Sub code: OCS752

Sub Name: Introduction to C Programming

UNIT-1

Introduction

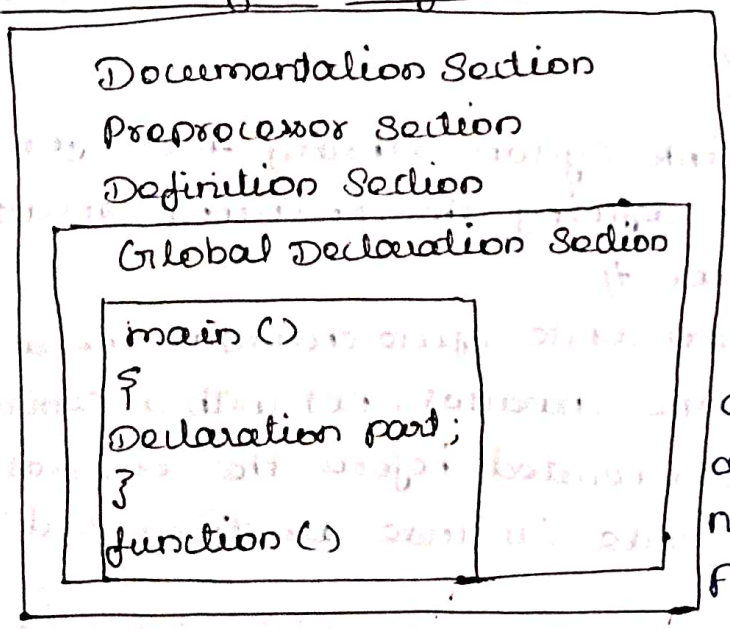
Structure of C Program – Basics : Data Types – constants – Variables – Keywords – Operators : Procedence and Associativity – Expressions – Input/output statements, Assignment statements – Decision-making statements – switch statement – Looping statements – Pre-processor directives – Compilation process – Exercise programs : check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers.

Applications of c Language

- Operating system - The first operating system to be developed using a high level programming language was UNIX which was designed in the C programming language.
- Embedded Systems
- GUI
- New Programming platforms
- Google
- Mozila firefox and Thunderbird
- MySQL
- Compiler Design

1. System Programming
   To implement os and embedded systems applications
2. Compilers, libraries and interpreters of other languages are implemented in C
3. Used as intermediate language
4. Used to implement end-user applications.

Structure of c Program



c program composed of 3 important sections
- Preprocessor directives
- Global Declarations
- Functions.

Here Preprocessor directives & Global declarations are optional ie., may or may not present in c program. But Functions are mandatory

c Program

```
#include <stdio.h>
main()
{
Printf ("Welcome to the world of c");
}
```

A c program is made up of functions

1. Documentation Section

It consists of a set of comment lines. It is used to specify the name of the program etc. It helps the readers to understand the code clearly.

This is a Non-executable statements which is not processed by the compiler.

Type of comments

Single-line command

Multi-line comment

Single line comment

'//' is used to comment a single line. It is automatically terminated with end of the line.

Eg. // Adding 2 numbers

Multiline Comment

It starts with '/*' and terminates with '*/'. This type is used when multiple lines of text are to be commented. Statements lie within the characters are commented.

Eg. /* _____
_____
_____ */

2. Preprocessor Section

This section is used to link system library files for defining the macros and for defining the conditional inclusion.

- Start with a pound symbol '#'
- # should be the first non white space character in a line
- terminate with a new line character, not with a semicolon;

Preprocessor directives are executed before the compiles compiles the source code. The code (include directive) will be required to add.

Ex #include <stdio.h> - Preprocessor directive statement. It includes standard input/output header (.h) file. The file is to be included if standard input/output functions like Printf, Scanf are to be used in a program.

Global declaration Section

It is optional. The variables that are used in many functions are declared as global variables in this section. It declares the variables in outside of the main function.

Functions Section

This section is mandatory and must be present in a c program. The program can have one or more functions. A function named 'main' is always required 2 parts such as.

- Header of the function
- Body of the function

Header

```
main()
{
----
}
```

General form

```
[return type] function_name([arg_list])
{
}
```

Body of the function

Set of statements enclosed within curly brackets known as braces

Type of statements

- Non-executable statement // declaration statement
- Executable statement

First non-executable statement are present than executable statements are written.

Ex. // C Program - Addition of 2 numbers

```
#include<stdio.h>
#include<conio.h>
main()
{
int a, b, c;                    // Declaration
Printf("Enter a number:");
Scanf("%d", &a);
Printf("Enter b number:");
Scanf("%d", &b);
c=a+b;
Printf("\n The sum is: %d", c);
getch();
}
```

output

```
Enter a number: 5
Enter b number: 10
The sum is: 15
```

C Programming

Data types

The type of the data that are going to access within the Program. Data type is one of the most important attributes of an identifier.

It determines the possible values that an identifier can have the valid operations that can be applied on it.

Each data type may have predefined memory requirement and storage representation.

classification of Data type

- Basic data types (primitive data types)
- Derived data types
- User-defined data types

Basic Data types

S.NO	Data type	Keyword	
1.	character	char	→ single character
2.	Integer	int	→ numbers
3.	Single Precision floating point (32 bits)	float	
4.	Double-Precision floating point (64 bits)	double	
5.	No value available	void	→ null data type, has no arguements

Bytes occupied

Data type	Memory bytes	Range	control string	Example
int	2 bytes	-32,768 to +32,768	%d or %i	int a = 39;
char	1 byte	-128 to +128	%c	char a = 'n';
float	4 bytes	3.4E-38 to 3.4E+38	%f or %g	float f = 29.77;
double	8 bytes	1.7E-308 to 1.7E+308	%lf	double d = 29771770 76

Integer
- Signed
  - int
  - short int
  - long int
- unsigned
  - unsigned int
  - unsigned short int
  - unsigned long int

character
- char
- signed char
- unsigned char

Float
- float
- double
- long double

2. Derived data types

It is derived from basic data types

- Array type ex: char[], int[], etc
- Pointer type ex: char *, int *, etc
- Function type ex: int (int, int), float (int), etc

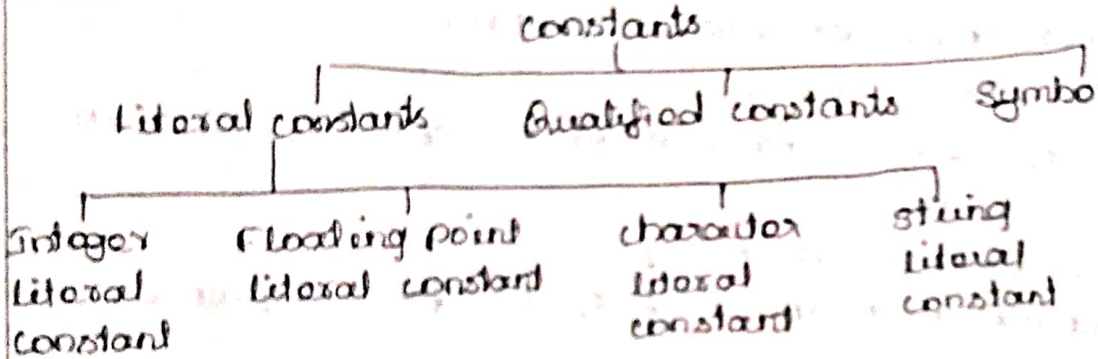
3. User defined data type

It provides flexibility to the user to create new data types. Newly created data types are called user-defined data types which is created by using structure, union, Enumeration

Constants

A constant is an entity whose value remains the same throughout the execution of the program. It can be placed on the right side of the assignment operator.

Classification



Literal constants

It denotes a fixed value. It may be an integer, floating point number, character or a string. It determined by its value.

Types

1. literal Integer literal constant

Integer values like -1, 2, 8 etc.

Rules

- An integer literal constant must have atleast one digit
- It should not have any decimal point
- It can be either positive or negative
- No special characters and blank spaces are allowed within an integer literal constant

2. Floating point literal constant

Floating values like -23.1, 10.8 etc. It can be written in a fractional form or in an exponential form.

Rules - Fractional form

- It must have atleast one digit
- It should have a decimal point

- It can be either positive or negative
- No special characters and blank spaces are allowed within a constant
- type double ex: 23.45

Rule - exponential form

mantissa part / exponential part

ex: 2e10 ie) 2×10^{10}

25E12 25e-12

Character literal constant

It can have one or at most two characters enclosed within single quotes ex 'A', 'a', '\n'

classification

- Printable character literal constant
- Non printable character literal constant

Printable character literal constant

All characters of source character set except ?, \, and \n

ex: 'b', 'A', '#'

Non printable character literal constant

It is represented with the help of escape sequences

Escape sequences:

It consists of \ (back slash) followed by the character and both enclosed within a single quotes. It is treated as a single character.

ex \', \", \?, \\, \a, \b, \f, \n, \r, \t, \v, \0

4. String literal constant

It consists of a sequence of characters enclosed within double quotes. It is terminated by a null character ('\0')

Ex. "ABC"

Number of bytes occupied -> number of character + 1

(double quotes) " " occupied -> one byte

length of the string -> number of characters.

Qualified constant

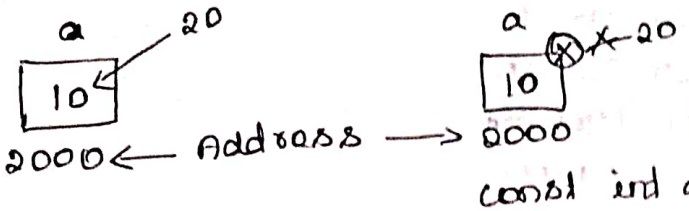
It is created by using 'const' qualifier.

Ex. const char a = 'A';

↑ qualified constant

int a=10 -> allocates 4 bytes to 'a' & initialize 10.
It is possible to modify the value of a

const int a=10 -> The qualifier places a lock on the box after placing the value in it. It is possible to see but not possible to modify.



int a=10

const int a=10;

Symbolic constant

It is created with the help of the define preprocessor directive. ex. # define PI 3.141124

Symbolic constant

It is replaced by its actual value during the preprocessing stage.

Enumeration constant

It is a list of constant integer value.

enum boolean (NO, YES);

First name in an enum has value 0, the next 1 and so on unless explicit values are specified.

```
enum escapes {BELL='\a', BACKSPACE='\b',
              TAB='\t', NEWLINE='\n',
              VTAB='\v', RETURN='\r'};
enum months {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
        /* FEB is 2, MAR is 3 etc */
```

Defined in 2 ways

i) enum week {Mon, Tue, Wed};
enum week day;

ii) enum week {Mon, Tue, Wed} day;

- user defined data type
- used to assign names to integral constants

Ex:
```
enum week {Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{enum week day;
 day = wed
 Printf("%d", day);
 return 0;
}
```

O/P
2

Keywords

C has a set of 32 reserved words known as Keywords. Th having fixed meaning and cannot be used as an identifier All Keywords must be written in lowercase letters.

List of Keywords

1. auto	9. while	17. if	25. Static
2. break	10. double	18. int	26. Strict
3. case	11. else	19. long	27. Switch
4. char	12. enum	20. register	28. typedef
5. const	13. extern	21. return	29. union
6. continue	14. float	22. Short	30. unsigned
7. default	15. for	23. Signed	31. void
8. do	16. go to	24. Size of	32. Volatile

Operators

Precedence and Associativity

operators

C language supports a lot of operators to be used in expressi An operator specifies the operation to be applied to its constants

Ex. a = Printf ("Hello") + 2

3 operators

function call operator ()

arithmetic operator +

assignment operator =

classification of operators

* Based on number of operands it is classified into 3 types

- Unary operator - operates on only one operand Ex. -3 // unary minus
- Binary operator - operates on 2 operands Ex. 2-3 // binary minus
- Ternary operator - operates on 3 operands Ex. ? : conditional operator

<< , == , && , &

* Based on Role of operators it is classified into 11.

- Arithmetic operators
- Relational
- Equality
- Logical
- Unary
- conditional
- Bitwise operators
- Assignment operators
- comma operators
- Size of operators
- operator Precedence chart

1 Arithmetic operators

Arithmetic operations like addition, subtraction, multiplication, division etc can be performed by using arithmetic operators

operator	name
+	unary plus, Addition
-	unary minus, Subtraction
++	Increment
--	Decrement
*	Multiplication
/	Division
%	Modulus

3 modes - Binary Arithmetic

- Integer mode → both operands are integer type. Ex. 4/3
- floating point mode → both operands are floating point Ex. 4.0/3.0
- Mixed mode → one is integer type & another is floating point Ex. 4/3.0

Unary

- unary plus - appear only towards the left side of its operand
- unary minus - appear only towards the left side of its operand

Increment

- Pre-increment: ++a; The value of the operand is incremented first and it is used for evaluation
- Post-increment: a++; The value of the operand is used first and then it is incremented

Decrement

Pre-decrement --a; value is decremented first and then used for evaluation

Post-decrement a--; value is used for evaluation and then decremented.

Division operator (/)

It is used to find the quotient. Sign of the result is based on numerator and denominator

Modulus operator (%)

It is used to find the remainder. Operands must be of integer type. Ex. a = 8%3; Sign depends upon the numerator

2 Relational Operators

It is used to compare 2 quantities (operands). Six relational operators are there. No white space character inserted in between 2 symbols. Result should be boolean constant i.e. 0 or 1 (0: false, 1: true)

operator	Name
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	Equal to
!=	Not equal to

Three modes - Binary arithmetic

a) Integer mode → both operands are of integer type eg. 4/3

b) floating point mode → both operands are of floating point type. eg. 4.0/3.0

c) Mixed mode → one is integer type and another is floating point type. eg. 4/3.0

unary plus - appear only towards the left side of its operand

unary minus - appear only towards the left side of its operand

Increment - Pre-increment : eg. ++a;
The value of the operand is incremented first and it is used for evaluation.

- Post-increment : eg. a++;
The value of the operand is used first and then it is incremented.

Decrement - Pre-decrement : eg. --a;
The value is decremented first and then used for evaluation.

- Post-decrement : eg. a--;
The value is used for evaluation and then decremented.

Division operator (/)

It is used to find the quotient. Sign of the result is based on numerator and denominator.

Modulus operator (%)

It is used to find the remainder. The operands must be of integer type. eg. a = 3 % 2;

Sign depends upon the numerator.

2. Relational operator

It is used to compare 2 quantities (operands). There are 6 relational operators such as

operator	Name
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	Equal to
!=	Not equal to

No white space characters are there in between 2 symbols. Result will be a boolean constant ie., 0 or 1 (0=false, 1=true)

An Expression that involves a relational operator forms a condition eg: a<b

3. Logical operator

It is used to logically relate the sub-expressions

operator	Name
!	logical NOT
&&	logical AND
||	logical OR

AND operation

operand1	operand2	Result
F	F	F
F	T	F
T	F	F
T	T	T

NOT operation

operand	Result
F	T
T	F

OR operation

operand1	operand2	Result
F	F	F
F	T	T
T	F	T
T	T	T

Expressions are evaluated left to right

eg: E1 && E2

E1 || E2

l = i && j++

4. Bitwise operators

6 operators are there for bit manipulation

operator	Name
~	Bitwise NOT
<<	left shift
>>	Right shift
&	Bitwise AND
^	Bitwise X-OR
|	Bitwise OR

It operates on the individual bits of the operands. It can be applied on operands of type char, short, int, long.

5. Assignment operator

A variable can be assigned a value by using an assignment operator

operator	Name
=	Simple assignment
*=	Assign product
/=	Assign quotients
%=	Assign modulus
+=	Assign sum
-=	Assign difference
&=	Assign bitwise AND
|=	Assign bitwise OR
^=	Assign bitwise XOR
<<=	Assign left shift
>>=	Assign Right shift

General form

operand1 operator = operand2 Ex. a/=2 => a = a/2

⇓

op1 = op1 op op2

x = a + b

x = 10

No white space between the symbols.

6. Miscellaneous operators

other operators

- Function call operator – ( )
- Array subscript operator – [ ]
- Member select operator
  - * Direct member access operator -> . (dot)
  - * Indirect member access operator -> -> (arrow)
- Indirection operator *
- conditional operator
- comma operator
- size of operator
- Address of operator

a. conditional operator

It is a ternary operator.

operator	Name
? :	conditional operator

General form

E1 ? E2 : E3 Ex: a*b ? a : b

Sub exprs

E1 is evaluated first. If E1 is true, E2 is evaluated and E3 is ignored. if E1 is false, E3 is evaluated and E2 is ignored

b. comma operator

It is used to join multiple expressions together.

operator	Name
,	comma operator

form: E1, E2, E3, . . , En

c. Size of operator

It is used to determine the size in bytes, which a value or a data object will take in memory.

operator	Name
Sizeof	Size of operator

General form

Sizeof expression

Sizeof (expression)

Sizeof (type_name)

Ex. Sizeof (int)

Sizeof (int *)

Sizeof (char)

Ex Sizeof 2

Sizeof (a)

Sizeof (2+3)

The type of result of evaluation of the size of operator is int. The operand of size of operator is not evaluated. It cannot be applied on operands of incomplete type or function call.

d. Address of operator

It is used to find the address

operator	Name
&	Address - of operator

Syntax

& operand
└ variable or function

It cannot be applied to constants, expressions and variables with register storage class.

Associativity

If operators of the same precedence appear together then the operator are evaluated according to their associativity

- left to right
- right to left

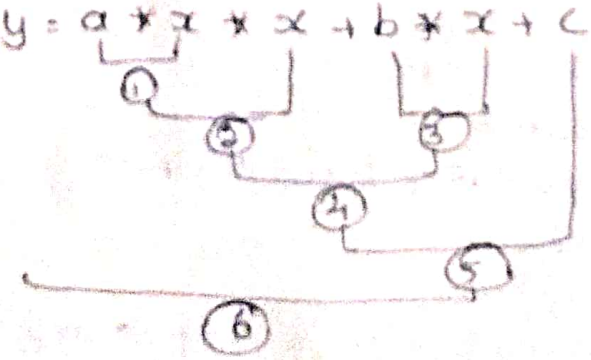
Ex: 2 * 3 / 5
2 * 3 = 6
6 / 5

Associativity of operators determines the order in which operators of equal precedence are evaluated when they occur in the same expression. It defines the direction left to right or right to left in which the operator acts upon its operands.

L to R	R to L
() [] . ++ --	++ , --
* / %. + -	! ~ + - & *
<< >>	?:
<<= > >=	= += -= *= /= %=
== !=	> >= <<= &= ^= !=
&	
^	
|	
&&	
||	
,	

Ex

① y = a * x * x + b * x + c



② z = p * r % q + w / x - y

6 1 2 4 3 5

Expressions

An expression is made up of one or more operands and operators that specify the operations to be performed on operands. An expression is a sequence of operands and operators that specifies the computation of a value.

Ex `a = 2 + 3`

operands = 3 => a, 2, 3

operators = 2 => =, +

operands

An operand specifies an entity on which an operation is to be performed. An operand can be a variable name, a constant a function call or a macro name.

Ex: `a = Printf ("Hello") + 2`

- `a` → variable name
- `=` → Assignment operator
- `Printf ("Hello")` → function call
- `+` → Arithmetic operator
- `2` → constant

operators

An operator specifies the operation to be applied to its constant

Ex: `a = Printf ("Hello") + 2`

classification of Expressions:

- Simple Expressions

  An Expression that has only one operator. Ex: a + 2

- compound Expressions

  An expression that involves more than one operator

  Ex: 2 + 3 * 5

  Determine the order in which operators will operate

  It depends on the precedence & associativity of operators

```
b = 2 + 3 * 5
        ~~~
         15        * -> highest precedence
b =  2 + 15        +
b =    17          =
```

Input/output statements

- Streams
- Formatting input/output
- Printf()
- Scanf()
- Examples of printf/scanf
- Detecting Errors during Data input

Streams

A sequence of bytes of data

Types

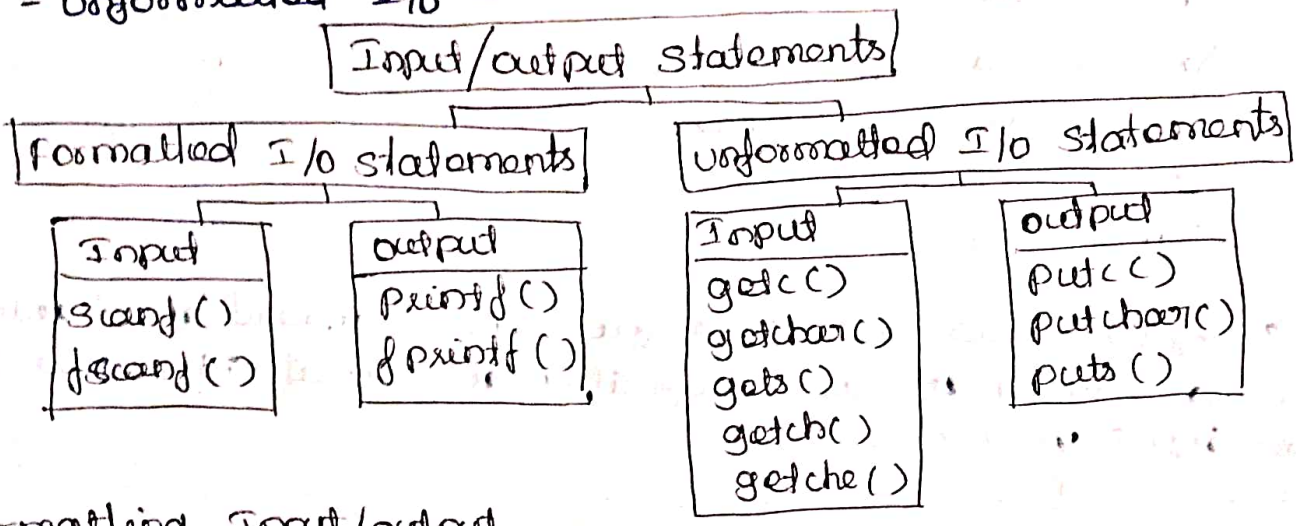
- Input stream - A sequence of bytes flowing into a program
- output stream - A sequence of bytes flowing out of a pgm

Modes of Streams

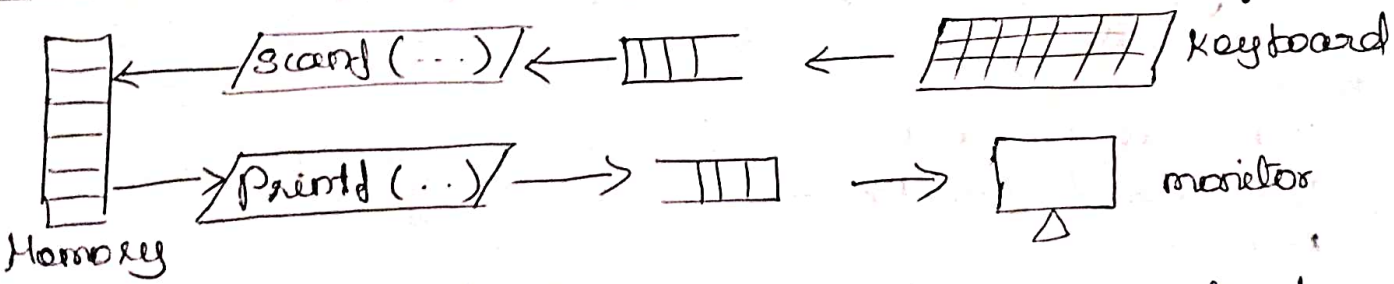
- Text Stream - only characters
- Binary stream - Any sort of data

Types of Input/output statements

- Formatted I/O statements
- unformatted I/O statements



Formatting Input/output



When input and output is required in a specified format the standard library functions are used

- Printf
- scanf

Printf()

Printf() function allows the user to output data of different data types on the console in a specified format. It translates internal values to characters.

Output data can be written from the computer to standard output device using printf() function.

```
Printf("control string", var1, var2, ..., varn);
```

control string = required formatting specifications enclosed with in double quotes.

- Different control strings are used based on data type

Printf() converts formats and prints its arguments on the standard output.

Types of objects

- ordinary characters -> copied to the output stream
- conversion specifications -> conversion & printing begins with a "%"

control code	Action
\b	backspace
\f	form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\'	Single quote
\0	Null.

Scanf()

Scanf() reads characters from the standard input interprets them according to the specification in format. It is used to read input values.

General form

Scanf ("control string", var1_addr, var2_addr, ..., varn_addr);

Ex: Scanf ("%d %f", &a, &b);

Ex:1

```
#include <stdio.h>
#include <conio.h>
main()
{
char name[25];
puts("Enter the name");
gets(name);
puts("\n Print the name");
puts(name);
getch();
}
```

Assignment statement

Assigning the value to a variable using assignment operator is known as an assignment statement

Syntax

Variable = constant / variable / Expression

operator is "="

Stores a value in the memory location which is denoted by a variable name

```
                                  total
int total;                        [ ? ]
total = (1+2) * 4;                [ 12 ]  total
```

The expression on the right hand side of the assignment statement can be

- Arithmetic expression
- Logical expression
- Relational expression
- Mixed expression

Ex:
```
int a;
float b, c, avg, t;
avg = (b+c)/2 ;    // Arithmetic exps
a = b && c ;       // Logical exps
a = (b+c) && (b+c) // mixed exps
```

Multiple Assignment statement

A single value is assigned to 2 or more variables

syntax: `var1 = var2 = ... = varn = value / Expression`

Ex:

i) `m = n = 3`

ii) `a = b = (c*c + d*d)/2`

Ex:
```
#include <stdio.h>
main()
{
int a=b=10, c, d, e=25;
c=a+b;
d=(c+b)*a+e;
printf("%d %d", c, d);
}
```

output:
```
20 325
```

Decision making statements

Flow of program control

The order in which the program statements are executed

Branching statements

It is used to transfer the program control from one point to another

Branching

Selection — conditional branching
→ Program control is transferred from one point to another based on the condition

Jumping — unconditional branching
→ Program control is transferred from one point to another without checking any condition.

Selection statement

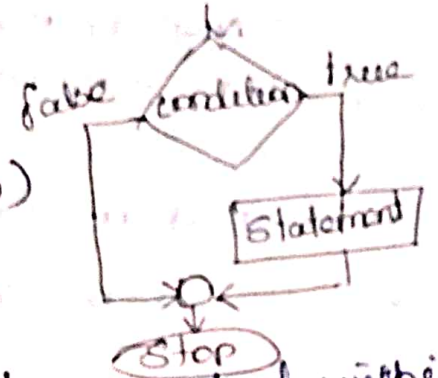
It is based upon the outcome of a particular condition, Selection statements transfer control from one point to another statements.

- if statement
- if-else statement
- if-else if statement
- Switch Statement

1. If statement

General form

```
if (expression)
    statement
```



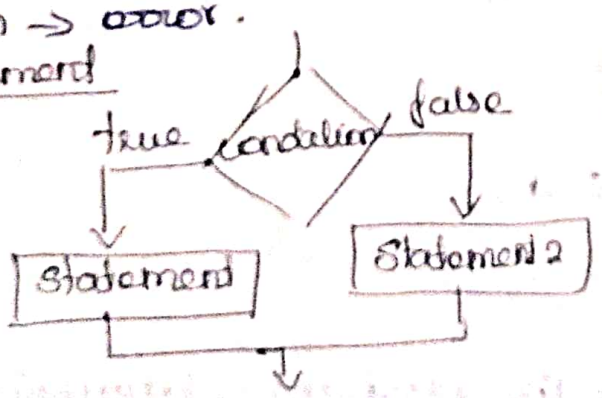
Ex:
```
if (n>=18)
Printf ("Eligible");
```

- The condition should be specified within parenthesis.
- condition is executed first.
- If the condition is true then the statement will be executed.
- If the condition is false the control will be directly transferred to the statements outside the braces.
- No semicolon should be placed at the end of if
  semicolon → error.

2. if else Statement

General form

```
if (expression
    statement 1
else
    statement 2
```



Ex:
```
if (a % 2 == 0)
Printf ("a is even");
else
Printf ("a is odd");
```

Set of actions to be performed if a particular condition is true and another set of actions to be performed if the condition is false.

Nested if statement

If statement contains another if statement is called nested if statement.

Stores a value in the memory location which is denoted by a variable name


```
int total;          total
                    [ ? ]
total = (1+2) * 4;  [ 12 ]  total
```

The expression on the right hand side of the assignment statement can be

- Arithmetic expression
- logical expression
- Relational expression
- Mixed expression

Ex:
```
int a;
float b, c, avg, t;
avg = (b+c)/2 ;    // Arithmetic expr.
a = b && c ;       // logical expr
a = (b+c) && (b+c) // mixed expr
```

Multiple Assignment statement

A single value is assigned to 2 or more variables.

syntax: var1 = var2 = ... = varn = value / Expression

Ex:

i) m = n = 3

ii) a = b = (c * c + d * d) / 2

Ex:
```
#include <stdio.h>
main()
{
int a=b=10, c, d, e=25;
c=a+b;
d=(c+b) * a + e ;
Printf("%d %d", c, d);
}
```

output:

20 325

Decision making statements

Flow of program control

The order in which the program statements are executed

Branching statements

It is used to transfer the program control from one point to another

General form

```
if (expression)
   if statement
```

Nesting can be done upto any level

```
if (expression 1)
   if (expression 2)
      if (expression n)
         statement
```

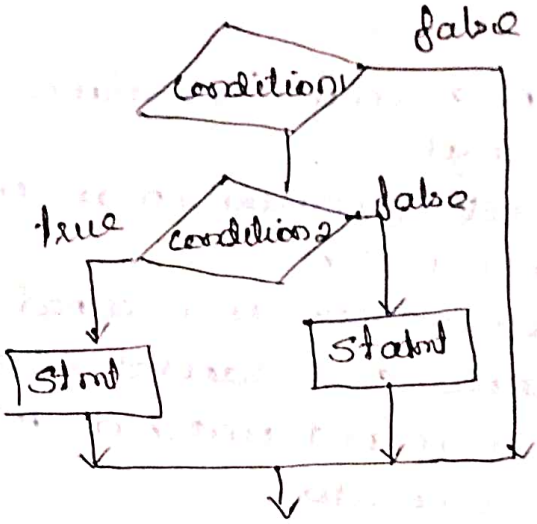
Nested if else statement

if else statement is or contains another if statement or if else statement.

dangling else problem -> more number of if than else

Syntax

```
if (condition 1)
 {
   if (condition 2)
    {
     true statement 2;
    }
    else
    {
     false statement 2;
    }
 }
else
 {
  false statement 1;
 }
```



3. if else ladder (if else if statement)

nested if & else if ladder

Syntax

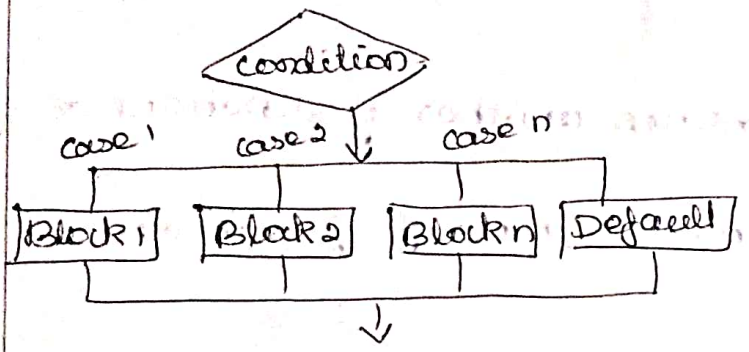
```
if (condition 1)
{
  statement 1;
}
else if (condition 2)
{
  statement 2;
}
else if (condition 3)
{
  statement 3;
}
else
{  default statement;
```

4. Switch statement

It is used to control complex branching operations. It is used for many conditions checking required. It provides an easy and organized way to select among multiple options depending upon the outcome of a particular condition.

General form

```
switch (expression)
  statement
```



Ex:

```
switch (expression or var)
{
 case 1:
    statements;
    break
   :
 case N:
    statements N;
    break;
 default:
    statement;
}
```

- The switch selection operation is evaluated
- The result of evaluation is compared with the case labels until there is a match.
  - * if expression is matched the execution starts from the matched case labeled statement
  - * if it is not matched then the default statements will be executed.

Jump statements

It transfers the control from one point to another without checking any condition i.e) unconditionally.

- goto statement
- break statement
- continue statement
- return statement

a. goto statement

It is used to branch unconditionally from one point to another within a function

General form

Forward jump

```
goto label;
...
...
...
label:
  statement;
```

→ some statements will be ignored.

Backward jump

```
label:
  statement
...
...
...
goto label;
```

→ some statements will be repeatedly executed.

b. break statement

It is used to terminate or to exit from a switch statement

General form

```
break;
```

c. continue statement

loop does not terminated

It is used to transfer the control to the begining of the loop

It is used within a while, do while, for loops.

General form

```
continue;
```

Ex:

```
for(i=1; i<=12; i++)
{
  if (i==6)
  continue;
  printf("\n\t %d ", i);
}
```

o/p

1
2
3
4
5
6
7
8
9
10
11
12

d. return statement

- without an expression -> can appear only in a function

Syntax

```
return;
```

- with an expression -> should not appear in a function

Syntax

```
return expression;
```

It terminates the execution of a function and returns the control to the calling function

Looping statements (Iteration statement)

Iteration - the process of repeating the same set of statements again and again until the specified condition holds true.

Computers execute the same set of statements again and again by putting them in a loop.

Three looping statements

- for statement
- while statement
- do-while statement

In general loops are classified as:

Counter controlled loops - number of iteration is known in advance.

Sentinal controlled loops - number of iteration is not known beforehand.

a) for statement

most popular one.

General form

```
for(initialization ; condition ; increament/decrement)
{
  statements ;
}
```

3 sections are separated by semicolon

initialization section

It is used to initialize the loop counter.

condition section

It tests the value of the loop counter i.e) determines whether the loop should continue or not.

Manipulation section / increasing or decreasing operation

It manipulates the value of the loop counter. So that the condition evaluates to false and the loop terminates. i.e) increasing or decreasing the value of the loop counter each time the program segment has been executed.

The for statement is not terminated with a semicolon. If it is terminated with a semicolon then the semicolon is interpreted as a null statement.

Execution

a) Initialization section is executed only once at the start of the loop.

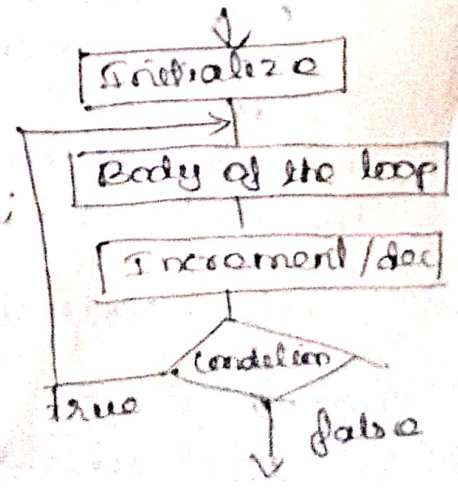
b) The expression present in the condition section is evaluated
   - if it evaluates to true, the body of the loop is executed
   - if it evaluates to false, the loop terminates and the program control is transferred to the statement present next to the for statement

c) After the execution of the body of the loop, the manipulation expression is evaluated.

- 3 steps represent the first iteration of the for loop.
- for the next iteration, steps b and c are repeated until the expression in step b evaluates to false.

Ex:

```
for (i=0; i<n; i++)
{
  Printf ("The numbers are %d", i);
}
```

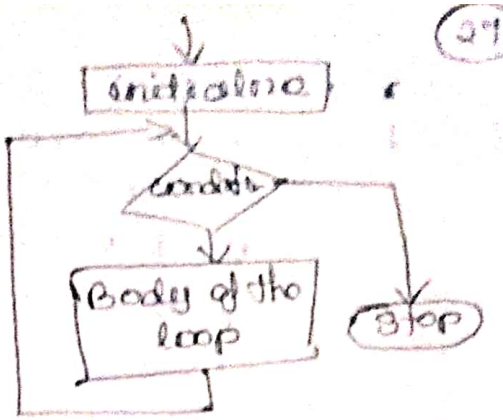


b. while statement

General form

```
while (expression)
statement
```

```
initialize loopcounter;
while (condition)
{
statements;
incre/decre;
}
```



Ex:

```
int a=1, sum=0;
while (a<=10)
{
printf ("%d", a);
sum = sum + a;
a++;
}
```

while statement should not be terminated with a semicolon. if it is terminated with a semicolon, it is treated as null statement

Execution

a) controlling expression is evaluated
   - if it evaluates to true, the body of the loop is exec
   - if it evaluates to false, the program control is transferred to the statement present next to the while statement

b) After executing the body the program control returns back to while statement.

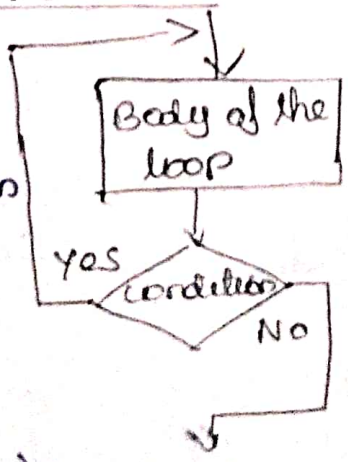
c) steps a & b are repeated until the condition evaluates to false.

- Initialize the loop counter before the while statement
- Manipulate the loop counter is inside the body of the while statement

c. do-while statement

General form

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



Ex:

```
do
{
Printf ("Enter 2 number");
Scanf ("%d %d", &a, &b);
Printf ("sum=%d", a+b);
Printf ("Do you want to
   continue say Y/N: ");
Scanf ("%c", &yes);
}
while ((yes==y);|(yes==Y);
```

* while is terminated with a semicolon.

Execution

a) The statement (body of the loop) is executed.

b) After the execution of body once, the condition is evaluated.

- if it evaluates to true, the body is executed again
- if it evaluates to false, the program control is transferred to the statement present next to the do-while statement

- Initialize the loop counter before do-while statement
- Manipulate (incre/decre) is inside the body of the loop
- The statement (body) is executed once, even when the do-while condition is initially false.

Preprocessor Directives

Introduction - Preprocessor

Before C program is compiled, the source code is processed by a program called preprocessor. The preprocessor directives starts with # symbol. It do not require a semicolon at the end.

Types of preprocessor directives

- Macros substitution directives
- File inclusion directives
- conditional compilation directives
- Miscellaneous directives

1. Macro substitution directives

Macro: a piece of code in a program which is given some name.

Whenever the name is encountered by the compiler, the compiler replaces the name with the actual piece of code
A macro is defined by using

#define

Types of macros

- object-like Macros
- Function-like Macros

a. object-like Macros

An identifier that is replaced by value.

#define x 25

b. Function-like Macros

like function call

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

2. File inclusion Directives

Tells the compiler to include a file in the source code pgm

Types of files

- Header file or standard file
- User defined files

Header files contains definition of pre defined functions like Printf(), Scanf() etc

```
#include <filename>
```

Ex #include <stdio.h>

b. User defined files

It is used to include some other files to your source pgm

```
#include "filename"
```

Ex #include "hello.c"

3. conditional compilation directives

Directives which help to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions

a. #if

It evaluates the expression or condition. If condition is true, it executes the code

Syntax:

```
#if exp
  // code
#endif
```

#else

It evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif and #nded directives.

Syntax with #if

```
#if exp
  // code
#else
  // else code
#end if
```

Syntax with #elif

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

#elif

It is used to write #else and #if in one statement

#ifdef

The macro with name as 'macro_name' is defined, then the block of statement will execute. If it is not defined, the compiler will simply skip the block of statements.

#endif -> specifies the end of the block.

Syntax

```
#ifdef macro_name
    stmt 1;
    stmt 2;
    ...
    stmt N;
#endif
```

#ifndef

It checks if macro is not defined by #define. If yes, it executes the code.

Syntax

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

4. Miscellaneous directives

#undef - To undefine a macro. It is used to cancel its definition

Ex: #define PI 3.1415
#undef PI

#Pragma - To provide additional information to the compiler

Compilation and Linking Processes

Executing a C Program steps

- Creating the program
- Compiling the program
- Linking the program with functions that are needed from the C library.
- Executing the Program

1. Creating the program

The program must be entered into a file. Save file with .c extension:

Ex hello.c

File is created with the help of text editor.

Compiling and Linking

2. Compilation

After the program is ready, it should be compiled ie) After the program is entered in C editor, the next is to compile the program.

Shortcut key: Alt+F9 or compile option in compile menu

The process of converting the high level language program

into machine understandable form. There is a possibility for errors. It still show errors. ie) Syntax errors. (not in proper syntax)

After the compilation, look for errors and warnings. warnings will not prevent the execution of the program. If there are errors, check the code properly.

There should be no typing mistake. If there is no error you can execute the program.

3. Linking

It is the process of putting together other program files and functions that are required by the program.

C language program is the collection of predefined function which is written in standard 'C' header files.

Before executing a 'c' program, need to link with system library. It can be done automatically at the time of execution

Ex. for exp() function
math library

4. Executing the program

It is the process of running and testing the program

Types of error

- Logical error
- Data error

Shortcut Key → ctrl + F9 or choose Run option from Run Menu.

Steps

- Enter the program in a C editor
- Save the program. File → save (or) F2. use the extension .c
- Compile the program (compile → compile (or) Alt + F9)
- Run the program (Run → Run (or) ctrl+F9)

Exercise Programs

1. check whether the required amount can be withdrawn based on the available amount

```
#include <Stdio.h>
unsigned long amount = 1000;
int pin = 9090;
int main()
{
int ch;
char transaction;
Printf("welcome to ATM Service \n");
while (1)
{
Printf("enter your secret PIN number: ");
Scanf("%d", &ch);
if (ch != pin)
{
Printf("Please enter a valid PIN\n");
Continue;
}
Printf("\n *** Main Menu ***
Printf("1. check Balance\n");
Printf("2. withdraw cash\n");
Printf("3. Deposit cash\n");
Printf("4. Quit\n");
Printf("*******\n");
Printf("Enter your choice: ");
Scanf("%d", &ch);
Switch (ch)
{
Case 1:
    Printf("\n Your Balance in Rs: %lu\n", amount
    break;
Case 2:
    unsigned long withdraw;
    Printf("\n Enter the amount to withdraw: ");
    Scanf("%lu", & withdraw);
    if (withdraw % 100 != 0)
    {
     Printf("\n Please enter the amount in multiples of 100\n");
    }
```

```
else if (withdraw > (amount-500))
{
 Printf ("\n Insufficient balance \n");
}
else
{
amount -= withdraw;
Printf ("\n Please collect cash \n");
Printf (" Your current balance is %lu \n", amount);
}
break;
case 3:
    unsigned long deposit;
    Printf ("\n Enter the amount to deposit: ");
    Scanf ("%lu", & deposit);
    amount += deposit;
    Printf ("your balance is %lu \n", amount);
    break;
case 4:
    Printf ("\n Do you want to continue? (y/n): ");
    fflush (stdin);   // clearing the input buffer
    Scanf ("%c", & transaction);
    if (transaction == 'n' || transaction == 'N')
    break;
 }
Printf ("\n Thanks for using our ATM Service \n");
return 0;
}
```

output

```
    Welcome to ATM Service
Enter your secret PIN number: 9090
*** Main Menu ***
1. check Balance
2. withdraw cash
3. Deposit cash
4. Quit
*******
Enter your choice: 1
Your Balance in Rs: 1000
```

Do you want to continue? (y/n): Y
Enter your secret PIN number : 9090
*** Main Menu ***
1. Check Balance
2. Withdraw Cash
3. Deposit Cash
4. Quit
****
Enter your choice: 2
Enter the amount to withdraw: 500
Please collect cash
Your current balance is 500
Do you want to continue? (y/n): Y
Enter your secret PIN number : 9090
*** Main Menu ***
1. Check Balance
2. Withdraw Cash
3. Deposit Cash
4. Quit
****
Enter your choice : 3
Enter the amount to deposit : 400
Your balance is 900
Do you want to continue? (y/n): y
Enter the secret PIN number : 8080
Please enter a valid PIN
Enter your secret PIN number : 9090
*** Main Menu ***
1. Check Balance
2. Withdraw Cash
3. Deposit Cash
4. Quit
****
Enter your choice : 4
Thank you for using ATM

2. Menue - driven program to find the area of different shapes

```
#include <stdio.h>
int main ()
{
int ch, rad, length, width, breadth, height;
float area;
Printf ("Input 1 for the area of a circle \n");
Printf ("Input 2 for the area of a rectangle \n");
Printf ("Input 3 for the area of a triangle \n");
Printf ("Input your choice :");
Scanf ("%d", &ch);
Switch (ch)
{
case 1 :
  Printf ("Input the radius of the circle :");
   Scanf ("%d", &rad );
   area = 3.14 * rad * rad;
   break;
Case 2 :
    Printf ("Input the length and width of the rectangle :");
    scanf ("%d %d", &length, &width);
    area = length * width;
    break;
Case 3 :
     Printf ("Input the base and height of the triangle :");
     Scanf ("%d %d", &breadth, &height );
     area = 0.5 * breadth * height;
     break;
default :
      Printf ("Invalid choice \n");
      return 1;
}
Printf ("The area is : %f \n", area);
return ();
}
```

output

```
Input 3 for the area of a triangle
Input your choice : 2
Input the length and width of the rectangle : 4  8
The area is : 32.000000
```

3. Find the sum of even numbers

```
#include<stdio.h>
int main()
{
int i, x, sum=0;
Printf("Enter the upper limit : ");
Scanf("%d", &x);
for(i=0; i<=x; i+=2)
{
Sum = Sum+i;
}
Printf("Sum of all even numbers between 1 to %d = %d\n", x,
                                                    Sum);
return 0;
}
```

output

```
Enter the upper limit :6
Sum of all even numbers between 1 to 6 = 12
```

UNIT - II

Arrays

Introduction to Arrays - one dimensional arrays - Declaration - Initialization - Accessing elements - operations: Traversal, Insertion Deletion, Searching - Two dimensional arrays: Declaration - Initialization - Accessing elements - operations: Read - Print - Sum - Transpose - Exercise Programs. Print the number of positive and negative values present in the array - Sort the numbers using bubble sort - Find whether the given is matrix is diagonal or not.

UNIT - II

Arrays and Strings

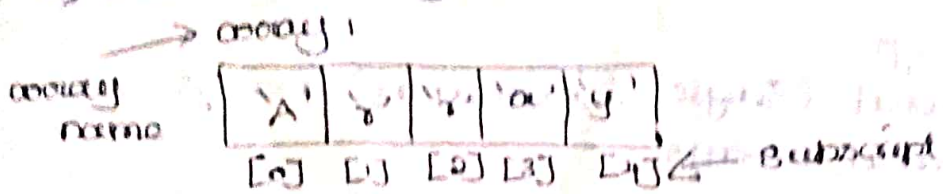
Introduction to Arrays

Definition

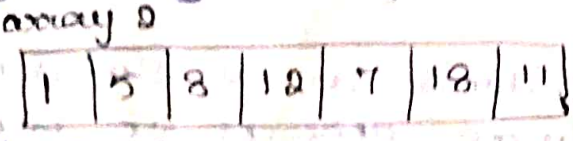
An array is a data structure that is used for the storage of homogeneous data ie) data of the same type

Types

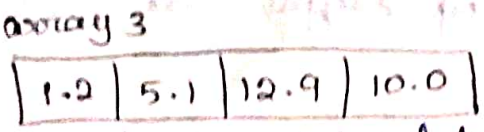
a) character array



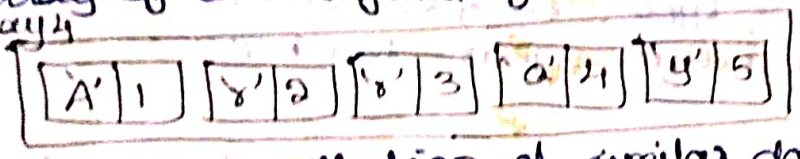
b) Integer array



c) float array



d) array of user-defined type



An array is a collection of similar data items, that are stored under a common name

- A value in an array is identified by index or subscript enclosed in square brackets with array name.
- Each array element is referred by specifying the array name with subscript (position of an element)

Ex x[5] // sixth element

Data type of an element is called element type. The array index starts with 0. ie) index of the first element of an array is 0.

Memory space required by an array can be computed as

(size of element type) × (Number of element in an array)

Ex char → size → 1

array 1 → 1×5 = 5 bytes

int → size → 2

array 2 → 2×8 = 16 bytes

Array are stored in contiguous memory locations

Eg: array 1:

char 2000, 2001, 2002, 2003, 2004

int array 2

2000-2001, 2002-2003, 2004-2005 and so on

Needs of an array:

It is used to define a set of similar data items. It is faster than dynamic memory allocation.

Classification of arrays

- One dimensional (Single dimensional) arrays
- Two-dimensional arrays
- Multi-dimensional arrays

One dimensional Arrays

The collection of data items can be stored under a one variable name using only one subscript. The elements of an array can be accessed by using single subscript. It is the type of linear array.

Declaration

It consists of a type specifier, an identifier and a size specifier enclosed within square brackets.

General form

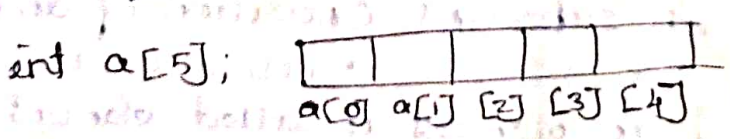
data_type array_variable [size specifier];

Ex:

int a[5]; ← type of data, name of an array

char name[20];

float avg[10]; ← size of the array



a is the array name which can hold 5 value of type integer.

Size specifier specifies the number of elements in an array

- Should be a compile-time constant expression of integral type
- The memory space is allocated at the compile time
- Should be greater than or equal to one

Initialization

The values can be initialized to an array. An initializer is an expression that determines the initial value of an element of the array

2 ways
- At compile time
- At run time

i) At compile time

Syntax

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

Ex: integer array;

int marks[3] = {70, 80, 90};

List of values must be separated by commas

70	marks[0]
80	marks[1]
90	marks[2]

(or)

marks[0] = 70;
marks[1] = 80;
marks[2] = 90;

character array

Ex: char name[] = {'L', 'A', 'K'}

ii) At run time

The array can be explicitly initialized at run-time

Ex:

```
while (i<=10)
{
  if (i<5)
    sum[i]=0;
  else
    sum[i]= sum[i]+i;
}
```

```
int a[9];
scanf("%d %d",
      &a[1], &a[2]);
```

The number of initializers should be less than or at most equal to the value of size specifier

If the number of initializers is less than the value of the size specifier the leading array locations equal to the number of initializer get initialized with the values of initializers The rest of the array locations get initialized to 0 (int) or 0.0 (float) or '\0' (char)

a) Basic initialization

int num[5]={3,7,12,24,25};

3	7	12	24	25

b) Initialization without size

int num[] = {3, 7, 12, 24, 45};

3	7	12	24	45

c) Partial Initialization

int num[5] = {3, 7};

3	7	0	0	0

rest are filled with 0's

d) Initialization to all zeros

int num[5] = {0};

0	0	0	0	0

All are filled with 0's

operation on a single dimensional Array

① Subscripting a single dimensional array

The only operation allowed to an array is subscripting Subscripting is an operation that selects an element from an array

Ex: a[3] → denotes the 3rd element

② Assigning an array to another array

A normal variable can be assigned to or initialized with

another variable to an array which is assigned to or initialized with another way array.

Ex: int a[10], b[10];
a = b; // error

Example Program
To find sum Reading, storing and accessing elements of one dim-array

```
#include <stdio.h>
#include <conio.h>
main()
{
  int a[10], i, n;
  Printf("\n Enter the number of elements: ");
  Scanf("%d", &n);
  Printf("\n Enter the elements of an array: ");
  for(i=0; i<n; i++)
  Scanf("%d", &a[i]);
  Printf("\n Array Elements are: ");
  for(i=0; i<n, i++)
  Printf("%d", a[i]);
  getch();
}
```

Two dimensional Arrays

A two dimensional array has its elements arranged in a rectangular grid of rows and columns.

- The elements of a two dimensional array can be accessed by using a row subscript and a column subscript
- Both the row subscript and the column subscript are required to select an element
- Known as a matrix

2D Array

Columns ->

Rows ↓							
0	1	2	3	4	5	6
1	6	8	4	5	9	0
2	7	2	4	8	0	4
6	3	1	1	8	3	0

Two dimensional arrays are used in situation where a table of values need to be stored in an array.

Declaration

Syntax

data_type array_name[row_size][column_size];

- data_type: type of the data
- array_name: name of the array
- row_size: size of the row
- column_size: size of the column

Ex:
int a[3][3];

	col 0	col 1	col 2
row 0	a[0,0]	a[0,1]	a[0,2]
row 1	a[1,0]	a[1,1]	a[1,2]
row 2	a[2,0]	a[2,1]	a[2,2]

	col 0	col 1	col 2
row 0	75	21	64
row 1	90	72	51
row 2	34	49	80

Initialization

The values can be initialized to the two dimensional arrays at the time of declaration

Syntax

data_type array_name[rowsize][columnsize] = {list of values};

Ex

```
int a[2][2] = {{10,5,6},{9,7,1,9}};
int stud[2][2] = {{6680,80},{6681,81},{6680,80},{6683,83}};
        (or)
int stud[2][2] = {6680,80,6681,81,6682,82,6683,83};
        (or)
int stud[][2] = {6680,80,6681,81,6682,80,6683,83};
```

column size should be mentioned row size is optional. Then only the compiler knows where the last row ends.

① `int a[4][4] = {{2,1},{2,3,4},{5},{6,1,6,8}};`

column →

row ↓							
0	2	1	0	0	0	0	0
1	2	3	4	0	0	0	0
2	5	0	0	0	0	0	0
3	6	1	6	8	0	0	0

② `int a[3][4] = {{2,1},{2,3,4}};`

column →

row ↓							
0	2	1	0	0	0	0	0
1	2	3	4	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0

Multi Dimensional Array

An array with 3 or more dimensions

Declaration

Syntax

data_type array_name[size1][size2][size3] ... [sizen];

Ex `int a[3][3][3];`

String operations

String – is a sequence of characters enclosed within double quotes " "

character is enclosed within single quote ' '

Ex: String "A"

character 'A'

Every string is automatically terminated by a null character

C string library provides the following functions as predefined functions

- reading
- copying
- comparing
- combining
- searching etc

Memory space

Strings is a sequence of characters enclosed within double quotes " ". character is enclosed within single quote ' '. Ex: String "A"
character 'A'

Every string is automatically terminated by a null character. C string library provides the following functions as predefined functions

Memory space

Strings are stored in contiguous memory locations with terminating null character. The amount of memory space required depends upon the number of characters present in the string. The number of bytes required is one more than the number of characters present in it.

Ex: "ABC" -> requires 4 bytes
3 bytes - string
1 bytes - null character.

Length of the string - number of characters present

Empty string

- A string with length zero.
- written as " "
- no character is enclosed within double quotes.

Data type

In C language, string data type is not available. character array is used to represent strings.

character Array

character Array is used to store a string. It stores elements of type "char"

Syntax: char identifier[size] = initialization_list;

no of characters (length+1)

Size is optional, if initialization list is present

a. String Declaration

Syntax: datatype str_name[size];

Ex: char var[5];

Operations

operations on array includes

- Traversal
- Selection
- Insertion
- Deletion
- Searching

Traversal

Traversal is an operation in which each element of a list, stored in an array, is visited. The travel proceeds from the zero element to the last element of the list.

Program

```
#include <stdio.h>
void main()
{
int list[10];
int n;
int i, neg=0, zero=0, pos=0;
Printf("\n enter the size of the list \n");
Scanf("%d", &n);
Printf("enter the elements one by one");
for(i=0;i<n;i++)
{
 Printf("\n enter number %d number", i);
 Scanf("%d", &list[i]);
}
for(i=0;i<n;i++)
{
 if(list[i]<0)
    neg=neg+1;
 else
    if(list[i]==0)
       zero=zero+1;
 else
    pos=pos+1;
 }
Printf("No of negative numbers in given list are %d", neg);
Printf("No of zeros in given list are %d", zero);
Printf("No of positive numbers in given list are %d", pos);
}
```

Selection

An array allows selection of an element for given index array is called as random access data structure.

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
float marks[10];
```

```
int size, i, pos, choice;
float percentage;
Printf ("\n Enter the size of the list");
Scanf ("%d", &size);
Printf ("\n Enter the marks list one by one");
for (i=0; i<size; i++)
{
  Printf ("\n enter data ");
  Scanf ("%f", &marks[i]);
}
do
{
Printf ("\n menu");
Printf ("\n Query ... 1");
Printf ("\n Quit ... 2");
Printf ("\n enter your choice ");
Scanf ("%d", &choice);
Switch (choice)
{
Case 1:
Printf ("\n enter position ");
Scanf ("%d", &pos);
percentage = marks [pos];
Printf ("\n percentage = %4.2f", percentage);
break;
Case 2:
Printf ("\n Quitting");
Printf ("\n press a key to continue ...");
}
while (choice != 2);
}
```

Insertion

It is the operation that inserts an element at a given location of the list. To insert an element at i^{th} location of the list then all elements from the right of $i+1^{th}$ location have to be shifted one step towards right

Program

```
#include <stdio.h>
int main()
{
int array[100], position, i, n, value;
Printf ("enter number of elements in array \n");
Scanf ("%d", &n);
Printf ("enter %d elements \n", n);
for (i=0; i<n; i++)
```

```
scanf ("%d", &array[i]);
printf ("enter the location where you wish to insert an element\n");
scanf ("%d", &position);
printf ("enter the value to insert\n");
scanf ("%d", &value);
for (i=n-1; i>=position-1; i--)
array[i+1] = array[i];
array[position-1] = value;
printf ("Resultant array is\n");
for (i=0; i<=n; i++)
printf ("%d\n", array[i]);
return 0;
}
```

Deletion

It is the operation that removes an element from a given location of the list. To delete an element from the i^{th} location of the list then all elements from the right of $i+1^{th}$ location have to be shifted one step towards left to preserve contiguous locations in the array.

Program

```
#include<stdio.h>
int main()
{
int array[100], position, i, n;
printf ("enter number of elements in array\n");
scanf ("%d", &n);
printf ("enter %d elements\n", n);
for (i=0; i<n; i++)
scanf ("%d", &array[i]);
printf ("enter the location where you wish to delete\n");
scanf ("%d", &position);
if (position >= n+1)
printf ("Deletion not possible\n");
else
{
for (i=position-1; i<n-1; i++)
array[i] = array[i+1];
printf ("Resultant array is\n");
for (i=0; i<n; i++)
printf ("%d\n", array[i]);
}
return 0;
}
```

Searching

It is an operation in which a given list is searched for a particular value. A list can be searched sequentially where in the search for the data item starts from the beginning and continues till the end of the list.

Eg Linear search
binary search

operations on 2D array

Read
Print
Sum
Transpose

Program - Transpose

Transpose of a matrix in c language This c program prints transpose of a matrix

```
#include <stdio.h>
void main()
{
int arr1[50][50], brr1[50][50], i, j, r, c;
Printf("\n\n Transpose of a matrix:\n");
Printf("____\n");
Printf("\n Input the rows and columns ");
Scanf("%d %d", &r, &c);
Printf("Input elements in the first matrix\n");
for(i=0; i<r; i++)
{ for(j=0; j<c; j++)
  {
    Printf("element - [%d],[%d]: ", i, j);
    Scanf("%d", &arr1[i][j]);
}}
Printf("\n The matrix is :\n");
for(i=0; i<r; i++)
{
Printf("\n");
for(j=0; j<c; j++)
Printf("%d\t", arr1[i][j]);
}
for(i=0; i<r; i++)
{
for(j=0; j<c; j++)
{
brr1[j][i]= arr1[i][j];
}}
Printf("\n\n The transpose of a matrix is ");
for(i=0; i<c; i++)
```

```
Printf ("\n");
for (j=0; j<r; j++)
{
Printf ("%d", arr[i][j]);
}
}
```

Sorting

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical or any user defined order.

Types of sorting

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort

Exercise Programs

1. Print the number of positive and negative values present in the array

```
#include <stdio.h>
void main()
{
int list[10];
int n;
int i, neg=0, zero=0, pos=0;
Printf("\n Enter the size of the list: ");
Scanf("%d", &n);
Printf("Enter the elements one by one: \n");
for(i=0; i<n; i++)
{
Printf("Enter number %d: ", i);
Scanf("%d", &list[i]);
}
for(i=0; i<n; i++)
{
  if (list[i]<0)
     neg = neg+1;
  else if (list[i]==0)
     zero = zero+1;
  else
     pos=pos+1;
}
Printf("Number of negative numbers in the given list: %d\n", neg);
Printf("Number of zeros in the given list %d \n", zero);
Printf("Number of positive numbers in the given list: %d\n", pos);
}
```

output

```
Enter the size of the list: 5
Enter the elements one by one
Enter number 0: 3
Enter number 1: 0
Enter number 2: 1
Enter number 3: -4
Enter number 4: 0
Number of negative numbers in the given list: 1
```

Number of zeros in the given list : 2
Number of positive numbers in the given list 2

2. Sort the numbers using bubble sort

Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order until the entire list is sorted.

```
#include<stdio.h>
int main()
{
  int count, num[50], i;
  Printf ("How many elements to be sorted:");
  Scanf("%d", &count);
  Printf ("Enter the elements:\n");
  for(i=0; i< count; i++)
  {
   Printf("num[%d]:", i);
   Scanf("%d", &num[i]);
  }
  Printf("\n Array Before Sorting:\n");
  for (i=0; i< count; i++)
    Printf("%5d ", num[i]);
  int pass, current, temp;
  for (pass=1, pass <count; pass++)
  {
    for(current=1; current <=count-pass; current++)
    {
      if(num[current-1] >num[current])
      {
        temp=num[current-1];
        num [current-1] =num[current];
        num [current] =temp;
      }
    }
  }
  Printf("\n Array After Sorting :\n");
  for(i=0; i<count; i++)
    Printf("%5d", num[i]);
  return 0;
}
```

output

```
How many elements to be sorted 5
Enter the elements
num[0]:72
num[1]:31
num[2]:49
num[3]:97
num[4]:22
Array Before Sorting
  72  31  49  97  22
Array After Sorting
  22  31  49  72  97
```

3. Find whether the given matrix is diagonal or not

A matrix having non-zero elements only in the diagonal running from the upper left to the lower right is called diagonal matrix.

```
#include<stdio.h>
void main()
{
 int x[10][10],nr,nc,r,c,flag;
 Printf("Enter the number of rows and columns:");
 Scanf("%d %d",&nr,&nc);
 if(nr==nc)
 {
 Printf("Enter elements of the matrix:\n");
  for(r=0;r<nr;r++)
     for(c=0;c<nc;c++)
        scanf("%d",&x[r][c]);
  flag=1;
  for(r=0;r<nr;r++)
  {
   for(c=0;c<nc;c++)
   {
     if(r==c)
     {
       if(x[r][c]==0)
          flag=0;
     }
     else
     {
       if(x[r][c]!=0)
```

```
            flag=0;
        }
    }
}
if (flag==1)
    Printf("The matrix is a diagonal matrix");
else
    Printf("The matrix is not a diagonal matrix");
}
else
{
    Printf("The matrix is not a square matrix");
}
}
```

output

```
Enter the number of rows and columns 2 2
Enter elements of the matrix
2 0
0 1
The matrix is a diagonal matrix.
```

UNIT - III

Strings

Introduction to strings - Reading and writing a string - string operations (without using built-in string function) length - compare - concatenate - copy - Reverse - Substring - Insertion - Indexing - Deletion - Replacement - Array of strings - Introduction to pointers - pointer operators . pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names

The variable can hold 4 characters 5th space is for '\0'

b. String Initialization

Assigning the values strings can be initialized in 3 ways

* By using string literal constant char str[6] = "Hello";
* By using list of characters initializers char str[6] = {'H','a','i','\0'};

'\0' → end of a string Eg. char str[6] = {'H','a','i','\0'};

* Size is not mentioned char str[] = "name!";

Size will be automatically allocated based on the number of characters

c. Reading a string

Reading a single character or series of characters (strings) from the input device there are 3 ways.

- scanf()
- getchar()
- gets()

i) scanf()
- formatted input function
- format string %s
- Automatically terminates the string that is read with a null character

Ex: char name[6];
scanf("%s", name);

ii) getchar()

It is used to read any alphanumeric character from the input devices. It reads one character at a time & unformatted function.

Syntax:
```
char var_name;
var_name = getchar();
```

iii) gets()

String input function (unformatted) -
Reads a group of characters until an enter key is pressed

Syntax:
```
char var_name;
gets(var_name);
```

d) Printing a string

Printing - displaying or writing a single character or a string to the output devices.

three ways: - Printf()
- Putchar()
- puts()

i) Printf()
- formatted output function to write a word or a single character to the o/p devices.
- format string is %s

Syntax
```
char var[size];
Printf("%s", var);
```
Ex:
```
char name[6];
Scanf("%s",&name);
Printf("%s", name);
```

ii) Putchar():
- Unformatted o/p function to write any alphanumeric character to the output device.
- Prints one character at a time

Syntax
```
char var;
Putchar(var);
```

iii) puts()
- Unformatted output function to write any alphanumeric character to the o/p device.
- stdio.h header file is needed.

Syntax
```
char var[n];
puts(var);
```

String operations
- Length
- compare
- concatenate
- copy

String Library Functions / String Manipulation Functions / string
Standard Functions

Strlen(s) - Strlen returns the length of a string
strcpy() - strcpy copies one string to another
Strcat() - strcat combines two strings
Strcmp() - compares 2 strings and returns an integer indicating the difference b/w the strings.
Strrev() - used to reverse a string
strlwr() - used to convert string to lowercase
strupr() - used to convert string to uppercase
Strset(- sets all characters in a string to a specific character

strchr() - determines the first occurrence of a string in another str
strstr() - finds the first occurrence of a string in another string.
strdup() - used to duplicate a string.

(1) Length

Strlen() is used to find the length of a string. The function returns the length of the string as output. The terminating null character will not be counted.

Syntax

```
var = strlen (string);
```

Ex

```
s1 = "Hello";                    o/p
length = strlen (s1);             5
Printf ("%d", length);
```

(2) compare

functions of comparing 2 strings are,

- strcmp()
- strcmpi()
- strncmp()
- strncmpi()

i. strcmp()

This function performs the comparison of 2 strings character by character until the corresponding characters differ or the end of the string is reached. It returns the ASCII difference of the first mismatch characters or zero if both are same.

Syntax

```
strcmp (string1, string2);
```

Ex:

```
char s1[5] = "Hello";            o/p
char s2[5] = "Good";              -1
n = strcmp (s1, s2);
Printf ("%d", n);
```

ii) strcmpi()

It compare 2 strings without case sensitivity

Syntax:

```
strcmpi (str1, str2)
```

Ex:

```
char s1[5] = "Good";             o/p
char s2[5] = "GooD";              0
n = strcmpi (s1, s2);
Printf ("%d", n);
```

Strncmp()

It is used to compare a portion of 2 strings

n -> no of characters to be compared.

Syntax
strncmp(string 1, string 2, n);
Ex
char s1[20]: "Welcome";
char s2[20]: "Welcome You All";
n = strncmp(s1, s2, 3);
Printf("%d", n);
o/p
0

iv) strncmpi()
It is used to compare a portion of 2 strings without case sensitivity
i → ignore case
n → no of characters
Syntax
strncmpi(str1, str2, n);

③ concatenate:
It is used to concatenate one string with another string append a source string to the destination string
Functions
strcat()
strncat()

i) strcat()
concatenate one string with another string.
Syntax:
strcat(String1, String 2);
source destination
The function appends a source string to the destination string It returns a pointer to the destination string as output.
Ex:
char s1[20] = "Good";
char s2[20] = "Morning";
strcat(s1, s2);
puts(s1);
s1, s2
o/p
Good Morning

ii) strncat
It concatenates a portion of one string with another string. It append almost n character of a source string to the destination string. n = number of characters of the source string to be copied
Syntax
strncat(d, s, n);
Ex:
char s1[20] = "Good";
char s2[20] = "To all";
strncat(s1, s2, 2);
puts(s1);
o/p
Good To

④ copy

It is used to copy the source string to the destination string

Functions

- strcpy()
- strncpy()

i) strcpy()

It copies the source string to the destination string. It returns a pointer to the destination string as output.

Syntax

```
strcpy(dest, source);
```

Ex:

```
char s1[20] = "welcome";
char s2[20];
strcpy(s2, s1);
puts(s2);
```

O/P

welcome

ii) strncpy()

It is used to copy atmost n characters of a source string to the destination string

Syntax

```
strncpy(str1, str2, n);
```

Ex:

```
char s[20] = "welcome";
char d[20];
strncpy(d, s, 3);
d[3] = '\0';
puts(d);
```

O/P

wel

other functions

⑤ strrev()

It is used to reverse all the characters of a string except the terminating null character. It reverses the string and returns a pointer to the reversed string as output.

Syntax

```
strrev(string);
```

Ex:

```
char s[20] = "Hello";
strrev(s);
puts(s);
```

O/P

olleH

⑥ strlwr()

It converts all the character in a string to lowercase. It returns a pointer to the converted string as O/P

Syntax

```
strlwr(str);
```

Ex

```
char s[20] : "WELCOME";
strlwr(s);
puts(s);
```

o/p

welcome

⑦ strupr()

It is used to converts all the characters in a string to uppercase. It returns a pointer to the converted string as o/p.

Syntax

```
strupr(str);
```

Ex

```
char s[20] = "Welcome";
strupr(s);
puts(s);
```

o/p

WELCOME

⑧ a) strset()

It sets all characters in a string to a specific character. input is string and a character.

Syntax

```
strset(str, ch)
```

Ex

```
char s[5] = "Good";
strset(s, 'H');
puts(s);
```

o/p

HHHH

b) strnset()

It sets first n characters in a string to a specific charac input is a string, a character and an integer value n.

Syntax

```
strnset(string, ch, n);
```

Ex

```
char s[10] = "welcome";
strset(s, 'H', 3);
puts(s)
```

o/p

HHH come

⑨ a) strchr()

It searches a string for the first occurrence of a gn char of a character is found, it returns a pointer to the first occurrence of the character in the given string. If a character is not found, it returns NULL.

Syntax

```
strchr(string, ch);
```

b) strrchr()

It searches a string for the last occurrence of a given character.

Syntax:
```
strrchr (string, ch);
```
Ex:
```
char s[20] = "welcome";
char *ptr;
ptr = strchr (s,'e');
printf ("%d ", ptr); //first
ptr = strrchr (s,'e');
printf ("%d ", ptr); //last
```
o/p
1
6

(10) strstr ()

It finds the first occurrence of a string in another string. If a string s2 is found, it returns the position from where the string starts. If a string s2 is not found in string s1, it returns NULL.

Syntax:
```
strstr (s1,s2);
```
Ex:
```
char s1[20] = "Welcome";
char s2[20] = "come";
char *ptr;
ptr = strstr (s1,s2);
printf ("found at %d ", ptr);
```
o/p
found at 3

String Arrays

A list of strings can be stored in 2 ways:

a) using an array of strings

b) using an array of character pointers.

a) Array of strings

List of strings can be stored by two dimensional character array.

Declaration
```
char id[row][coln] = list;
```
Ex
```
char array[2][30];
```
Initialization

- using string literal constant
```
char str[][20] = {"CIVIL", "CSE", "ECE"};
```
- using list of character initializers
```
char str[][20] = { {'c','i','v','i','l','\0'},
                   {'c','s','E','\0'},
                   {'E','c','E','\0'} };
```

b) Array of character pointers

Array of strings can be stored by using an array of character pointers.

Ex:

```
char *lang[20] = {"CIVIL", "CSE", "ECE"};
```

Selection Sort

A sorting algorithm that selects the smallest elements from an unsorted list in each iteration and places that element at the begening of the unsorted list.

Steps:

- Set the first elt as min
- compare the remaining elts and select the smallest elt
- If that elt is smaller than min then swap the 2 elts.
- Again reassign the min elt from the second elt
  compare the remaining elts
  swap if min > elt.
- The process is continued until the last elt.
- Finally, the array will be sorted.

Ex

arr

0	1	2	3	
20	12	10	15	2

Step 1:

min ← arr[0] = 20

compare the remaining elts and select the smallest elt.

12 > (10)

(10) < 15

10 > (2)

select 2. swaps with min

arr

2	12	10	15	20
0	1	2	3	4

Step 2

Now, min ← arr[1] = 12

compare the remaining elts & select the smallest elt.

(10) < 15

(10) < 20

Select 10. compare with min ie) 12, 12 > 10

So swap

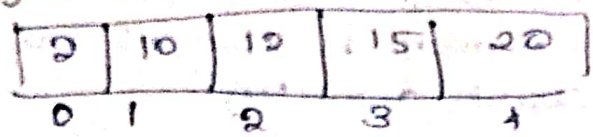
2	10	12	15	20

Step 3

min ← arr[2] = 12

compare. No smallest elt.

Step 4:
min ← arr[3] = 15, No smallest elt.
Step 5:
min ← arr[4] = 20, No remaining elts.
Sorted Array:



Program

```
#include <stdio.h>
void main()
{
int a[100],n,i,j,pos,swap;
Printf ("Selection Sort \n");
Printf ("Enter the number of elements: ");
Scanf("%d",&n);
Printf ("Enter the elements: ");
for(i=0; i<n; i++)
Scanf ("%d", &a[i]);
for (i=0; i<n-1; i++)
{
pos = i;
for (j = i+1; j <n; j++)
{
if (a[pos]>a[i])
pos = j;
}
if (pos!=i)
{
swap = a[i];
a[i] = a[pos];
a[pos]= swap;
}}
Printf (" Sorted Array \n");
for (i=0; i<n; i++)
Printf ("%d\t ", a[i]);
}
```

o/p

```
Selection Sort
Enter the number of elements : 5
Enter the elements : 20 12 10 15 2
Sorted Array
2 10 12 15 20
```

Searching

Search is an operation in which a given list is searched for a particular value. The location of the searched element is informed. activity of looking for a value or item in a list

Types of Search
- Linear Search
- Binary Search

1) Linear Search (or) Sequential Search

The search starts from the first elements and continues in a sequential fashion from element to element till the desired entry is found. simplest search algorithm.

Operation Steps:

- Traverse the array in sequence from the first element to the last.
- Each element is compared to the key
  * If the key is found in the array, the corresponding array index is returned.
  * If the item is not found in the array an invalid index -1 is returned
- In the worst case, the number of comparisions is proportional to the size of the array

a[4] -> located at

a	24	21	36	14	62	91	8	22	7	81	77
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Key = 62

Program

```
#include<stdio.h>
int main()
{
int a[10], i, n, m, c=0;
Printf("Enter the size of an array");
Scanf("%d", &n);
Printf("Enter the elements of the array");
for(i=0; i<=n-1; i++)
{
Scanf("%d", &a[i]);
}
Printf("Enter the number to be search");
Scanf("%d", &m);
for(i=0; i<=n-1; i++)
{
if (a[i]==m)
{
  c=1;
  break;
}
}
```

```
if(c==0)
Printf("The number is not in the list");
else
printf("The number is found");
getch();
return 0;
}
```

o/p

Enter the size of an array 5
Enter the elements of the array : 4 6 8 0 3
Enter the number to be search : 0
The number is found

Example:

1	2	3	4	5
4	6	8	0	3

Key is 0

m=0 ① → i) (a[0] = 4) == (m=0) ii) 4 ≠ 0

② → i) (a[1] = 1) == 0 ii) 1 ≠ 0

③ → i) (a[2] = 8) == 0 ii) 8 ≠ 0

④ → i) (a[3] = 0) == 0 ii) 0 = 0

so set c = 1

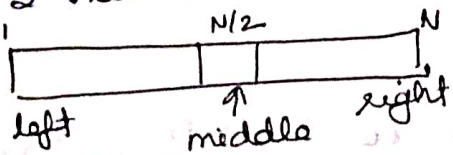
Then the number is found

Disadvantages:

- slow
- ineffficient

ii) Binary search

Binary search is a divide and conquer search algorithm to find out the position of a specified value within an array. The array must be sorted in either ascending or descending order. The binary search requires arrays to be sorted. The list is divided into 2 halves separated by the middle element.



operation

Steps:

- The middle element is tested for the required entry. If found then its position is reported, else the following test is made.
- If Key < middle, search the left half of the list, else search the right half of the list.
- Repeat step 1 and step 2 on the selected half until the entry is found, otherwise report failure.

In each iteration, the given list is divided into 2 parts. The search becomes limited to half the size of the list.

	0	1	2	3	4	5	6	7	8
a	8	4	5	7	11	13	14	12	17

middle value
averageing the first and last indices and truncating the result

ex:
$$\frac{0+8}{2} = \frac{8}{2} = 4$$

ie) the content of the fourth location

Ex: Key = 14

Step 1:
$$\frac{0+8}{2} = 4 \qquad mid = a[4] = 11 \neq 14$$

$\rightarrow a[4] \neq 14$

$\rightarrow 14 > 11$

The search element is present in the right half.

Step 2
$$\frac{5+8}{2} = \frac{13}{2} = 6$$

$a[6] = 14$

$Key = 14 = a[6]$

The element is found

Program

```
#include<stdio.h>
main()
{
int a[10], i, n, m, c=0, l, u, mid;
Printf("Enter the size of an array");
Scanf("%d", &n);
Printf("Enter the elements in ascending order");
for(i=0; i<n; i++)
{
Scanf("%d", &a[i]);
}
Printf("Enter the number to be search elements in ascending order");
for(i=0;i<n;i++)
{ Scanf("%d", &m);
scanf l=0, u=n-1;
while (l<=u)
{
mid = (l+u)/2;
if (m==a[mid])
{
c=1;
break;
}
```

```
else if (m<a[mid])
{
u = mid - 1 ;
}
else
  l = mid + 1 ;
if (c==0)
Printf ("The number is not found");
else
Printf ("The number is found");
getch();
return 0;
}
```

O/P

```
Enter the size of an array: 5
Enter the elements in ascending order: 4 7 8 11 21
Enter the number to be search 11
The number is found
```

characteristics

- List must be sorted
- faster than linear search

Exercise Programs

1. To find the frequency of a character in a string

```
#include<stdio.h>
#define MAX 100
int main()
{
 char str[MAX]={0};
 char ch
 int count, i ;
 Printf(" Enter a String: ");
 Scanf ("%[^\n]s", str);   // Read string with spaces
 getchar();                // Get extra character (enter/return key)
 // Input character to check frequency.
 Printf(" Enter a character");
 ch = getchar();
 // calculate frequency of character
 count=0;
 for(i=0; str[i]!='\0'; i++)
 {
  if(str[i]==ch)
     count++;
 }
 Printf("%c found %d times in \"%s\" \n", ch. count, str);
 return 0;
}
```

output

```
Enter a String: Hello
Enter a character: l
l found 2 times in "Hello"
```

2. To find the number of vowels, consonants and white spaces in a given text

```
#include<stdio.h>
#include<ctype.h> // for the isalpha() function
int main()
{
 char line[50];
 int vowels, consonants, digit, space;
 vowels = consonant = digit = space=0;
 Printf("Enter a line of text: ");
 fgets (line, sizeof(line), stdin);
```

```
for(int i=0; line[i] != '\0'; ++i)
{
if(tolower(line[i]) == 'a' || tolower(line[i]) == 'e' || tolower(
   line[i]) == 'i' || tolower(line[i]) == 'o' || tolower(line[i])
   == 'u')
  {
    ++vowels;
  }
else if(isalpha(line[i]))
{                    // use isalpha() to check if the character is a letter
  ++consonant;
}
else if(isdigit(line[i]))
{
  ++digit;
}
else if(line[i] == ' ')
{                    // only count space
  ++space;
}
}
Printf("Vowels: %d\n", vowels);
Printf("consonants: %d\n", consonants);
Printf("Digits : %d\n", digit);
Printf("white spaces : %d \n", space);
return 0;
}
```

output

```
Enter a line of text : Have a nice day
Vowels : 6
consonants : 6
Digits : 0
white space : 3
```

3. Sorting the names

```
#include<stdio.h>
#include<string.h>
void main()
{
char name[10][20], tname[10][20], temp[20];
int i,j,n;
Printf("Enter the value of n:");
Scanf("%d", &n);
```

```
Printf("Enter %d names : \n", n);
for(i=0; i<n; i++)
{
  scanf("%s", name[i]);
  strcpy(tname[i], name[i]);
}
for(i=0; i<n-1; i++)
{
  for(j=i+1; j<n; j++)
  {
    if(strcmp(name[i], name[j])>0)
    {
      strcpy(temp, name[i]);
      strcpy(name[i], name[j]);
      strcpy(name[j], temp);
    }
  }
}
Printf("\n.....\n");
Printf("Input Names\t\t Sorted Names\n");
Printf("..... \n");
for(i=0; i<n; i++)
{
  Printf("%20s\t\t%s\n", tname[i], name[i]);
}
Printf("..... \n");
}
```

output

```
Enter the value of n : 4
Enter 4 names
Rajan
Hosea
Priya
Suthan
```

Input Names	Sorted Names
Rajan	Hosea
Hosea	Priya
Priya	Rajan
Suthan	Suthan

UNIT- IV

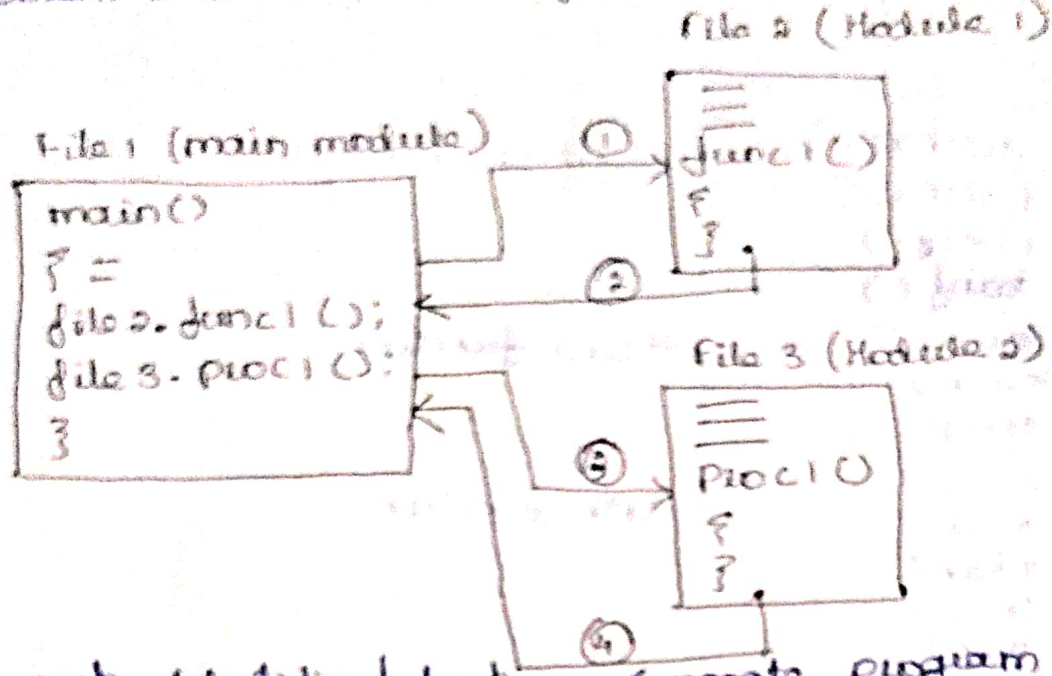
Functions

Introduction to function - Types: User-defined and built in functions - function Prototype - Function definition - Function call - parameter passing: pass by value - pass by reference - Built in functions (string functions) - Recursive functions - Exercise programs: calculate the total amount of power consumed by 'n' devices (passing an array to a function) - Menu-driven program to count the numbers which are divisible by 3, 5 and by both passing an array to a function) - Replace the punctuations from a given sentence by the space character (passing an array to a function)

Functions and Pointers

Modular Programming

Breaking down of a program into a group of files whose each file consists of a program that can be executed independently. The problem is divided into different independent but related tasks.



For each identified task, a separate program (module) is written which is a program file that can be executed independently. The different files of the program are integrated using a main program file. The main program file invokes the other files in an order that fulfills the functionality of the program.

Function

A function is a self contained program or a subprogram of one or more statements which is used to do some particular task.

Classification

i) Pre defined functions (Library functions)
ii) User defined functions

Pre-defined functions / Library functions / Built-in functions

Functions whose functionality has already been developed by some one and are available to the user for use.
ex: Printf, Scanf, Sqrt(x,y), Strcpy(), Strcmp() etc

Two aspects
i) Declaration of library functions
ii) Use of library functions.

i) Declaration of library functions

A library function needs to be declared before it is called

It is available in header files. Header files are included to access the functions.

ex: Printf is available in stdio.h. So stdio.h is included before calling the printf function

Syntax

```
#include <filename.h>
```

Library Function

Header file	Function	
stdio.h	getchar() putchar() printf() scanf()	Built-in function
string.h	strcat() strcmp() strcpy()	String function
math.h	acos() asin() atan() cos() exp() sqrt()	Math function
stdlib.h	malloc() rand()	
ctype.h	isdigit() islower() isupper()	

i) Use of library functions using a function call operator ()

ex: strcpy();

Example

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{ int a,b;
printf("Enter the number");
scanf("%d", &a);
b=sqrt(a);
printf("The Square is %d", b);
getch();
}
```

1) User defined functions / programmer defined functions

Functions that are defined by the user at the time of writing a program. The user develops the functionality by writing the body of the function. Functions are used to break down a large program into a number of smaller functions

Merits

- Easy to locate and debug an error
- length of the program can be reduced
- avoid coding of repeated programming

Three Aspects:

- Function declaration / function prototype
- Function definition
- Function use (function call / function invocation)

Function Declaration

All the functions need to be declared as defined before they are used

General form

return type function_name (parameter_list or parameter_type list);

Ex:

```
int add (int , int);
int sub (int x , int y);
```

Rules

- The parameter list must be separated by commas
- The parameter names do not need to be the same in the prototype declaration & function definition
- The types must match the types of parameters in the function definition in number & order

Function Definition

composing a function. It is the process of specifying and establishing the user defined function by specifying all of its elements and characteristics

Two parts

- Header of the function
- body of the function

Header of the function

General form

return_type function_name (parameter_list)

Body of the function

It consists of a set of statements enclosed within braces

Syntax

```
return_type function_name (parameter_list)
parameter declaration
{
local variable declaration;
:
body of the function;
:
return (expression);
}
```

Example

```
int add (int x, int y)
{
int z;
z = x+y;
return (z);
}
```

Function use / Function call

The function can be called by simple specifying the name of the function, return and parameters if present.

Syntax

- function_name ( );
- function_name (parameters);
- return_value = function_name (parameters);

Ex:

```
add ();
add (a, b);
c=add (a, b);
```

Program

```
#include <stdio.h>
#include <conio.h>
Print_message();        // function declaration
main()
{ Print_message();      // function call
}
Print_message()         // function definition
{
Printf ("Hello");
getch();
return;
}
```

Accessing a function

A function can be accessed by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas. There may be several different calls to the same function from various places within a program.

Parameters or Arguments

Parameters provide the data communication between the calling function and called function

Two types
- Actual parameters
- Formal parameters

Actual parameters – parameters transferred from calling function to called function.

Formal parameters – Parameters used in the called function for the values that are passed from the called function.

Example:

```
main()
{
...
add(x,y);   -> calling function
...
}
add(a,b)    -> called function
{
...
return();
}
```

Function call
x, y - actual parameters

Function definition
a, b - formal parameters

return Statement

It is used to return the result of the computations performed in the called function and/or to transfer the program control back to the calling function

Two forms

```
return;            // just transfer the control to calling function
return (expression); // transfers the control, returns a value to
                        the calling function.
```

Ex:

```
- if (x <= i)
      return (1);
- return (x);
- return (a+b*c);
```

Function Prototypes

Functions are classified into 4 types based on return values and arguments.

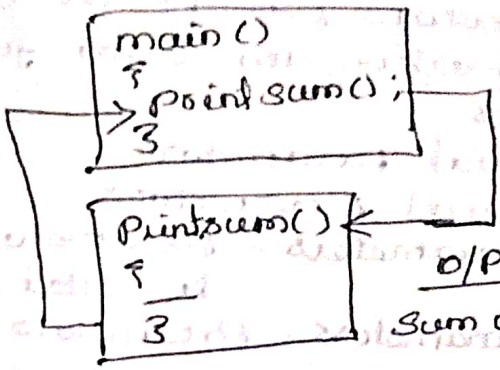
- Function without argument and no return value
- Function with arguments and no return values
- Function with arguments and with return values
- Function without arguments and with return value.

Function without argument and no return values

It doesn't accept any input and doesn't return any result. Parameter list is empty.

Ex:

```
#include<stdio.h>
void printsum(); // fn decln
main()
{
printsum();          // fn call
}
printsum()           // fn defn
{
printf("sum of 2 & 3 is %d", 2+3);
}
```



O/P

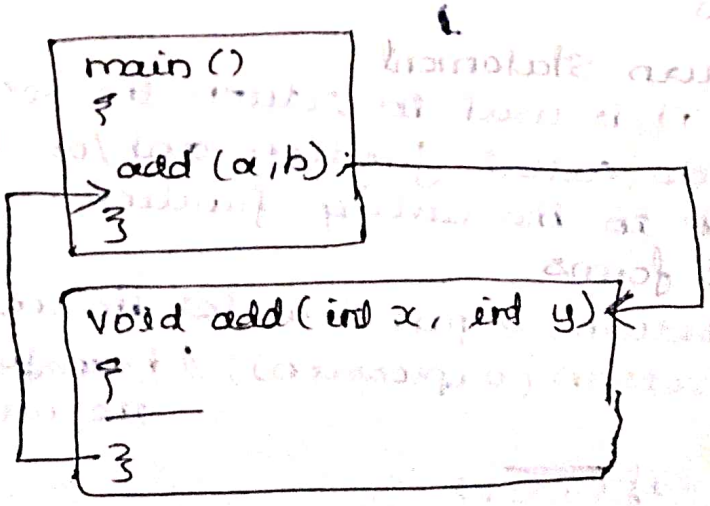
Sum of 2 & 3 is 5

Function with argument and no return values

A function has arguments. It receives data from the calling function. The calling function reads the data from input terminal and pass it to the called function. The program control is transferred to called function. The execution of calling function is suspended and the called function starts the execution. When the execution of the called function is completed the program control returns to the calling function and the calling function resumes its execution.

Ex:

```
#include<stdio.h>
main()
{
void add(int, int);
int a,b;
printf("Enter a & b");
scanf("%d %d", &a, &b);
add(a,b);
}
void add(int x, int y)
{
int z;
z=x+y;
printf("sum is %d", z);
getch();
}
```



O/P

Enter a & b

2 3

Sum is 5

Function with argument and return value

Data is transferred from calling function to called function to the called function receives data from calling function and send back a value return to calling function

```
Ex
#include<stdio.h>
main()
{
int add(int, int);
int a,b,c;
Printf("Enter 2 values");
Scanf("%d %d",&a,&b);
c=add(a,b);
Printf("Sum is %d",c);
}
int add(int x, int y)
{
int z;
z=x+y;
Return(z);
```

```
main()
{
---
c=add(a,b);
---
}
int add(int x, int y)
{
---
return(z);
}
```

o/p
Enter 2 values
5 4
Sum is 9

Function with no arguments but values are returned from the called function

Syntax

```
main()
{
---
c=fn();
---
}
fn()
{
---
return(c);
}
```

Program

```
#include<stdio.h>
main()
{
int add();
c=add();
Printf("Result is %d", c);
}
int add()
{
int a,b,c;
Printf("Enter a & b");
Scanf("%d %d", &a, &b);
c=a+b;
return(c);
}
```

```
main()
{
---
c=add();
---
}
int add()
{
---
return(c);
}
```

o/p Enter a & b
5 3
Result is 8

Passing Arrays to function

Arrays can also be arguments of functions when an array is passed to a function, the address of the array is passed and not the copy of the complete array.

When a function is called with the name of the array as the argument the address to the first element in the array is

handed over to the function. When an array is a function argument, only the address of the array is passed to the function called. It modify the contents of the array.

Syntax

```
data_type fn_name (datatype *);
main()
{
fn_name (arr_name);
}
data_type fn_name (datatype * arr_name)
{
}
```

Ex:

```
#include <stdio.h>
main()
{
int n,m, a[100], i;
int max (int*, int);
Printf ("Enter the number of elements:");
Scanf ("%d", &n);
Printf ("Enter the elements:");
for (i=0; i<n; i++)
{
Scanf ("%d", &a[i]);
}
m = max(a,n);
Printf ("The maximum element is %d", m);
}
int max (int* arr, int num)
{
int max_value, j;
max_value = arr[0];
for (j=1; j<num; j++)
{
if (arr[j] > max_value)
max_value = arr[j];
}
return max_value
}
```

O/P

```
Enter the number of elements in an array:3
Enter the array elements
3
1
2
Maximum element is 3
```

Recursion

A recursive function is one that calls itself either directly through another function. Recursion is a process by which a function calls itself repeatedly, until some specified condition

has been satisfied. It is used for repetetive computations in which each action is stated in terms of a previous result.

Syntax

```
data type fn_name ()
{
  fn_name ();
}
```

Example : calculating factorials

```
long int fact (int);
main ()
{
int n;
long int f;
Printf ("Enter the value of n");
Scanf ("%d", &n);
f = fact (n);
Printf ("Factorial of %d is %d", n, f);
}
long int fact (int i)
{
long int f1 = 1;
if (i <= 1)
return (1);
else
f1 = i * fact (i-1);
return (f1);
}
```

O/P

Enter the value of n : 3
Factorial of 3 is 6

Classification of Recursion

Recursion is classified according to

1. whether the function calls itself directly or indirectly
   a). Direct Recursion
   b) Indirect Recursion
2. whether there is any pending operations on return from recursive call.
   a) Tail Recursion
   b) Non-Tail Recursion
3. Based on the pattern of recursive call
   a) Linear Recursion
   b) Binary Recursion
   c) n-ary Recursion

Direct & Indirect Recursion

a) Direct Recursion	b) Indirect Recursion
It occurs when a function calls itself.	It occurs when a function calls another function which in turn calls the original function
Simple & commonly used	

Syntax

```
func()
{
  ....
  func();
  ....
}
```

```
f1()
{
  ...
  f2();
  ...
}
f2()
{
  ...
  f1();
}
```

2a Tail Recursion

A recursion in which the last operation of a function is a recursive call i.e) the recursive call is the last thing done by the function.

No need to keep record of the previous state i.e) no pending operations to be performed on return It eliminates the need to store the intermediate result.

Syntax

```
fn()
{
  ---
  fn();
}
```

Ex

```
main()
{
 int fun(int);
 int n=3;
 fun(3);
}
int fun(int n)
{
 if(n==0)
 return;
 else
 printf("%d",n);
 return fun(n-1);
}
```

o/p

3 2 1

b) Non-tail Recursion

A recursive call is not the last thing done by the function i.e) pending operations to be performed on return

Syntax

```
fn()
{
  ---
  fn();
  ---
}
```

Example

```
main()
{
 int fun(int);
 int n=3;
 fun(3)
}
int fun(int n)
{
 if(n==0)
 return;
 fun(n-1);
 printf("%d",n);
}
```

o/p

1 2 3

a) Linear Recursion

A linear recursive function makes only one recursive call

Ex:

```
#include<stdio.h>
main()
{
int n,f;
Printf("Enter a number");
Scanf("%d", &n);
f=fact(n);
Printf("factorial is %d",f);
}
int fact(int n)
{
if(n==0)
return 1;
else
return (n*fact(n-1));
}
```

o/p

Enter a number: 5
factorial is 120

b) Binary Recursion

A binary recursive function calls itself twice.

Syntax

```
fn()
{
...
fn();
...
fn();
...
}
```

Example

```
main()
{
int n,f;
Printf("Enter a number");
Scanf("%d", &n);
f=fib(n);
Printf("fibonacci term %d",f);
}
int fib(int n)
{
if(n==1)
return 0;
if(n==2)
return 1;
else
return fib(n-1)+fib(n-2);
}
```

o/p

Enter a number: 5
fibonacci term: 3

c) n-ary Recursion

Most general form of recursion. It is used in generating permutations.

Pointers

Pointer is a variable that holds the address of another variable

Every pointer variable takes the same amount of memory space ie) 2 bytes

Declaration of a pointer

Syntax type specifier * identifier;

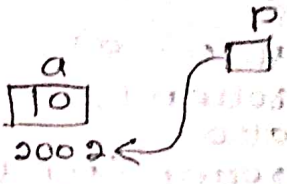
Ex: int *p;
type_specifier → type of the object referred
* → punctuator, read as "pointer to"
identifier → name of the pointer variable

pointer operators
operations on pointer / Accessing the pointer variable
- Reference operator
- Dereference operator

i) Referencing operation
A pointer variable is made to refer to an object with the help of reference or address of operator (&)
Reference operator is a unary operator and the operands are of arithmetic type or pointer type.

Syntax
data type *pt_var, var1;
pt_var = & var1

Ex:
int *p, a=10;
p = &a;



ii) Dereferencing operation
The object pointed or referred by a pointer can be accessed by dereference operator or indirection operator or value at operator (*).

Syntax
ptvar = & var1
var2 = * ptvar

Ex:
int *p, a, b;
p = & a;
b = *p.

* → used to get the value from the memory

Assigning to a pointer / Initialization of pointer
1. A pointer can be assigned or initialized of with the address of an object.
Ex: int *p;
p = &a;

2. A pointer to a type cannot be assigned or initialized with the address of an object of another type.
Ex:
int *p;
float a;
p = &a; // not valid

3. A pointer can be assigned or initialized with another pointer of the same type.

Pointers to pointers
Pointer variable can store the address of a pointer variable
Syntax:
int **pt;

Ex: int *p1, **p2, x=100;
p1 = &x;
p2 = &p1;

pointer Arithmetic.

Arithmetic operations can be applied to pointers in a restricted form.

1. Addition operation

- An expression of integer type can be added to an expression of pointer type

Ex:

```
int *p;      // p = 2000
p = p+1;     // p = 2002
float *p;    // p = 2000
p = = p+1;   // p = 2004
```

- Addition of two pointers is not allowed
- Addition of a pointer and an integer is commutative

ie) p+1, 1+p } same.

2. Subtraction operation

A pointer and an integer can be subtracted.

Ex:

```
int *p;      // p = 2000
p = p-1      // p = 1998
float *p;    // p = 2000
p = p-1      // p = 1996
```

Subtraction of two pointers. Subtraction of a pointer and an integer is not commutative.

ie) p-1, 1-p } not same

3. Increment operation

Increment operation can be applied to an operand of pointer type.

Ex:

```
int *ptr, *p; // ptr = 2000
p = ptr++;    // ptr = 2002   p = 2000
p = ++ptr;    // ptr = 2004
```

4. Decrement operation

Decrement operation can be applied to an operand of pointer type

```
int *ptr, *p;   // ptr = 2000
p = ptr--;      // ptr = 1998
p = --ptr;      // ptr = 1996
```

5. Relational operation

A pointer can be compared with the pointer of the same

type or with 0. The result is either true or false ie)) a (

Ex: int *p1, *p2, r; // p1 = 2000 p2 = 2004
r = p1 < p2; // r = 1

Illegal pointer operations
- Addition of two pointers is not allowed
- only integers can be added to pointers
- Multiplication & division operation are not allowed.
- Bitwise operators are not allowed.
- pointer of one type cannot be assigned to another type

Void pointer
- void is one of the basic data type
- Void means nothing
- It is not possible to create an object of type void but it is possible to create a pointer to void.
- Such a pointer is known as void pointer and has the type void *

Ex:
```
void *ptr;
```

Operations on void pointer
- A pointer to any type of an object can be assigned to a void pointer.
- void pointers can be compared for equality & inequality
- void pointers cannot be dereferenced
- pointer arithmetic is not allowed.

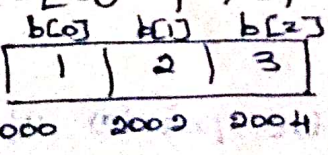
Null pointer
A null pointer is a pointer that does not point anywhere. It doesn't hold the address of any object.

Ex:
```
int *ptr = 0;
int *ptr = Null;
```

Null = Symbolic constant with value 0.

Arrays and pointers
There is a strong relationship between pointers and arrays. Any operation that can be achieved by array subscripting can also be done with pointers. The expression of the form E1[E2] is automatically converted into the expression of the form * (E1 + E2) ie) E1[E2] => *(E1 + E2)

Ex: int b[3] = {1, 2, 3};



The name of the array refers to the address of the first element of the array.

```
int *x;
x = x + 1;
```

Base address is incremented by 2.

Ex:

```
#include <stdio.h>
main()
{
int a1[5]={10,15,20};
Printf("Elements of an array: %d %d %d", a1[0], a1[1], a1[2]);
Printf("Elements of an array: %d %d %d", *(a1+0), *(a1+1), *(a1+2));
}
```

O/P

Elements of an array: 10 15 20
Elements of an array 10 15 20

Array of pointers

An array of pointers is a collection of addresses. All pointers in an array must be of same type.

Syntax

```
*a[] = {&v1 ..}
```

Ex:

```
#include <stdio.h>
main()
{
int a=10, b=20, c=30;
int *a1[] = {&a, &b, &c};
Printf("Elements are %d %d %d", a, b, c);
Printf("Elements are %d %d %d", *a1[0], *a1[1], *a1[2]);
}
```

O/P

Elements are 10 20 30
Elements are 10 20 30

Pointer to an array

Create a pointer that points to a complete array instead of pointing to the individual elements of an array. Such pointer is known as pointer to an array.

Syntax

```
datatype (*variable_name)[size];
```

Ex:

```
int *ptr[2];
```

↓ size of array variable.

Ex.

```
#include <stdio.h>
main()
{
int a[2][2] = {10,15,20,25};
int (*ptr)[2] = a;
Printf("Elements in row1: %d %d", a[0][0], a[0][1]);
Printf("Elements in row2: %d %d", ptr[1][0], ptr[1][1]);
}
```

O/P
Elements in row1 10 15
Elements in row2 20 25

Advantages of using pointers
- Enable to access the memory directly
- Increases the execution speed of the program
- Saves memory space

Parameters passing

Argument passing methods

Depending upon whether the values or addresses are passed as arguments to a function, the argument passing methods are classified as:
- Pass by value
- Pass by Address (Reference)

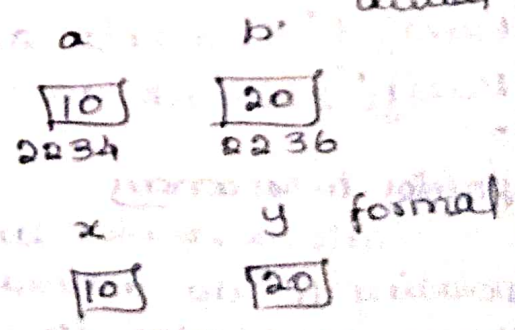
Pass by value

call by value

passing arguments by value.

The values of actual arguments are copied to the formal parameters of the function

The changes made in the values of formal parameters inside the called function are not reflected back to the calling function.



Program:

```
#include <stdio.h>
#include <conio.h>
main()
{
int a,b;
void swap(int, int);
Printf("\n Enter the value of A & B");
Scanf("%d%d", &a, &b);
Printf("\n Before swapping: A=%d, B=%d\n", a, b);
Swap(a,b);
Printf("\n After swapping - In main function\n A=%d, B=%d\n", a, b)
getch();
}
```

```
void swap(int x, int y)
{
x = x+y;
y = x-y;
x = x-y;
printf("\n After swapping-In swap function \n x=%d, y=%d\n",
                                                    x,y);
}
```

swap
x = x+y;
y = x-y;
x = x-y;

x	y
20	10
1005	1014

output
Enter the value of A & B: 10 20
Before swapping: A=10, B=20
After swapping - In swap function x=20, y=10
After swapping - In Main function A=10, B=20

a,b → actual parameters
x,y → formal parameters

- changes only in the swap function itself.
- On the execution of the function call, the values of actual parameters a & b are copied into the formal parameters x & y
- Formal parameters are allocated at separate memory locations.
- On returning from the called function, the formal parameters are destroyed and the access to the actual parameters gives values that are unchanged.

Pass by Reference

call by reference

The addresses of the actual parameters are passed to the formal parameters of the function.

The changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function i.e) change in formal parameters affects the actual parameters.

Program:

```
#include <stdio.h>
#include <conio.h>
main()
{
int a,b;
void swap(int *, int *);
printf("\n Enter the value of A & B: ");
scanf("%d %d", &a, &b);
printf("\n Before swapping: A=%d, B=%d\n", a,b);
```

```
swap( &a, &b);
Printf("\n After swapping - In main function \n A = %d, B = %d",
                                                    a, b);
getch();
}
void swap(int *x, int *y)
{
int t;
t = *x;
*x = *y;
*y = t;
Printf("\n After swapping - In swap function \n x = %d, y = %d \n",
                                                    *x, *y);
}
```

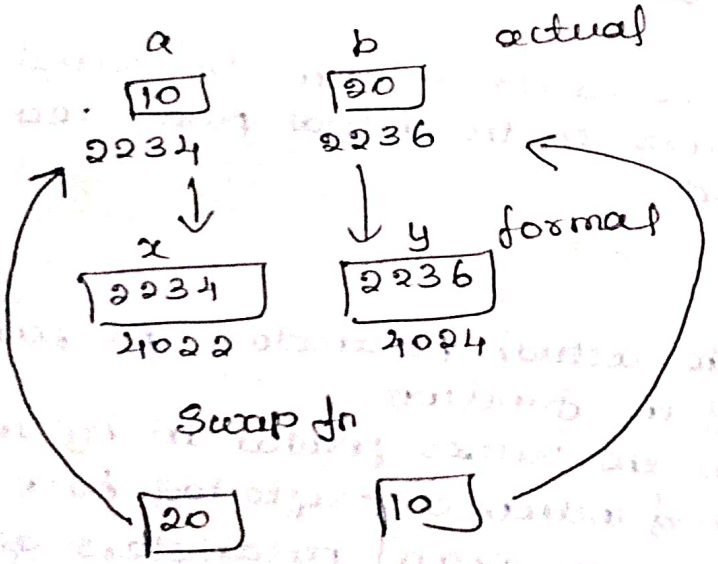
output

Enter the value of A & B : 10 20

Before swapping : A = 10, B = 20

After swapping - In swap function x = 20, y = 10

After swapping - In Main function A = 20, B = 10



Exercise programs

Calculate the total amount of power consumed by 'n' devices (passing an array to a function

```
#include <stdio.h>
#include <stdlib.h>
int calc_electricity();
devices(int n[], int size);
int main()
{
int size, n[50], s;
Printf("enter the size of an array");
Scanf("%d", &size);
s=devices(n, size);
Printf("The values is", s);
return();
}
int calc_electricity(int unit)
{
Printf("enter total units consumed \n");
Scanf("%d", &unit);
double amount;
if(unit>=1) && (unit<=50)) // between 1-50 units
{
amount = units * 1.50;
}
else if((unit >50) && (unit <=150))
{
amount=((50 * 1.5)+(unit-50) * 2.00);
}
else if((unit >150) && (unit <=250))
{
amount = (50 * 1.5)+((150-50) * 2.00)+(unit -150) * 3.00;
}
else if(unit>250)
{
amount = (50 * 1.5)+((150-50) * 2.00 + ((250-150) * 3.00)+(unit -250
                                                          * 4;
}
else {
Printf("No usage");
amount = 0;
}
return amount;
}
```

```
int devices(int n[], int size)
{
int total=0;
int i;
int unit=0;
int p;
for(i=0;i<size;i++)
{
p=calc_electricity(unit);
Printf("the amount of one bill %d is", p);
n[i]=p;
total=total+n[i];
Printf("the total amount of n devices is %d", total);
}}
```

output

```
Enter the number of devices: 3
enter total unit consumed: 100
The total amount of 1 device is 425, Total amount of n devices
                                                is 425
enter total unit consumed: 250
The total amount of 1 device is 575
Total amount of n device is 1000
enter total unit consumed: 200
The total amount of 1 device is 425
Total amount is 425
Total amount of n devices is 1425
```

Menu-driven program to count the numbers which are divisible by 3, 5 and both (passing an array to a function)

```
#include<stdio.h>
int menudriven(int a[]);
int main()
{
int k,s,i;
int a[3]={0,1,2};
menudriven(a);
}
int menudriven(int a[])
{
int i,n,s,p,j;
Printf("enter the value of n");
Scanf("%d", &n);
for(j=0;i<3;i++)
```

```
{
switch(a[j])
{
case 0:
for(i=1;i<=n;i++)
{
if(i%5==0)
{
p=p+1;
}}
Printf("\n the total numbers divisible by 5 is %d", p);
case 1:
p=0;
for(i=1;i<=n;i++)
{
if(i%.3==0)
{
p=p+1; }}
Printf("\n the total numbers divisible by 3 is %.d", p);
case 2:
p=0;
for(i=1;i<=n;i++)
{
if(i%.3==0 && i%5==0)
{
p=p+1; }
Printf("\n the total numbers divisible by both are %.d", p);
}}}
```

output

```
enter the value of n: 100
The total numbers divisible by 5 is : 20
The total numbers divisible by 3 is : 33
The total numbers divisible by both is : 6
```

UNIT - V

Structures

Introduction to structures - Declaration - Initialization - Accessing the members - Nested structures - Array of structures - Structures and functions - passing an entire structure - Exercise program- compute the age of a person using structure and functions (passing a structure to a function) - compute the number of days an employee came into to the office by considering his arrival time for 30 days (use array of structures and function)

Unit - IV
Structures and Unions

Introduction
C language provides a rich set of primitive and derived data types for the efficient storage and manipulation of data. Using C language new data types can be created. These data types are known as user-defined data types and created by using structures, unions and enumeration.

Need for structure Datatype / uses of structures
- It allows grouping together of different type of elements
- Complex data types can be handled using nesting of structures
- Structures can be used to defined records to be stored in files
- It gives flexibility to programmers to define their own data types as per the requirement
- It is also possible to create structure pointers

structure
It is a collection of variables of different data types grouped under a single name.

Ex: Student : name, roll_no, marks

There are 3 aspects of working with structures
- Defining a structure
- Declaring variables & constant of newly created type
- Using & performing operations on objects of structure type

structure Definition
A structure definition consists of the keyword struct followed by an optional identifier name known as structure tag-name and a structure list enclosed within the braces.

general form

```
struct structure_name
{
type membername 1;
type membername 2;
 ---
 ---
};
```

Eg:

```
struct book
{
char title[25], author[25];
 int pages;
 float price;
};
```

- Structure definition can have an infinite number of members
- A structure definition can't contain an instance of itself.
- A structure definition does not reserve any space in the memory.

Eg
```
struct book
{
 int pages=10; // Not valid
};
```
if a structure definition does not contain a structure tag name then the created structured is unnamed. It is also known as anonymous structure type

Declaring structure objects/variables
Variables & constants of the created structure type can be declared either at the time of structure definition or after the structure definition

General form
struct structure_name identifier [=initialization_list];
[=initialization_list] is optional
(or)
struct structure_name v1, v2, ... vn;

Ex:
```
struct book
{
 char title[20];
 int pages;
 float price;
};
struct book b1, b2, b3;
```

operations on structures
The operations that can be performed on an object of structure type can be classified into 2 types.
- Aggregate operations - operates on the entire operand as a whole
- Segregate operations - operates on the individual members of a structure object.

Aggregation operation
- Accessing members of an object of structure type
- Assigning a structure object to a structure variable
- Address of a structure object
- Size of a structure

Accessing members of an obj of structure type
- Direct member access operator (. dot operator)
- Indirect member access operator (-> arrow operator)

Initialization of structures
The member of a structure can be initialized to constant value

by enclosing the values to be assigned within the braces after the structure definition

Syntax

```
struct struct_name
{
  member1;
  member2;
  .
  .
} struct_variable = { constant1, constant2, ..};
```

Ex:

```
struct date
{
int date;
int month;
int year;
} independence = {15,08,1947};
        or
struct date independance = {15,08,1947};
```

Accessing Structure members

The members of the structures can be accessed by using the Structures variable along with the (.) dot operator.

Syntax

Variable name . member name ;

Ex:

```
struct book
{
  int id;
  char name[20];
};
struct book b1;
```

For accessing the structure members from the above example

```
b1.id;
b1.name;
```

The Structure can be defined either before main() as globally or inside main() locally.

Program.

```
#include<stdio.h>
struct book
{ int id;
  char name[20];
  char author[15];
};
main()
{
struct book b1;
Printf("\n Enter the book id, book name\n");
Scanf("%d\n %.s\n", &b1.id, b1.name);
```

O/P

```
Enter the book id, book name
101
Maths
Book id is = 101
Book name is : Maths
```

```
Printf("\n Book id is = %d", b1.id);
Printf("\n Book name is = %s", b1.name);
```

Nested structures (structure with in a structure)

A structure can be declared within another structure. Some times it is required to keep a compound data items within another compound data item is called structure within structure or it means nesting of structures

Syntax:

```
Stud stud_name1
{
 dad 1;
 dad 2;
 ...
 dad n;
};
Stud stud_name2
{
dad 1;
dad 2;
stud stud_name1 variable_name1;
...
dadn;
};
```

Ex:

```
#include <stdio.h>
Stud date
{ int date, month, year;
};
Stud stu_data
{ char name[20];
  stud date dob;
};
main()
{ stud stu_data s={"vino", {01, 03, 1992}};
Printf("\n Name %s", s.name);
Printf("\n\n Date of birth : %d-%d-%d", s.dob.date,
                                        s.dob.month, s.dob.year);
return;
}
```

o/p

Name Vino

Date of Birth : 01-03-1992

Pointer and structures

Assign pointers to structures. The pointer variable that holds the address of a structure. It takes 4 bytes of memory.

Declaring a pointer to a structure

Syntax

```
Stud studname
{
 member 1;
 member 2;
 :
 member n;
};
main()
{
stud studname *ptr, var;
ptr = &var;
...
}
```

Accessing the members of the structures

(*ptr).member1 (or) ptr->member1

Ex: ptr->regno;

Initialization

Syntax

(*ptr).member-name = constant; (or) ptr->member-name = constant;

Ex: ptr->regno = 47;

Program for printing Employee details

```
struct employee
{
int idno;
float salary;
char name[50];
};
main()
{
struct employee *emptr, e;
emptr = &e;
Printf("enter employee id no");
Scanf("%d", &emptr->idno);
Printf("Enter employee name");
Scanf("%s", emptr->name);
Printf("Enter employee salary");
Scanf("%f", &emptr->salary);
Printf("The employee details are");
Printf("idno is %d", emptr->idno);
Printf("Name is %s", emptr->name);
Printf("Salary is %f", emptr->salary);
}
```

o/p

```
Enter employee idno : 100
Enter employee name : Raj
Enter employee salary : 100000
The Employee details are
id no is 100
Name is Raj
Salary is 100000
```

Arrays of structures

The C language permits to declare an array of structure variable. If we want to handle more records within one structure we need not specify the number of structure variable.

Syntax

```
struct structname
{
dec1;
dec2;
...
} variablename[size];
```

```
};
Struct marks
{
   int sub1;
   int sub2;
   int sub3;
};
main()
{
Struct marks student[3]={{95,90,89},{65,63,70},{87,76,61}};
}
```

Difference between Array and Structure

Array	Structure
* An array is a collection of related data elements of same type * An array is derived data type * An array behaves like a built in data type * An array can be increased or decreased.	* Structure can have elements of different types * Structure is a user-defined on * It must be declared & defined * A structure element can be added if necessary

Self Referential Structures.

A structure containing a member that's a pointer to the same structure type. one or more pointers pointing to the same type of structure as their member. It is used in dynamic data structure such as trees, linked list etc.

Syntax:

```
Struct node
{
int data;
Struct node *next;
};
Struct node
{
int data;
char value;
Struct node *link;
};
main()
{
Struct node obj1;
obj1.link = Null;
obj1.data = 10;
obj1.value = 20;
Struct node obj2;
obj2.link = Null;
obj2.data = 30;
obj3.value = 40;
obj1.link = &obj2;
Printf("%d", obj1.link->data);
Printf("%d", obj1.link->value);
}
```

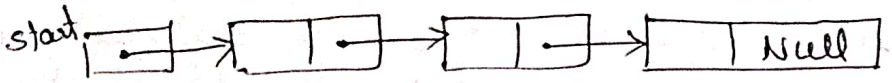
o/p

30 40

Self-referential structures are very useful in applications that involve linked data structures.

Linked data structure

- Each component within the structure includes a pointer indicating where the next component can be found.
- Relative order of the components can easily be changed by altering the pointers
- Individual components can easily be added or deleted by altering the pointers



Dynamic memory allocation

The ability for a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed

Need

Using Arrays

- possibility of overflow
- C does not check bounds
- Wastage of space

Dynamic memory allocation

Required memory allocation at run time

Static memory allocation

When fixed arrays are used. Memory allocated at compile tim

Dynamic memory allocation is the way to defer the decision of how much memory is necessary until the program is actually running or give back memory that the program no longer needs. Heap is used. It is used to allocate & deallocate dynamic heap memory.

Types of functions

- malloc()
- calloc()
- free()
- realloc()

malloc (Memory Allocation)

It allocates a new area in memory of the number of bytes and stores a pointer to the allocated memory. It requests a contiguous block of memory of the size in the heap.

Syntax

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

Ex:
```
int *ptr;
ptr = (int *) malloc (5 * sizeof (int));
                          └> 4 bytes
```

ptr = 

← 20 bytes of memory →

20 bytes of memory block is dynamically allocated to ptr. If space is insufficient, allocation fails and returns a Null pointer.

calloc () (contiguous Allocation)

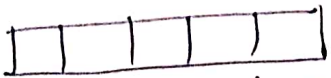
It is similar to malloc () but initializes the memory to zero. Dynamically allocates the specified number of blocks of memory of the specified type.

Syntax

```
ptr = (cast_type *) calloc (n, element_size);
```

Ex

```
int *ptr = (int *) calloc (5, sizeof (int));
```

ptr =  20 bytes of memory

5 blocks of 4 bytes, each is dynamically allocated to ptr.

free ()

It is used to dynamically deallocate the memory. It takes a pointer to a heap block earlier allocated by malloc () and returns that block to the heap for reuse. After free (), the client should not access any part of the block.

Syntax

```
free (ptr);
```

It frees the space allocated in the memory pointed by p

Ex

```
main ()
{
int *ptr, *ptr1;
int n,i;
n=5;
Printf ("number of elements %d", n);
ptr = (int *) malloc (n * size of (int));
ptr1 = (int *) calloc (n, size of (int));
if (ptr == Null || ptr1 == Null)
{
Printf ("Memory allocated);
free (ptr);
Printf ("Memory successfully freed }
```

O/P

Number of elements : 5
Memory allocated
Memory successfully freed.

realloc() (re-allocation)

It dynamically change the memory allocation of a previously allocated memory. It takes an existing heap block and tries to reallocate it to a heap block of the given size which may be larger or smaller than the original size of the block. It returns a pointer to the new block.

Reallocation of memory maintains the already present value and the new block will be initialized with default garbage value

Syntax

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

Ex:

```
int *ptr = (int*) malloc (5 * sizeof (int));
```

ptr = [ ] ← 20 bytes memory

```
ptr = realloc (ptr, 10 * sizeof (int));
```

ptr = [ ] ← 40 bytes of memory

stdlib.h

To allocate memory dynamically, library functions are malloc(), calloc(), free(), realloc() are used.

These functions are defined in stdlib.h header or alloc.h file. So include these header files

```
#include <stdlib.h>
        (or)
#include <alloc.h>
```

Singly linked list

Linked list

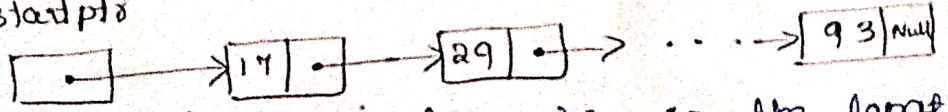
It is a linear collection of self-referential structures called nodes, connected by pointer links. A linked list is accessed via a pointer to the first node of the list.

- Subsequent nodes are accessed via the link pointer member stored in each node.
- The link pointer in the last node of a list is set to NULL to mark the end of the list.

Linked list elements are not stored at the contiguous location the elements are linked using pointers. Each structure of the list consists of 2 fields

- data item
- address of the next item in the list (pointer)

Graphical Representation



Linked lists are dynamic, so the length of a list can increase or decrease at execution time.

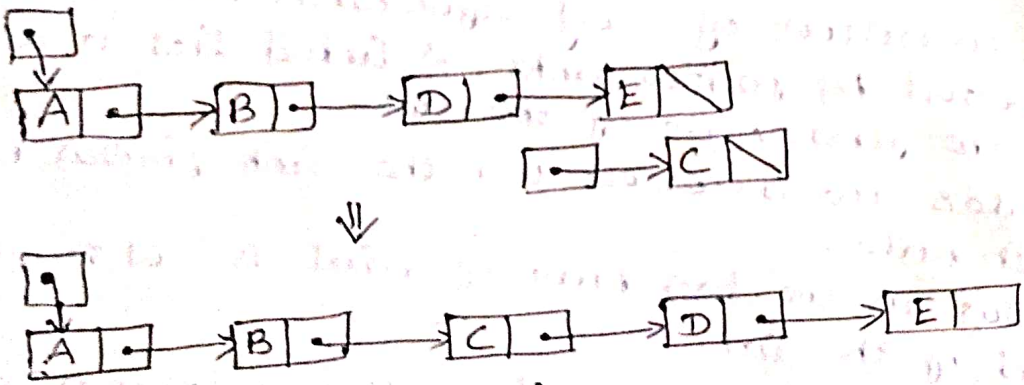
- Doubly linked list
- Circular linked list

Creation of a singly linked list - structure definition

```
struct node
{
int data;
struct node *next;
};
main()
{
typedef struct node *position;
position P,L;
L->next = NULL;
}
```

operations
- Creation
- Traversing
- Insertion
- Deletion
- Concatenation
- Searching

Insertion



Insertion at the beginning
- obtain space for new node
- Assign data
- Set the next field of the new node to the beginning of the li
- change the reference pointer to pointer to the new node

Ex:

```
position insert_beg(int val, position L)
{
  position new;
  new = malloc (sizeof (struct node));
```

```
if (new != NULL)
{
new->data = val;
new->next = l->next;
l->next = new;
}
```

Insertion at last

- obtain space for new node
- Assign data
- Set the next field of the new node to NULL

Ex:

```
Position insert_last (int val, position l)
{
position new, p;
new = malloc (size of (stud node));
if (new != NULL)
{
new->data = val;
new->next = NULL;
p = l->next;
while (p->next != NULL)
    p = p->next;
p->next = new;
}
```

Insertion at the middle

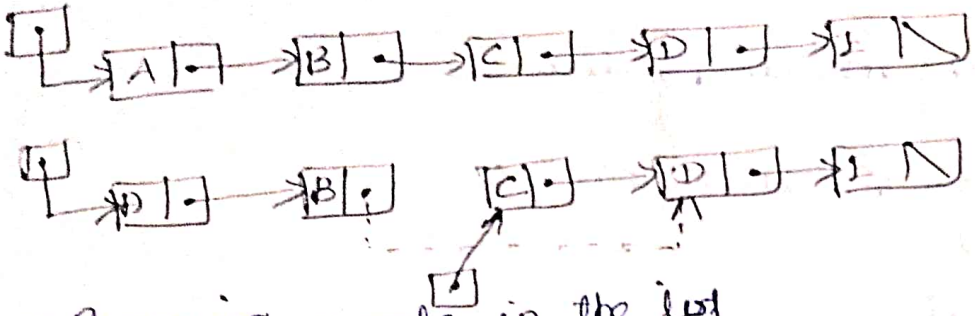
- obtain space for new node
- Assign data to the data field of the new node
- Enter the desired for after which the node has to be inserted
- Move to the corresponding node.
- Set the next field of the new node to pointing to the next of the corresponding node.

Ex:

```
position insert_mid (int val, position l)
{
position new, p;
int c;
new = malloc (sizeof (stud node));
if (new != NULL)
{
Printf ("Enter the value after which the value to be inserted");
Scanf ("%d", &c);
p = find (c, l);
if (p != NULL)
{
new->data = val;
new->next = p->next;
p->next = new;
}}}
```

```
Position find (int x, position L)
{
 position p;
 p = L->next;
 while (p != NULL && p->data != x)
    p = p->next;
 return p;
}
```

Deletion



* Removing a node in the list
* Deletion can be done at
  - beginning
  - middle
  - last

Deletion at beginning
Make the head pointing to the second node in the list.

Ex:

```
position delete_beg (position L)
{
 position temp;
 if (L->next == NULL)
 Printf ("The list is empty");
 else
 { temp = L->next;
   L->next = temp->next;
 Printf ("Deleted element is %d", temp->data);
 free (temp);
 } }
```

Deletion at last
Move to the last node in the last. Make the last node next pointer to NULL

Ex:

```
position delete_last (position L)
{
 position temp, P;
 if (L->next == NULL)
 Printf ("The list is empty");
```

```
else
{
temp = L->next;
L->next = temp->next;
Printf ("Deleted element is %d ", temp->data);
free (temp);
}}
```

Deletion at last

Move to the last node in the last

Make the last node next pointer to NULL

Program

```
position delete_last (position L)
{
position temp, P;
if (L->next == NULL)
Printf ("The list is empty");
else
{
P = L;
while (P->next != NULL)
P = P->next;
temp = P;
P->next = NULL;
Printf ("Deleted element is %d ", temp->data);
free (temp);
}
```

Deletion at middle:

- Move to the node containing the element to be deleted.
- Make the previous node next pointer to the node after that node

Program

```
position delete_mid (position L)
{
position temp, P;
int x;
if (L->next == NULL)
Printf ("The list is empty");
else
{
Printf ("Enter the element to delete");
scanf ("%d", &x);
P = find_prev (x, L);
if (! is last (P))
{
temp = P->next;
```

```
P->next = temp->next;
Printf ("Deleted element is %d", temp->data);
free (temp);
}}}
Position find-prev (int x, position L)
{
Position P;
P=L;
while (P->nex != NULL && P->next->data != x)
P= P->next;
return P;
}
```

Display the list

```
Position display (position L)
{
position P;
P = L->next;
Printf ("\n");
while (P != NULL)
{
Printf ("%d ->", P->data);
P = P->next;
}
Printf ("NULL");
}
```

Typedef

typedef Keyword allows the progeammer to create a new data type for an existing data type. The Alternate name is given to a Known data type which makes the code more portable.

Declaration

typedef existing data type newdata type...;

Ex:

a) 1) typedef int length;
length len, maxlen;

- length is type int
- len, maxlen are regarded as int ie) int len, maxlen

2) typedef char lower_case;
lower_case a, b, c;

b) Array & Pointers

① typedef int length;
length *lengths[];

② typedef char *string;
string p, s[50];

c) Structure

complex data type like structure can use typedef

Ex:

```
typedef struct point
{
  int x;
  int y;
};
main()
{
  typedef struct point dot;
  dot left, right;
}
```

When typedef is used to name a structure, the structure tag name is not necessary.

Ex: typedef struct

```
{ float real;
  float img;
} complex;
complex u, v;
```

Program that prints x-y coordinates of the two ends of a line

```
#include <stdio.h>
typedef struct                    //no tag name
{ int x;
  int y;
};
main()
{
typedef struct dot;
dot left, right;
Printf("Enter x & y coordinates of left \n");
Scanf("%d %d", &left.x, &left.y);
Printf("Enter x & y coordinates of right");
Scanf("%d %d", &right.x, &right.y);
```

```
Printf("left: x = %d, y = %d\n
        Right: x = %d, y = %d", left.x, left.y, right.x, right.y);
}
```

output

Enter x & y coordinates of left:

4 20

Enter x & y coordinates of right:

30 20

left: x: 4 y: 20

Right: x: 30 y: 20

union

It is a collection of variable of different data types. It is also a derived data-type which is used to represent dissimilar data items. They are used to create user defined types.

In structure a separate memory is allocated to each member while in unions all the members of union share the same memory.

characteristics of union

- Members of union have same memory location
- Collection of variables of different data types
- The keyword 'union' is used to declare a union
- Members of the union can be accessed using the dot operator.
- Size allocated is equal to the largest data member of the union
- only one union member can be accessed at a time

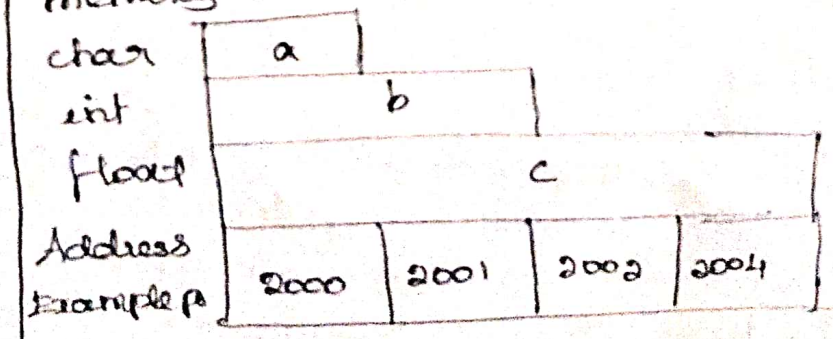
Syntax

```
union union_name
{
    union member 1;
    union member 2;
    ...
    union member n;
};
union union_name variable;
```

Eg:

```
union numbers
{
    char a;
    int b;
    float c;
};
```

memory allocation in union



Exercise

compute the age of a person using structure and function

```
#include <stdio.h>
#include <stdlib.h>
void age (int pre_m, pre_d, int pre_yea, int b_d, int b_m, int b_y)
{
int month[] = {31,28,31,30,31,30,31,31,30,31,30,31};
if (b_d > pre_d)
{
 pre_d = pre_d + month[b_m - 1];
 pre_m = pre_m - 1;
}
if (b_m > pre_m)
{
pre_y = pre_y - 1;
pre_m = pre_m + 12;
}
int final_da = pre_d - b_d;
int final_m = pre_m - b_m;
int final_y = pre_y - b_y;
Printf ("Present Age year %. months %.d Days %. d", f_y, f_m, f_d);
}
int main ()
{
 int  pre_d = 21;
 int  pre_m = 9;
 int  pre_y = 2019;
 int  b_d = 25;
 int  b_m = 9;
 int  b_y = 1996;
age (pre_d, pre_m, pre_y, b_d, b_m, b_y);
return 0;
}
```

output

Present age years : 22 months : 11 days : 26

computes the number of days an employee came late to the office by considering his arrival time for 30 days

```
#include <stdio.h>
#include <time.h>
Struct student {
 char lastName[100];
 char firstName[100];
 char *date;
 int id;
};
```

```
int main()
{
int n=1;
struct student s[n];
int x;
do {
printf("main menu:\n 1.add \n 2.ddelete \n 3. display \n4. exit\n");
scanf("%d", &x);
switch(x)
{
case 1:
for(int i=0; i<n; i++)
{
Printf("order first name \n");
scanf("%s", s[i].firstname);
Printf("enter last name \n");
Scanf("%s", s[i].lastName);
Printf("order your age \n");
Scanf("%d", &s[i].age);
time_t timer;
timer = time(NULL);
s[i].data = asctime(localtime(&timer));
}
for(int i=0; i<n; i++)
{
Printf("id\t firstName\t lastName\t age\t date\n%d\t%s\t%d\t%
                                        s[i].id, s[i].firstName,
                                        s[i].lastName, s[i].age, s[i].da
}
break;
case 2:
break;
case 3:
break;
default:
Printf("wrong choice");
break;
}
}while(x!=4);
return 0;
}
```