrt(x);
```

(function call; function name: sqrt; argument: x)

'C' library functions

Table:

Function	standard header file	Purpose: Example	Arguments	Result
abs(x)	<stdlib.h>	Returns the absolute value of its integer argument. if x is -5, abs(x) is 5	int	int
ceil(x)	<math.h>	Returns the smallest integral value that is not less than x: if x is 45.23, ceil(x) is 46.0	double	double
exp(x)	<math.h>	Returns e^x where e = 2.71828. if x is 1.0, exp(x) is 2.71828	double	double
sqrt(x)	<math.h>	Returns the nonnegative square root of x ($\sqrt{x}$) for $x \geq 0.0$ if x is 2.25, sqrt(x) is 1.5	double	double

Function prototype (Function without Arguments)

FORM: ftype fname(void);

Example: void draw_circle(void);

Interpretation: The identifier fname is declared to be the name of a function. The identifier ftype specifies the data type of the function result.

Batch version of Miles-to-kilometers conversion program.

```
#include <stdio.h>  /* printf, scanf definitions */
#define KMS_PER_MILE 1.609  /* conversion constant */
int main (void)
{
   double miles, kms;  /* distance in miles & equivalent distance in kms */
   /* Get and echo the distance in miles */
   scanf ("%lf", &miles);
   printf("The distance in miles is %.2f. \n", miles);
   /* Convert the distance to kilometers */
   kms = KMS_PER_MILE * miles;
   /* Display the distance in kilometers. */
   printf("That equals %.2f kilometers. \n", kms);
   return (0);
}
```

o/p: The distance in miles is 112.00
That equals 180.21 kilometers.

Errors: Syntax errors, Runtime errors, & Logic errors.

Example for logic errors:

```
#include <stdio.h>
int main(void)
{
   int first, second, sum;
   printf("Enter two integers> ");
   scanf("%d %d", first, second);  /* Error! should be &first, &second */
   sum = first + second;
   printf(" %d + %d = %d \n", first, second, sum);
   return(0);
}
```

o/p Enter two integers> 14 3
5971289 + 5971297 = 11942586.

Function Definitions:

Fig. Function draw_circle

```
/*
 * Draw a circle
 */
void
draw_circle(void)
{
    printf("    *    \n");
    printf("  *   *  \n");
    printf("  *  *  \n");
}
```

Syntax:

```
ftype fname(void)
{ local declarations
  executable statements
}
```

Examples:

```
/*
 * Displays a block_letter H
 */
void
print_H(void)
{
    printf("**   **\n");
    printf("**   **\n");
    printf("** *** \n");
    printf("**   **\n");
    printf("**   **\n");
}
```

```
printf("  /\\  \n");  /* use 2 \'s to print 1 */
```

Functions with input arguments:

Input arguments: arguments used to pass information into a function subprogram.

Output arguments: arguments used to return results to the calling function

eg. rim_area = find_area(edge_radius) - find_area(hole_area);

Function definition (Input Arguments and single Result)

Syntax:

```
function interface comment
ftype fname (formal parameter declaration list)
{
    local variable declarations
    executable statements
}
```

Example:

```
/* finds the cube of its argument. Pre : n is defined. */
int
cube(int n)
{
  return (n * n * n);
}
```

Function with multiple arguments:

```
scale(2.5, 2)
```

In function scale, statement

```
scale_factor = pow(10, n);
```

Introduction to Computer Graphics

text mode: A display mode in which a C program displays only characters.

graphics mode: A display mode in which a C program draws graphics patterns and shapes in an output window.

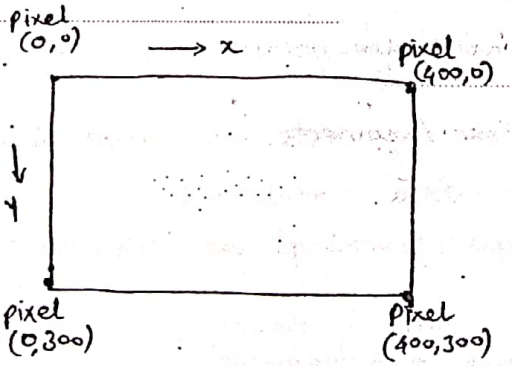
pixel : A picture element on a computer screen.

Some Graphics functions

The function getmaxwidth and getmaxheight return the position of the position of the last pixel in the X and Y-directions on your computer screen:

some typical screen dimensions are: 640 X 480 for 14 inch screens & 1366 X 768 for 15 inch screens. Include <graphics.h>

```
bigx = getmaxwidth();    /* get the largest x-coordinate */
bigy = getmaxheight();   /* get largest y-coordinate */
initwindow(bigx, bigy, "Full screen window - press a character to close window
```



Background color and Foreground color

```
setbkcolor(GREEN);
setcolor(RED);      /* RED is the foreground color */
```

Drawing Rectangles

rectangle (x_1, y_1, x_2, y_2); draws a rectangle that has one diagonal with end points (x_1, y_1) and (x_2, y_2)

Control Structures:

Control structures control the flow of execution in a program or function.
Instructions are organized into three kinds of control structures to control execution flow: sequence, selection and repetition.

Compound statements:→ a group of statements bracketed by { and } that are executed sequentially.

Selection control structure:→ a control structure that chooses among alternative program statements.

Conditions: condition is an expression that is either false (represented by 0) or true (usually represented by 1).

Relational and Equality operators

Operators	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality.

Logical operators

with three logical operators, && (and), || (or), ! (not), we can form logical expressions.

eg: salary < MIN_SALARY || dependents > 5
temperature > 90.0 && humidity > 0.90
n >= 0 && n <= 100
⇒ 0 <= n && n <= 100

Logical complement: (negation)

The complement of a condition has the value 1 (true) when the condition's value is 0 (false); the complement of a condition has the value 0 (false), when the condition's value is non-zero (true).

Table: The && operator (and)

Operand 1	Operand 2	Operand 1 && operand 2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	0 (false)
0 (false)	nonzero (true)	0 (false)
0 (false)	0 (false)	0 (false)

Table: The || operator (or)

Operand 1	Operand 2	Operand 1 || operand 2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	1 (true)
0 (false)	nonzero (true)	1 (true)
0 (false)	0 (false)	0 (false)

Table: The ! operator (not)

operand 1	!operand 1
nonzero (true)	0 (false)
0 (false)	1 (true)

Logical assignment:

senior_citizen = (age >= 65);

The condition in parenthesis is first evaluated.

logical operators can be applied.

!senior_citizen

is 1 (true) if the value of age less than 65.

Complementing the condition

item == SENT

are

! (item == SENT)

The If statement

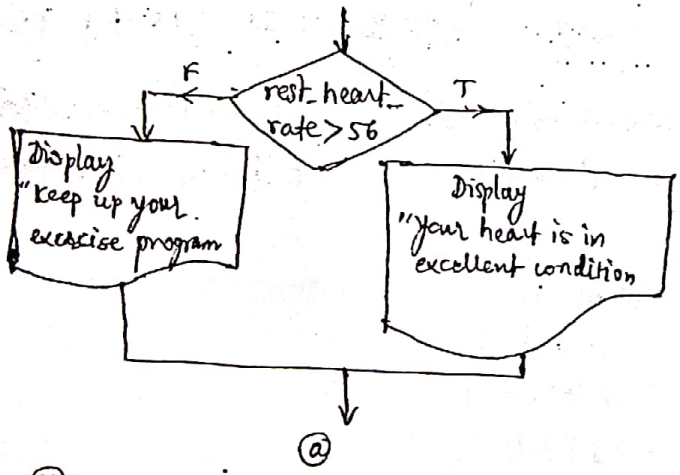
The if statement is the primary selection control structure.

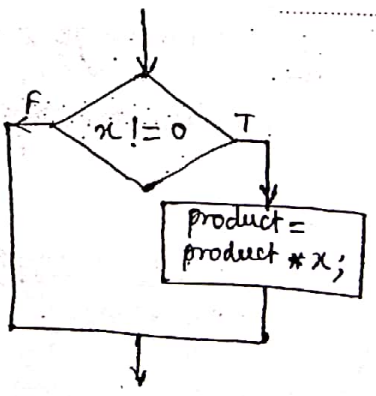
if statement with two ~~arguments~~ alternatives:

The if statement

```
if (rest_heart_rate > 56)
    printf("keep up your exercise program! \n");
else
    printf("Your heart is in excellent health :\n");
```

Flowcharts of if statements with ⓐ Two alternatives and ⓑ One alternative





Operator Precedence

Unary operator : an operator that has one operand.

- ! (not), + (plus sign), - (minus sign), and & (address of), which have a single operand are evaluated second and function calls are evaluated first in given table:

Table Operator precedence

Operators	Precedence
function calls	highest
! + - & (unary operators)	
* / %	
+ -	
< <= > >=	
== !=	
&&	
||	
=	lowest

writing English Conditions in 'C'

English Condition	logical Expression	Evaluation
x and y are greater than z	x>z && y>z.	1 && 1 is 1 (true)
x is equal to 1.0 or 3.0	x==1.0 || x==3.0	0 || 1 is 1 (true)
x is in the range z to y, inclusive	z<=x && x<=y	1 && 1 is 1 (true)
x is outside the range z to y	!(z<=x && x<=y)	!(1 && 1) is 0 (false)
	z>x || x>y	0 || 0 is 0 (false)

character comparison:

Expression	Value
'9' >= '0'	1 (true)
'a' < 'e'	1 (true)
'B' <= 'A'	0 (false)
'z' == 'z'	0 (false)
'a' <= 'A'	system dependent

if statement (one alternative)

Form: if (condition)
statement$_T$;

Example: if (x > 0.0)
pos_prod = pos_prod * x;

Interpretation: If condition evaluates to true (a nonzero value), then statement$_T$ is executed; otherwise, statement$_T$ is skipped.

if Statement (Two Alternatives)

Form: if (condition)
statement$_T$;
else
statement$_F$;

Example:
```
if (x >= 0.0)
    printf("Positive \n");
else
    printf("negative \n");
```

Interpretation: If condition evaluates to true (a nonzero value), then statement$_T$ is executed and statement$_F$ is skipped; otherwise, statement$_T$ is skipped and statement$_F$ is executed.

Nested if Statements and Multiple-Alternative Decisions:

nested if statement an if statement with another if statement as its true task or its false task.

```
if (x>0)
    num_pos = num_pos+1 ;
else
    if (x<0)
        num_neg = num_neg +1 ;
    else    /* x equals 0 */
        num_zero = num_zero +1 ;
```

Multiple alternative Decision

Syntax:
```
if (condition1)
    statement1
else if (condition2)
    statement2
    :
else if (conditionn)
    statementn
else
    statemente
```

Example:
```
/* increment num_pos, num_neg, or num_zero
   depending on x */
if (x>0)
    num_pos = num_pos +1 ;
else if (x<0)
    num_neg = num_neg +1 ;
else  /* x equals 0 */
    num_zero = num_zero +1 ;
```

The switch statement

- The switch statement may also be used in C to select one of several alternatives.
- The switch statement is useful when the selection is based on the value of a single variable or of a simple expression (called the controlling expression)

Program using a switch statement for selection

```
/* Reads serial number and displays class of ship */
#include<stdio.h>
int main (void)
{
    char class;    /* input - character indicating class of ship */
    /* Read first character of serial number */
    printf("Enter ship serial number> ");
    scanf("%c", &class);    /* scan first letter */
    /* Display first character followed by ship's class */
    printf("Ship class is %c:", class);
    switch (class) {
    case 'B':
    case 'b':
            printf("Battleship\n");
            break;
    case 'C':
    case 'c':
            printf("Cruiser\n");
            break;
    case 'D':
    case 'd':
            printf("Distroyer\n");
            break;
    case 'F':
    case 'f':
            printf("Frigate\n");
            break;
    default:
            printf("Unknown\n");
    }
    return(0);
}
```

o/p: sample run 1

```
Enter ship serial number> f3451
ship class is f: Frigate
```

sample run 2

```
Enter ship serial number> p210
ship class is p: Unknown
```

Repetition and Loop Statements:

Loop: a control structure that repeats a group of steps in a program.

Three C loop control statements: while, for and do-while.

Repetitions in programs

Loop body: the statements that are repeated in the loop.

Comparison of loop kinds

Kind	When used	C implementation structures
Counting loop	known in advance how many loop repetitions will be needed to solve the problem.	while, for
Sentinal-controlled loop	Input of a list of data of any length ended by a special value.	while, for
Endfile-controlled loop	Input of a single list of data of any length from a data file.	while, for
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered.	do-while
General conditional loop	Repeated processing of data until a desired condition is met	while, for

Counting loops and the while statement

Counter-controlled loop (counting loop) a loop whose required number of iterations can be determined before loop execution begins.

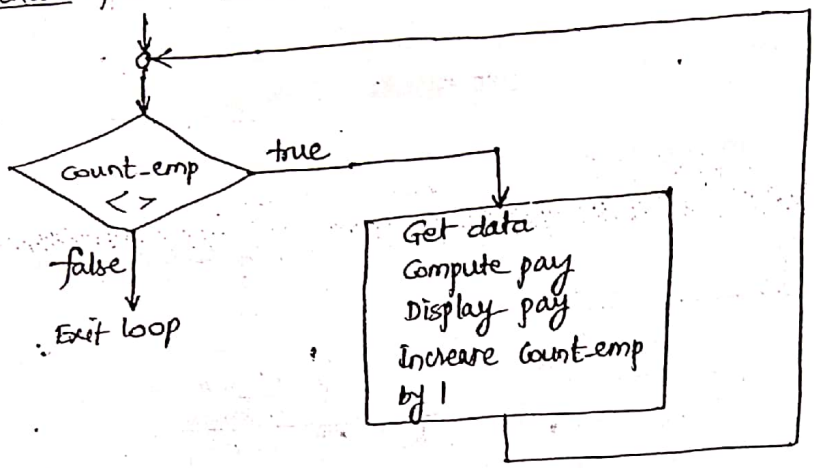
A counter controlled loop follows this general format.

Set loop control variable to an initial value of 0.
while loop control variable < final value

(23) Increase loop control variable by 1.

The while statement:

Flowchart for a while loop



variable count-emp is called the loop control variable.

- The loop control variable count-emp must be 1) Initialization., 2) tested, 3) Updated properly for the loop to execute.

* Initialization: count-emp is set to an initial value of 0 (initialized to zero) before the while statement is reached.
* Testing: count-emp is tested before the start of each loop repetition (called an iteration or a pass).
* Updating: count-emp is updated (its value increased by 1) during each iteration.

Infinite loop: a loop that executes forever.

while statement

Syntax:
```
while (loop repetition condition)
    statement ;
```

Example:
```
/* Display N asterisks */
count_star = 0;
while (count_star < N) {
    printf("*");
    count_star = count_star + 1;
}
```

Computing a Sum or Product in a loop

loops often accumulate a sum or a product by repeating an addition or multiplication operation as given in example.

accumulator: a variable used to store a value being computed in increments during the execution of a loop.

in the loop body, the assignment statement

```
total_pay = total_pay + pay;   /* add next pay */
```

Compound Assignment Operators:

- assignment statements of the form

```
variable = variable op expression;
```

- where op is C arithmetic operator. These include increments and decrements of loop counters.

```
count_emp = count_emp + 1;
time = time + 1;
time = time - 1;
```

Compound Assignment Operators

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
count_emp = count_emp + 1;	count_emp += 1;
time = time - 1;	time -= 1;
n = n * (x+1);	n *= x+1;
product = product * item	product *= item;

The for statement: just like other loop structure initialization, testing and updating loop control variable

- A for loop has important feature that it supplies a designated place for each of these three components.

Example:

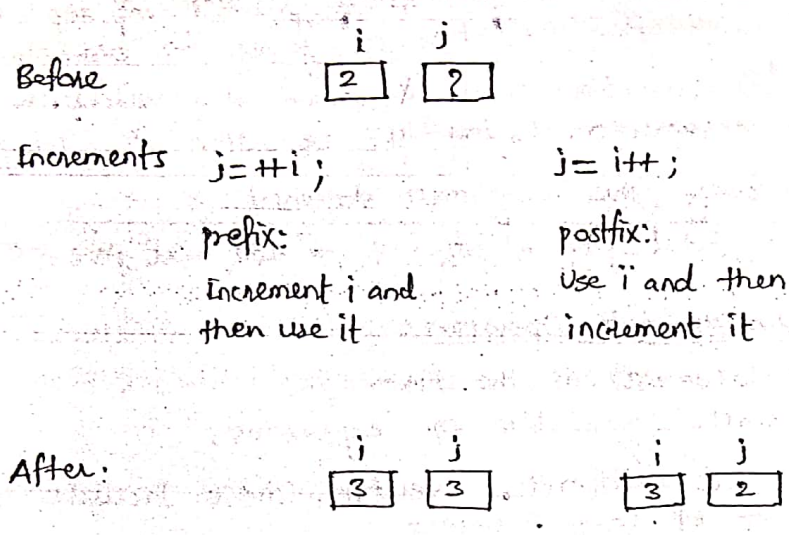
```
for (count_emp = 0;                    /* initialization */
     count_emp < number_emp;           /* loop repetation condition*/
     count_emp += 1)                   /* update */
```

Syntax:

```
for (initialization expression;
     loop repetition condition;
     update expression)
     statement;
```

```
/* Display N asterisks */
for (count_star = 0;
     count_star < N;
     count_star += 1)
   printf("*");
```

Comparison of Prefix and Postfix Increments



Conditional loops:

conditional loop typically has three parts that control repetition: initialization, testing of a loop repetition condition, and an update.

Example using while loop:

```
printf ("Enter number of observed values>");
scanf("%d", &num_obs);          /* Initialization */
while (num_obs < 0) {
    printf("Negative number invalid; try again>");
    scanf ("%d", &num_obs);     /* Update        */
}
```

Sentinel-Controlled loops

sentinal value: an end marker that follows the last item in a list of data

eg. sentinel-controlled while loop

```
/* compute the sum of a list of exam scores. */
#include<stdio.h>
#define SENTINEL -99

int
main(void)
{   int sum = 0, score;
/* Accumulate sum of all scores. */
  printf("Enter first score (or %d to quit)>", SENTINEL);
  scanf("%d", &score)        /* Get first score */
    while (score != SENTINEL) {
            sum += score;
      printf("Enter next score (%d to quit)>", SENTINEL);
      scanf("%d", &score);   /* Get next score */
    }
  printf("\n Sum of exam scores is %d \n", sum
return(0); }
```

```
/* Illustrates a pair of nested counting loops */
#include <stdio.h>
int main (void)
{
  int i,j;    /* loop control variables */
  printf("        i    j \n");  /* Print column labels */
  for (i=1 ; i<4; ++i)          /* Heading of outer loop */
  { printf("Outer %6d \n", i);
    for(j=0; j<i ; ++j)         /* Heading of inner loop */
    {
      printf("Inner %9d \n", j);
    } /* end of inner loop */
  }          /* end of outer loop */
 return (0);
}
```

o/p.

```
          i    j
outer     1
  inner        0
outer     2
  inner        0
  inner        1
outer     3
  inner        0
  inner        1
  inner        2
```

* Do while statement and Flag Controlled loops

The loop

```
do {
    printf( "Enter a letter from A through E> ");
    scanf("%c", &letter_choice);
   }
   while (letter_choice < 'A' || letter_choice > 'E');
```

do while statement

Syntax:

```
do
    statement;
while (loop repetition condition);
```

Example:

```
/* Find first even number input */
do
    status = scanf("%d", &num);
while (status > 0 && (num % 2)!=0);
```

break statement: (let us C by Yashwant)

- When break is encountered inside any loop, control automatically passes to the first statement after the loop.
- A break is usually associated with if.

continue statement: (let us C by Yashwant)

- When continue is encountered inside any loop, control automatically passes to the beginning of the loop.

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

o/p.

```
12
21
```

Function to compute factorial

```
/* compute n!
 * Pre: n is greater than or equal to zero.*/
int factorial (int n)
 {
    int i;        /* local variable */
      product; /* accumulator for product computation */
    product = 1;
   /* Compute the product n x (n-1) x ... x 2 x 1 */
for (i=n ; i>1 ;--i)
{
   product = product * i ;
}
/* Returns function result */
return (product);
}
```

Endfile controlled loop

- A data file is always terminated by an endfile character that can be detected by the scanf function.
- The pseudo-code for an endfile-controlled loop.
  1. Get the first data value and save input status.
  2. while input status does not indicate that end of file has been reached
  3. Process data value
  4. Get next data value and save input status.

eg. ~~input_status != EOF~~

input_status != EOF

(Let us C Yashwant)

goto statements

- goto statement to take the control where you want.
- when we use goto we do not sure how we got to a certain point in our code.
- goto can always be avoided.

Bitwise Operators

(Let us C Yashwant)

- The bitwise operators permit the programmer to access and manipulate individual bits within the piece of data.
- The various bitwise operators available in 'C' are:

Operator	Meaning
~	one's complement
>>	Right shift
<<	left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (Exclusive OR)

- These operators can operate upon ints and chars but not on floats and doubles.
- During discussion on bitwise operators function showbits() will be use.
- It displays binary representation of any integer or characters value.

One's Complement Operator

- one's complement operator is represented by the symbol ~.

Eg.

```
main()
{
  int j,k;
  for (j=0 ; j<=3 ; j++)
    {
      printf("\n Decimal %d is same as binary ",j);
      showbits (j);
      k= ~j ;
      printf("\n One's complement of %d is ",j);
      showbits(k);
    }
}
```

o/p. Decimal 0 is same as binary 00000000000000000
One's complement of 0 is 11111111111111111

Right shift operator:

- The right shift operator is represented by >>.
- It needs two operands. zero is added on the left side.
- It shifts each bit in its left operand to the right.
- The number of places the bits are shifted depends on the number following the operator (i.e. its right operand).
- Thus, ch>>3 would shift all bits in ch three places to the right.

eg. k= i>>3 ;

Left shift Operator

- This is similar to the right shift operator, the only difference being that the bits are shifted to the left; and for each bit shifted, a 0 is added to the right of the number.

Bitwise AND operator

- This operator is represented as &.
- It is different that && (logical AND operator).
- Operates on two operands.
- Compared on bit-by-bit basis.
- Hence both operands must be of the same type (either char or int)
- The second operand is often called an AND mask.

Truth table:

&	0	1
0	0	0
1	0	1

- The best use of the AND operator is to check whether a particular bit of an operand is ON or OFF.
- It is used to turn OFF a particular bit in a number.

Bitwise OR Operator

- The rules that govern the value of the resulting bit obtained after ORing of two bits is shown in truth table.

|	0	1
0	0	1
1	1	1

```
11010000   Original bit pattern
00000111   OR mask
--------
11010111   Resulting bit pattern
```

Bitwise XOR Operator

- The XOR operator is represented as ^ and is also called an Exclusive OR operator.
- The XOR returns 1 only if one of the two bits is 1.
- The truth table:

^	0	1
0	0	1
1	1	0

- XOR operator is used to toggle a bit ON or OFF.

Conditional operator

- The conditional operator ? : takes three operands.

```
c ? r1 : r2
```

- The evaluation could be expressed in pseudo code as.

```
if c
    result value is r1
else
    result value is r2
```

- It can be used to define a macro. to find the minimum of two values,

```
# define MIN (x,y) ((x) <= (y)) ? (x) : (y))
```

Sequential Operator (comma operator)

- The comma operator evaluates its two operands in sequence, yielding the value of the second operand as the value of the expression.
- The value of the first operand is discarded.

The effect of assignment statement

```
x= (i+=2 , a[i]);
```

is the same as the effect of these two statements.

```
i+=2;
x= a[i];
```

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

- Remember that all elements of any given array must be of the same type.

Eg.

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

Statements that manipulate Array x

Statement	Explanation
printf("%.1f", x[0]);	Displays the value of x[0].
x[3] = 25.0;	stores the value 25.0 in x[3]
sum = x[0] + x[1];	stores the sum of x[0] and x[1],
sum += x[2];	Adds x[2] to sum.

Array Declaration:

int marks[30];

- Here int specifies the type of variable.
- marks specifies the name of the variable.
- [30] tells how many elements of the type int will be in our array.

Accessing Elements of Array

- All the elements in array are numbered, starting with 0.
- Thus marks[2] is not the second element of the array, but the third.

Entering Data into an Array

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

Reading Data from an array

eg.
```
for(i=0; i<=29; i++)
  { sum = sum + marks[i];

avg = sum/30;
printf("\n Average marks = %d", avg);
```

- An array is a collection of similar elements.
- The first number in an array is numbered 0. so the last element is 1 less than the size of the array.
- An array is also known as a subscripted variable.
- Before using array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory location.

Array Initialization:

Eg.
```
int num[6] = {2,4,12,5,45,5};
int n[] = {2,4,12,5,45,5};
float press[] = {12.3,34.2,-25.4,-11.3};
```

Array Elements in memory

eg int arr[8];

- it occupies 16 bytes of memory.
- In case of (Windows/Linux) 32 bytes of memory is allocated as int take 4 bytes.

Bound checking

- There is no checking in C to see if the subscript used for an array exceeds the size of the array.
- Data entered with subscript exceeding ~~memory~~ array size will simply be placed in memory outside the array.

Passing array elements to a function

- Array elements can be passed to a function by calling the function by value, or by reference.
- In the call by value we pass values of array elements to the function, whereas in the call by reference we pass the addresses of array elements to the function.

eg.
```
/* Demonstration of call by value */
main()
{
    int i;
    int marks[] = {55,65,75,56,78,78,90};
    for (i=0; i<=6; i++)
        display (marks[i]);
}
display (int m)
{ printf("%d", m);
}
```

o/p. 55 65 75 56 78 78 90

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

o/p. 55 65 75 56 78 78 90

~~Pointer and Array~~ pointers and Arrays

Every time a pointer is incremented it points to the immediately next location of its type.

- When the integer pointer x is incremented, it points to an address two locations after the current location, since an int is always 2 bytes long. (under Windows / Linux since int is 4 bytes long, next value of x would be 65528).
- similarly, y points to an address 4 locations after the current location. (→ its float) and z points to 1 location after the current location. (→ its a char)
- The way pointer can be incremented, it can be decremented as well, to point to earlier locations.

What is function?

- Function is a self-contained block of statements that perform a coherent task of some kind.
- we will be focusing on a function that calls or activates the function and the function itself.

```
main()
  {
    message();
    printf("\n Cry, and you stop the monotony!");
  }
message()
  {
    printf("\n Smile, and the world smiles with you...");
  }
```

o/p:

Smile, and the world smiles with you...

Cry, and you stop the monotony!

Here main() itself is a function and through it we are calling the function message();

- when main() calls the function message() the control passes to the function message().
- The main() is temporarily suspended; it falls asleep while the message() function wakes up and goes to work.
- When the message() function runs out of statements to execute, the control returns to main().

- Thus main() becomes the 'calling' function, whereas message() becomes the 'called' function.

some conclusions

- Any C program contains at least one function.
- If a program contains only one function, it must be main().
- If a C program contains more than one function, then one (and only one) of those functions must be main(), because program execution always begins with main().
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in main().
- After each function has done its thing, control returns to main().
- When main() runs out of function calls, the program ends.

Eg.

```
main()
{
  printf("\n I am in main");
  italy();
  printf("\n I am finally in main");
}
italy()
{
  printf("\n I am in Italy");
  brazil();
  printf("\n I am back in italy");
}
brazil()
{
  printf("\n I am in brazil");
  argentina();
}
argentina()
{
  printf("\n I am in argentina");
}
```

op. I am in main
I am in Italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main

— main() calls other functions.

Why use function?

Why not squeeze the entire logic into one function, main()?

Two reasons:

(a) Writing functions avoids rewriting the same code over and over.

(b) Using functions it becomes easier to write programs and keep track of what they are doing.

Passing values between Functions

- The arguments are sometimes also called 'parameters'.
- The following program given as a, b, c as input parameters.
- The output of sum of a, b, and c is given.
- The calculation of sum is done in a different function called calsum().

```
/* Sending and receiving values between functions */
main()
  {
    int a,b,c,sum;
    printf("\n Enter any three numbers");
    scanf("%d %d %d", &a, &b, &c);
    sum = calsum(a,b,c);
    printf("\n Sum = %d", sum);
  }
calsum(x,y,z)
int x,y,z;
  {
     int d;
     d= x+y+z;
     return(d);
  }
```

And here is the output...

```
Enter any three numbers 10 20 30
sum = 60
```

The return statement serves two purposes:

1) On executing the return statement it immediately transfers the control back to the calling program.
2) It returns the value present in the parentheses after return, to the calling program.

Scope rule of functions:→

Looking at the following program,

```
main()
 {
   int i=20;
   display(i);
 }
display(int j)
 {
   int k=35;
   printf("\n%d",j);
   printf("\n%d",k);
 }
```

* Call by value and call by reference:

- Whenever we called a function and passed something to it we have always passed the 'values' of variables to the called functions.
- such function calls are called 'calls by value".
- The example of call by value are shown below:

```
sum = calsum(a,b,c);
f = factr(a);
```

- If we can pass the location number (also called address) we can say it as 'call by reference'

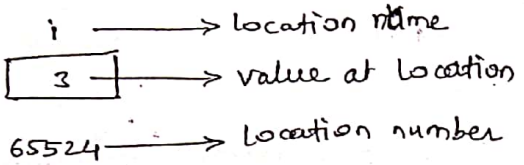
Introduction to pointers:

Pointer Notation:

Consider the declaration,

```
int i=3;
```

- The declaration tells the C ~~program~~ compiler to:
  a) Reserve space in memory to hold the integer value.
  b) Associate the name i with this memory location.
  c) Store the value 3 at this location.



we can print this address number through the following program:

```
main()
{
    int i=3;
    printf("\n Address of i = %u", &i);
    printf("\n Value of i = %d", i);
}
```

The output of the above program would be:

Address of i = 65524
Value of i = 3

- '&' used in the above statement is C's 'address of' operator.
- the expression &i returns the address of the variable i, which in this case happens to be 65524.
- The other pointer operator available in C is '*', called 'value at address' operator.
- It gives the value stored at a particular address.
- The 'value at address' operator is also called 'indirection' operator.

```
main()
{
    int i=3;
    printf("\n Address of i = %u", &i);
    printf("\n Value of i = %d", i);
    printf("\n value of i = %d", *(&i));
}
```

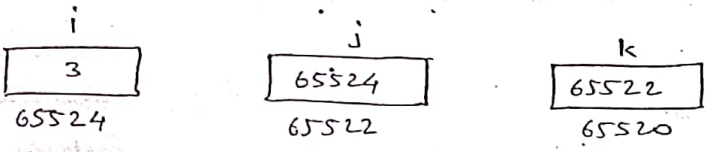
the O/p.

Address of i = 65524
Value of i = 3
Value of i = 3

- The value of *(&i) is same as printing the value of i.

```
int *alpha;
char *ch;
float *s;
```

- Here alpha, ch and s are declared as pointer variables. i.e. variables capable of holding addresses.
- Remember that, addresses (location nos) are always going to be whole numbers.
- pointers will always contain whole numbers.
- the declaration float *s does not mean that s is going to contain a floating-point value.
- It means that s is going to contain the address of a floating-point value.



- int i, *j, **k;
- Here i is an ordinary int, j is a pointer to an int (often called an integer pointer), whereas k is a pointer to an integer pointer.

Back to function calls

Two types of function calls
ⓐ sending the values of the arguments (call by value)
ⓑ sending the addresses of the arguments (call by reference)

Eg. Call by reference:
- The addresses of actual arguments in the calling function are copied into formal arguments of the called function.

```
eg.  main()
     { int a=10, b=20;
       swapr(&a, &b);
       printf("\n a= %d b= %d", a, b);
     }
     swapr(int *x, int *y)
     {
       int t;
       t= *x;
       *x= *y;
       *y= t;
     }
```

o/p: a=20 b=10

Recursion:

- In C, it is possible for the functions to call themselves.
- A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular function'

Eg.

```
main()
{
   int a, fact;
   printf("\n Enter any number");
   scanf("%d", &a);
   fact = rec(a);
   printf("Factorial value = %d", fact);
}
rec(int x)
{
   int f;
   if (x==1)
     return(1);
   else
     f= x * rec(x-1);
   return(f);
}
```

o/p

```
Enter any number 1
Factorial value=1
Enter any number 2
Factorial value= 2
Enter any number 3
Factorial value= 6
Enter any number 5
Factorial value =120
```

Structures

Declaring a structure

- The following statement declares the structure type:

```
struct book
{
  char name;
  float price;
  int pages;
};
```

- This statement defines a new data type called struct book.
- Once the new structure data type has been defined one or more variables can be declared to be of that type.
- Variables b1, b2, b3 can be declared to be the type struct book, as

```
struct book b1, b2, b3;
```

- we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
{
  char name;
  float price;
  int pages;
};
struct book b1, b2, b3;
```

Accessing structure Elements:

- Structure uses a dot (.) operator.

b1.pages and for price b1.price

How structure elements are stored:

- Structure are always stored in contiguous memory locations.

The following program will illustrate:

```
/* memory map of structure elements */
main()
{
  struct book
  { char name;
    float price;
    int pages;
  }
  struct book b1 = { 'B', 130.00, 550 };
```

```
printf("\n Address of name = %u", &bl.name);
printf("\n Address of price = %u", &bl.price);
printf("\n Address of pages = %u", &bl.pages);
}
```

o/p:

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Array of structures:

Following program shows how to use an array of structures:

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

Uses of Structures:

- The immediate application of structure comes to mind is Database management
- They can be used for a variety of purposes like:

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

Union Types

- Unions are derived data types, the way structures are.
- Demo of union working.

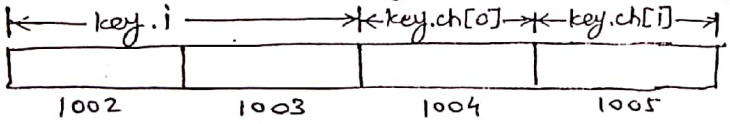
```
#include <stdio.h>
void main()
{
    union a
    {
      int i;
      char ch[2];
    };
    union a key;
  key.i = 512;
  printf("\n key.i = %d", key.i);
  printf("\n key.ch[0] = %d", key.ch[0]);
  printf("\n key.ch[1] = %d", key.ch[1]);
}
```

O/p: key.i = 512
key.ch[0] = 0
key.ch[1] = 2

- Structure and Unions are accessed exactly the same way.

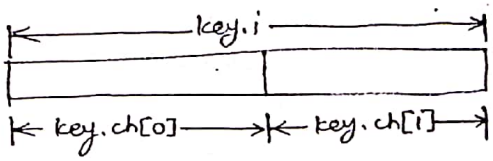
```
struct a
{
  int i;
  char ch[2];
};
struct a key;
```

- This data type would occupy 4 bytes in memory, 2 for key.i and one each for key.ch[0] and key.ch[1], as shown in fig.



Now for a union.

```
union a
{
  int i;
  char ch[2];
};
union a key;
```



- union occupy only 2 bytes in memory.
- we can access the two bytes simultaneously (by using key.i) or the same two bytes individually (using key.ch[0] and key.ch[1].

Union of ... Example of ...

Union of Structures

Example of structures nested in unions:

```
#include <stdio.h>
void main()
{
    struct a
    {
      int i;
      char c[2];
    };
    struct b
    {
      int j;
      char d[2];
    };
    union z
    { struct a key;
      struct b data;
    }
    union z strange;
    strange.key.i = 512;
    strange.data.d[0] = 0;
    strange.data.d[1] = 32;
    printf("\n %d", strange.key.i);
    printf("\n %d", strange.data.j);
    printf("\n %d", strange.key.c[0]);
    printf("\n %d", strange.data.d[0]);
    printf("\n %d", strange.key.c[1]);
    printf("\n %d", strange.data.d[1]);
}
```

And o/p:

```
512
512
0
0
32
32
```

Strings: A group of characters can be stored in a character array.
- character arrays are many a time also called strings.
- A string constant is a one-dimensional array of characters terminated by a null ('\0').

For example:

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

- Last character of string will always '\0'
- '\0' is called null character.
- ASCII value of '\0' is 0.
- whereas ASCII value of 0 is 48.

- String can be initialized as;

```
char name[] = "HASLER";
```

- In this declaration '\0' is not necessary.
- C inserts the null character automatically.

```
/* program to demonstrate printing of a string */
main()
{
    char name[] = "Klinsman";
    int i=0;
    while (i<=7)
    {
        printf("%c", name[i]);
        i++;
    }
}
```

o/p. Klinsman

Pointers and Strings

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

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

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

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

- Also, once a string has been defined it cannot be initialized to another set of characters.
- Unlike strings, such an operation is perfectly valid with char pointers.

```
main()
{
    char str1[] = "Hello";
    char *p = "Hello";
    str1 = "Bye"; /* error */
    p = "Bye"; /* works */
}
```

Standard Library String Functions

- There are list of more commonly used functions along with their purpose.

Functions	Use
strlen	Find length of a string
strlwr	Converts a string to lowercase.
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another.
strcpy	Copies a string into another.
strcmp	Compares two strings.
strncmp	Compares first n characters of two strings.
strdup	Duplicates a string
strrev	Reverse string

strlen()

- This function counts the number of characters present in a string.
- Its usage is illustrated in the following program.

```
main()
{
    char arr[] = "Bamboozled";
    int len1, len2;

    len1 = strlen(arr);
    len2 = strlen("Humpty Dumpty");
    printf("\n string = %s length = %d", arr, len1);
    printf("\n string = %s length = %d", "Humpty Dumpty", len2);
}
```

The output

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

pointers

The declaration

```
float *p;
```

- identifies p as a pointer variable of type "pointer to float".
- This means that we can store the memory address of a type float variable in p.

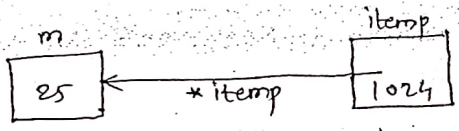
pointer type declaration:

Syntax: type *variable;

Example: float *p;

Interpretation: The value of the pointer variable p is a memory address. A data item whose address is stored in this variable must be of the specified type.

Indirect Reference:
Accessing the contents of a memory cell through a pointer variable that stores its address.



- It applies the unary address of operator & to variable m to get its address. which is then stored in itemp

```
itemp = &m;      /* store address of m in pointer itemp */
```

- In above fig. shows that we can use *itemp to reference the cell selected by pointer itemp.
- When the unary indirection operator * is applied to a pointer variable, it has the effect of following the pointer referenced by its operand.
- This provides an indirect reference to the cell that is selected by the pointer variable.

pointer to files

- C allows a program to explicitely name a file from which the program will take input and ~~output~~. write output.
- To use that we must declare pointer variables of type FILE *.
- The statements

```
FILE *inp;   /* pointer to input file */
FILE *outp;  /* pointer to output file */
```

- Before permitting access OS must prepare a file for input and output
- This preparation is the purpose of the calls to function fopen
  statements.

```
inp = fopen("distxt.txt", "r");
outp = fopen("distout.txt", "w");
```

- the next two statements demonstrate the use of the functions fscanf and fprintf, file equivalents of functions scanf and printf.

```
fscanf(inp, "%.1f", &item);
fprintf(outp, "%.2f\n", item);
```

- when program has no further use for its input and output files, it closes them by calling function fclose with the file pointers.

```
fclose(inp);
fclose(outp);
```

Program using file pointers

```
#include <stdio.h>
int main (void)
{
    FILE *inp;
    FILE *outp;
```

- fopen() returns the address of this structure, which we ~~called~~ ~~as~~ have collected in the structure pointer called fp.
- we declared fp as

```
FILE *fp;
```

- The file structure has been defined in the header file, "stdio.h".

Reading from a file:

- To read the file's contents from memory there exists a function called fgetc().
- This has been used in our program as,

```
ch = fgetc (fp);
```

- fgetc() reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable ch.

Trouble in Opening file:

- It may possible that when we try to open a file using the function fopen(), the file may not be opened.
- fopen() may fail due to a number of reasons like, disk space may be insufficient to open a new file, or the disk may be write protected. or the disk is damaged and so on.
- Here is how this can be handled in a program.

```
#include <stdio.h>
#Include "stdio.h"
main()
{
    FILE *fp;
    fp = fopen("PR1.C", "r");
    if (fp == NULL)
    {
        puts ("cannot open file");
        exit();
    }
}
```

Closing the file:

- This is done using f. fclose() through the statement:

```
fclose(fp);
```

- once we close the file we cannot read it using getc() unless we reopen the file.

File Input / Output

- Sometimes it is necessary to store the data in a manner that can later retrieved and displayed either in part or in whole. This medium is usually a 'file' on the disk.

File Operations:

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

These are:

ⓐ Creation of a new file
ⓑ Opening an existing file.
ⓒ Reading from a file.
ⓓ Writing to a file,
ⓔ Moving to a specific location in a file (seeking)
ⓕ closing a file.

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

Opening a file

- Before we can read (or write) information from (to) a file on a disk we must open the file.
- To open a file we must call the function fopen();
- It would open a file "PRI.C" in "read" mode.
- "r" is a string and not a character, hence the double quotes and not a single quotes.
- fopen() performs three important tasks. when you open the file in "r" mode
  ⓐ Firstly it searches on the disk the file to be opened.
  ⓑ Then it loads the file from the disk into a place in memory called buffer
  ⓒ It sets up a character pointer that points to the first character of the buffer.

- we can use fputc() to write characters into the file. The characters would get written to the buffer.
- When we close this file using fclose() three operations would be performed:
  - (a) The characters in the buffer would be written to the file on the disk.
  - (b) At the end of the file a character with ASCII value 26 would get written.
  - (c) The buffer would be eliminated from memory.

Homework: (a) Counting characters, tabs, spaces, ...
(b) A file copy program.
(c) writing to a file.

* File opening modes:

- The task performed by fopen() when a file is opened in each of these modes are also mentioned.

"r" - Searches file. If the file opened successfully fopen() loads it into a memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen() returns NULL.

Operations possible :- reading from the file.

"w" - searches file. If the file exists, its contents are overwritten. If the file doesnot exist, a new file is created. Returns NULL if unable to open file.

Operations possible :- writing to the file.

"a" - Searches file. If the file opened successfully fopen() loads it into memory. and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible :- adding new contents at the end of file.

"r+" - Operations possible: reading existing contents, writing new contents, modifying existing contents of the file.

"w+" - operations possible: writing new contents, reading them back and modifying existing contents of the file.

"a+" - operations possible: reading existing contents, appending new contents to end of file. Cannot modify existing contents.

Concept of Macros and preprocessor commands in C.

- Preprocessor processes our source program before it is passed to the compiler.
- The preprocessor offers several features called preprocessor directives.
- Each preprocessor directives begin with a # symbol.
- The directives can be placed anywhere in a program but most often placed at the beginning of a program, before the first function definition.
- The following preprocessor directives are:
  ⓐ Macro expansion, ⓑ File inclusion. ⓒ Conditional Compilation
  ⓓ Miscellaneous directives.

Macro expansion:

```
#define UPPER 25
main()
{
    int i;
    for(i=1; i<= UPPER ; i++)
    printf("\n%d", i);
}
```

```
#define UPPER 25
```

This statement is called "macro definition" or more commonly 'macro'.

- Above macro is simple macro.

Macros with arguments:

Macros can have arguments, just as functions can.

```
#define AREA(x) (3.14 * x * x)
main()
{
    float r1 = 6.25, r2 = 2.5, a;
    a = AREA(r1);
    printf("\n Area of circle = %f", a);
    a = AREA(r2);
    printf("\n Area of circle = %f", a);
}
```

O/p Area of circle = 122.656250
Area of circle = 19.625000

File Inclusion:

- This directive causes one file to be included in another.
- The preprocessor command for file inclusion looks like this;

  #include "filename"
- If you have large program, the code is best divided into several different files, each containing a set of related functions.
- These files are ~~#included~~ #included at the beginning of main program file.
- The prototypes of all the library functions are grouped into different categories and then stored in different header files.
- Mathematics related functions are stored in header file 'math.h', console input/output functions are stored in 'conio.h'

  #include "filename"

  #include <filename>

#include "goto.c": This command would look for the file goto.c in the current directory as well as the specified list of directories as mentioned in the include-search path that might have been set up.

#include <goto.c>: This command would look for the file goto.c in the specified list of directories only.

Conditional Compilation:

```
#ifdef macroname
        statement 1;
        statement 2;
        statement 3;
#endif
```

- If macroname has been defined, the block of code will be processed as usual; otherwise not.

where #ifdef would be useful?

(a) to "comment out" obsolete lines of code..

(b) to make programs portable;

#if and #elif Directives:

- The #if directive can be used to test whether an expression evaluates to a nonzero value or not.
- If the result of the expression is nonzero, then subsequent lines upto a

#else, #elif or #endif are compiled, otherwise they are skipped

Example:

```
main()
{
  #if TEST <=5
    statement 1;
    statement 2;
    statement 3;
  #else
    statement 4;
    statement 5;
    statement 6;
  #endif
}
```

Miscellaneous directives:

#pragma directive

- This directive is another special-purpose directive that you can use to turn on or off certain features.

Ex. @ #pragma startup and #pragma exit:

These directives allow us to specify the functions that are called upon program startup (before main()) or program exit (just before the program terminates).

C: Data types Revisited

To fully define variables one has to mention not only its 'type' but also its 'storage class'.

Storage class tells us:

ⓐ where the variable would be stored.
ⓑ what will be the initial value of the variable, if not specified.
ⓒ what is the scope of the variable, i.e. in which functions value will be available
ⓓ what is the life of the variable, i.e. how long it will exist.

There are four storage classes:

ⓐ Automatic storage class
ⓑ Register storage class,
ⓒ static storage class,
ⓓ External storage class.

ⓐ Automatic storage class:

The features of a variable defined to have an automatic storage class are as under:

storage	–	Memory
Default initial value	–	An unpredictable value, which is often called a garbage value.
scope	–	local to the block in which the variable is defin
life	–	Till the control remains within the block in which the variable is defined.

ⓑ Register storage class

The features of a variable defined to be a register storage class as under:

storage	–	CPU registers
Default initial value	–	Garbage value.
scope	–	local to the block in which the variable is defined.
life	–	Till the control remains within the block

© Static storage class:

The features of a variable defined to have a static storage class are as under:

storage	–	Memory
Default initial value	–	zero
scope	–	local to the block in which the variable is defined.
life	–	value of the variable persists between different function calls.

(d) External storage class:

The features of a variable whose storage class has been defined as external are as follows:

storage	–	Memory
Default initial value	–	zero.
scope	–	Global
life	–	As long as the program execution doesn't come to an end.

SORTING AND SEARCHING FUNCTIONS

Function	Use
bsearch	performs binary search.
lfind	performs linear search for a given value.
qsort	performs quick sort.

→ The selection sort is fairly intuitive (but not very efficient) sorting algorithm.

→ To perform a selection sort of ~~any~~ an array with n elements (subscripts 0 through n-1), we locate the smallest element in the array and then switch the smallest element with the ~~el~~ element at subscript 0, thereby placing the smallest element in the first position.

→ Then we locate the smallest element remaining in the subarray with subscripts 1 through n-1 and switch it with the element at subscript 1, thereby placing the second smallest element in the second position, and so on.

Algorithm for selection sort:

1. for each value of fill from 0 to n-2
2. Find index_of_min, the index of the smallest element in the unsorted subarray list [fill] through list [n-1].
3. If fill is not the position of the smallest element (index_of_min)
4. Exchange the smallest element with the one at position fill.

[0]	[1]	[2]	[3]
74	45	83	16

fill is 0. Find the smallest element in subarray list [1] through list [3] and swap it with list [0].

[0]	[1]	[2]	[3]
16	45	83	74

fill is 1. Find the smallest element in subarray list [1] through list [3] - no exchange needed.

[0]	[1]	[2]	[3]
16	45	83	74

fill is 2. Find the smallest element in subarray list [2] through list [3] and swap it with list [2].

[0]	[1]	[2]	[3]
16	45	74	83

Fig. Trace of selection sort.

Introduction to Object Oriented Programming

Key concepts :: objects, classes, data abstraction, Encapsulation, Inheritance, Polymorphism, Dynamic binding, Message passing, overloading, constructor, destructors.

Basic concepts of Object Oriented programming :

ⓐ objects, ⓑ classes, ⓒ Data abstraction and encapsulation, ⓓ Inheritance, ⓔ Polymorphism, ⓕ Dynamic binding, ⓖ Message passing.

Objects : Objects are basic runtime entities,

- They may represent, a person, a place, a bank account, a table of data or any item.

classes : A class is a collection of objects of similar type.

Eg. Mango, apple, and orange are members of class fruit.

- They are user defined data types and behave like the built-in types of programming language.

eg. fruit mango;

will create an object mango to the class fruit.

Data abstraction and Encapsulation :

Abstraction refers to the act of representing essential features without including the background details or explanations.

Algorithm for Binary search

1. Let bottom be the subscript of the initial array element.
2. Let top be the subscript of the last array element,
3. Let found be false.
4. Repeat as long as bottom isn't greater than top and the target has not been found.
5. Let middle be the subscript of the element halfway between bottom and top.
6. if the element at middle is the target
7. Set found to true and index to middle.
   else if the element at middle is larger than the target.
8. Let top be middle - 1
   else
9. Let bottom be middle + 1.

The bubble sort:

- The bubble sort is another technique for sorting an array. ~~After the~~
- A bubble sort compares adjacent array elements and exchanges their values if they are out of order.
- In this way, the smaller values "bubbles" to the top of the array (toward element 0), while the larger values sink to the bottom of array.
- After the first pass of a bubble sort, the last array element is in the correct position; after the second pass the last two elements are correct, and so on.
- Thus, after each pass, the unsorted portion of the array contains one less element.
- ~~xxxxxxxxxxxxxxxxxx~~