

UNIT-1

➔INTRODUCTION TO COMPUTER SYSTEM

Computer Definition: Computer is an electronic device which take some input process it and produce output



or

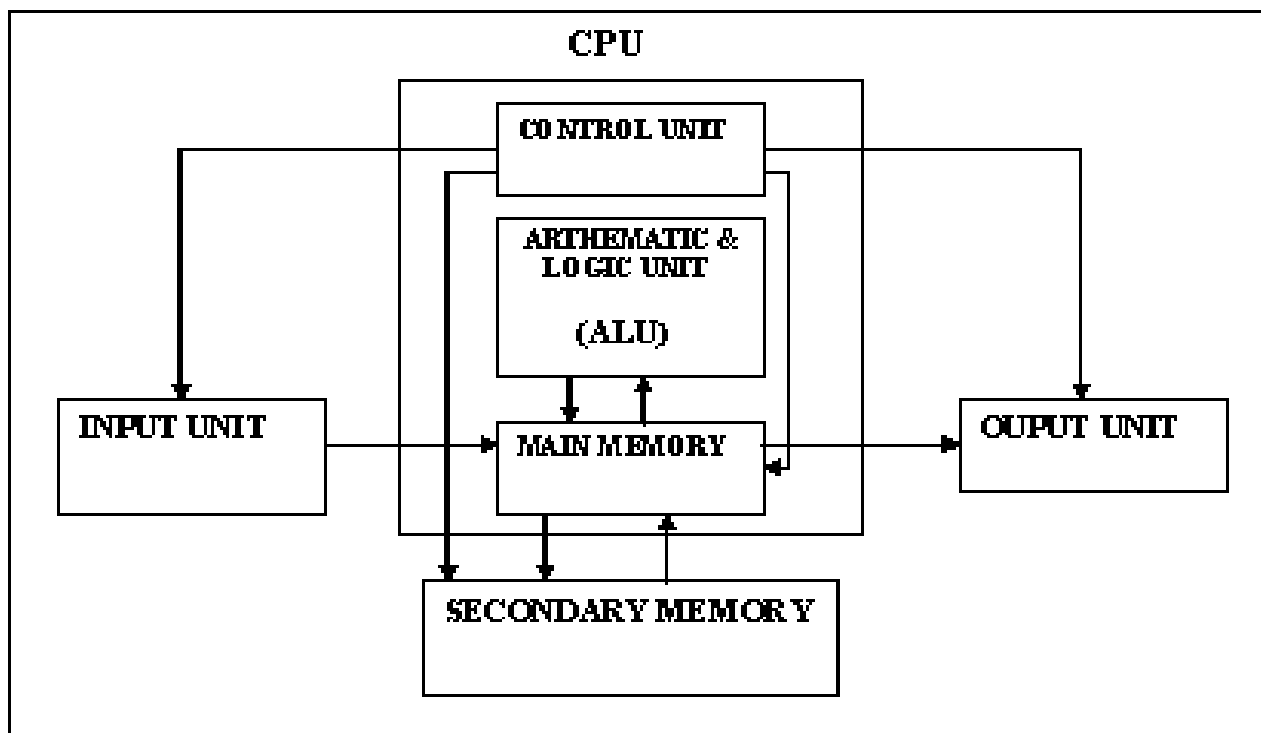
A computer is an electronic device which converts raw data into information.

Any computers consists of HARDWARE AND SOFTWARE.

Hardware:

Computer hardware is the collection of physical elements that constitutes a computer system. Computer hardware refers to the physical parts or components of a computer such as the monitor, mouse, keyboard, computer data storage, hard drive disk (HDD), system unit (graphic cards, sound cards, memory, motherboard and chips), etc. all of which are physical objects that can be touched.

COMPONENTS OF COMPUTER/BLOCK DIAGRAM OF A COMPUTER :



1. INPUT UNIT:

An input device is a hardware or peripheral device used to send data to a computer.

1. Keyboard	2. Mouse (pointing device)	3. Microphone
4. Touch screen	5. Scanner	6. Webcam
7. Touchpads	8. MIDI keyboard	9. Pen Input
10. Graphics Tablets	11. Cameras	12. Trackballs
13. Video Capture Hardware	14. Microphone	15. Joystick
16. Barcode reader	17. Digital camera	
18. Gamepad	19. Electronic Whiteboard	

2. CENTRAL PROCESSING UNIT:

CPU is the brain of a computer. It directs and controls the entire computer system and performs all arithmetic and logical operations.

It consists of

- i) Arithmetic & Logic Unit (ALU)
- ii) Computer Memory
- iii) Control Unit
- i) **Arithmetic and Logic Unit (A.L.U):**

A.L.U performs two functions

- It carries out arithmetical operations like addition, subtraction, multiplication and division .
- It performs certain logical actions based on AND and OR functions.
- ii) **Computer Memory**– Memory is storage part in computer. It is used to store the data, information, programs during processing in computer. It stores data either temporarily or permanent basis.

Types of Memory– Mainly computer have two types memory

1. Primary Memory / Volatile Memory.
2. Secondary Memory / Non Volatile Memory.

1. Primary Memory / Volatile Memory– Primary memory is internal memory of the computer. It is also known as Main memory and Temporary memory .Primary Memory holds the data and instruction on which computer is currently working. Primary Memory is nature volatile. It

means when power is switched off it lost all data.

Types of Primary Memory– Primary memory is generally of two types.

a. RAM

b. ROM

a. RAM (Random Access Memory) – It stands for Random Access Memory. RAM is known as read /writes memory. It generally referred as main memory of the computer system. It is a temporary memory. The information stored in this memory is lost as the power supply to the computer is switched off. That’s why RAM is also called “Volatile Memory”

b. ROM (Read Only Memory) – It stands for Read Only Memory. ROM is a Permanent Type memory. Its content are not lost when power supply is switched off. It is also called “Non-Volatile Memory”.

iii) Control Unit:

- It gives command to transfer data from the input device to the memory to ALU.
- It also transfers the results from ALU to the memory and onto the output device for printing.
- It stores the program in the memory, takes instructions one by one, understands them and issues appropriate commands to the other units.

3. OUTPUT UNIT:

An output device is any piece of computer hardware equipment.

Any device that [outputs](#) information from a computer is called an output device.

1. Monitor	2. LCD Projection Panels
3. Printers (all types)	4. Computer Output Microfilm (COM)
5. Plotters	6. Speaker(s)
7. Projector	

OPERATING SYSTEM:

It is the interface between the user and the computer hardware. These programs are in-built into the computer and are used to govern the control of the computer hardware components, such as processors, memory devices and input/output devices.

Some most popular OS examples are given below: windows, Linux, MacOS, Ios, Android, Ubuntu, CentOS, Solaris, Chrome OS, Fedora.

→Flowchart:

A flow chart is a step by step diagrammatic representation of the logic paths to solve a given problem.

Or

A flowchart is visual or graphical representation of an algorithm.

Algorithm:

A set of sequential steps usually written in Ordinary Language to solve a given problem is called **Algorithm**.

Example: Suppose we want to find the average of three numbers, the algorithm is as follows

Step 1 Read the numbers a, b, c

Step 2 Compute the sum of a, b and c

Step 3 Divide the sum by 3

Step 4 Store the result in variable d

Step 5 Print the value of d

Step 6 End of the program

Algorithm	Flowchart
1. A method of representing the step-by-step logical procedure for solving a problem	1.Flowchart is diagrammatic representation of Algorithm.It is constructed using different types of boxes and symbols.
2. It contains step-by-step English descriptions, each step representing a particular operation leading to solution of problem	2. The Flowchart employs a series of blocks and arrows, each of which represents a particular step in an algorithm
3. These are particularly useful for small problems	3. These are useful for detailed representations of complicated programs.

Symbols used in Flow-Charts

The symbols that we make use while drawing flowcharts as given below are as per conventions followed by International Standard Organization (ISO).

→**Oval**: Rectangle with rounded sides is used to indicate either START/ STOP of the program. ..



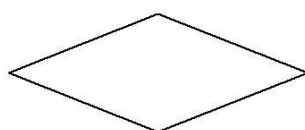
→**Input and output indicators**: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.



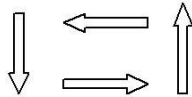
→**Process Indicators**: - Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.



→**Decision Makers**: Decision boxes are used to test the conditions or ask questions and depending upon the answers, the appropriate actions are taken by the computer. The decision box symbol is

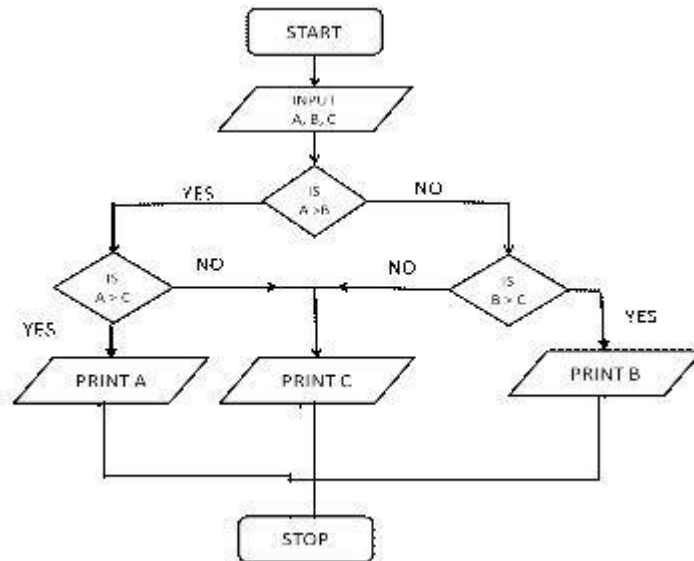


→ **Flow Lines:** Flow lines indicate the direction being followed in the flowchart. In a Flowchart, every line must have an arrow on it to indicate the direction. The arrows may be in any direction



Draw the Flowchart for the following

Draw a flowchart to find out the biggest of the three unequal positive numbers.



→INTRODUCTION to C Programming:

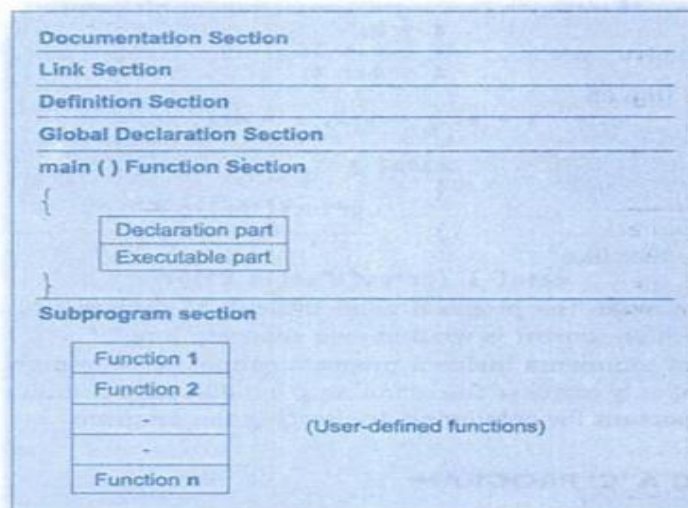
Characteristics/Features/Importance of C:

- It is a robust language whose rich set of built-in functions and operators can be used to write any complex programs.
- Programs written in 'C' are efficient and fast. This is due to variety of data types and powerful operators.
- There are 32 keywords in 'C'

- 'C' is highly portable, means 'C' programs are written on one computer can be run on another computer with little or no modifications.
- 'C' language is well suited for structured programming. It consists of function modules or blocks. This modular structure makes program debugging, testing and maintenance easier.
- 'C' program is basically a collection of functions that are supported by 'C' library. We can continuously add our own functions to 'c' library.

1. **stdio.h: I/O functions:**
 - a. **getchar()** returns the next character typed on the keyboard.
 - b. **putchar()** outputs a single character to the screen.
 - c. **printf()** as previously described
 - d. **scanf()** as previously described
2. **string.h: String functions**
 - a. **strcat()** concatenates a copy of str2 to str1
 - b. **strcmp()** compares two strings
 - c. **strcpy()** copies contents of str2 to str1
3. **ctype.h: Character functions**
 - a. **isdigit()** returns non-0 if arg is digit 0 to 9
 - b. **isalpha()** returns non-0 if arg is a letter of the alphabet
 - c. **isalnum()** returns non-0 if arg is a letter or digit
 - d. **islower()** returns non-0 if arg is lowercase letter
 - e. **isupper()** returns non-0 if arg is uppercase letter
4. **math.h: Mathematics functions**
 - a. **acos()** returns arc cosine of arg
 - b. **asin()** returns arc sine of arg
 - c. **atan()** returns arc tangent of arg
 - d. **cos()** returns cosine of arg
 - e. **exp()** returns natural logarithm e
 - f. **fabs()** returns absolute value of num
 - g. **sqrt()** returns square root of num
5. **time.h: Time and Date functions**
 - a. **time()** returns current calendar time of system
 - b. **difftime()** returns difference in secs between two times
 - c. **clock()** returns number of system clock cycles since program execution
6. **stdlib.h: Miscellaneous functions**
 - a. **malloc()** provides dynamic memory allocation, covered in future sections
 - b. **rand()** as already described previously
 - c. **srand()** used to set the starting point for rand()

➔ Basic structure of a C program:



Documentation section: Comment lines giving details of program such as program name,author etc

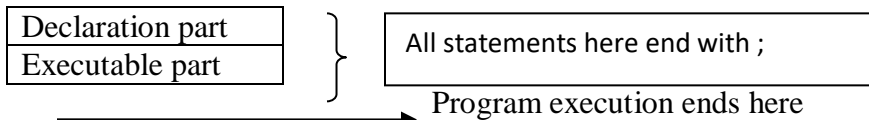
Link section: Provides instructions to the compiler to link functions from system library.

Defination section: Defines all symbolic constants.

Global declaration section: Declares global variables(variables used in more than one function.

main()

{ _____ → program execution begins at this



}

Subprogram section:contains user defined functions that are called in main

Example C-program

```
/* This Program is an illustration of a simple C-program*/
```

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

```
void main()
```

```
{
```

```
//Beginning of the program
```

```
printf("Hello wolrd!\n");
```

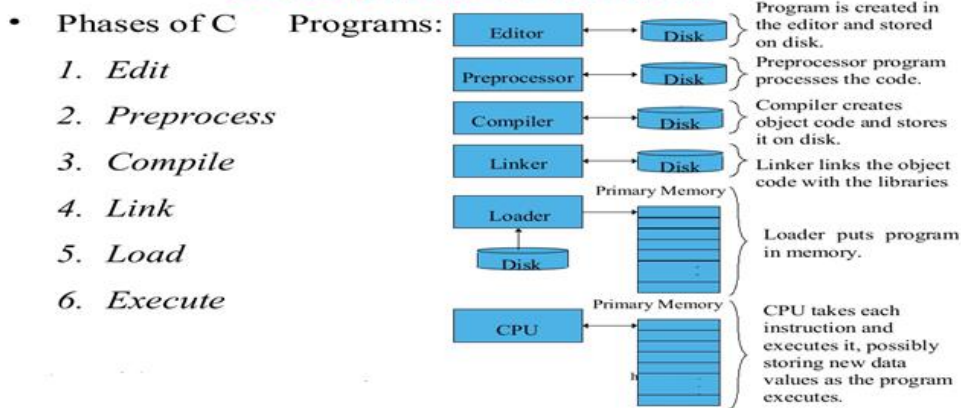
```
//End of the program
```

```
}
```

- main(): It is a special function used by the c compiler where the program starts.
 - Every program must have exactly one main function.
 - If more than one main function is used, the compiler cannot tell which one marks the beginning of program.
 - ()-Indicates that function main() has no arguments
 - Every statement in C ends with ;
-

➔ Execution Phases in C

1.14 Basics of a Typical C Program Development Environment



‘C’ programs go through six phases to be executed .

1. The first phase consists of **editing** a file. This is accomplished with an editor program. The programmer types a c program & stores the program on a secondary storage device such as a disk with .c extension
2. The programmer then gives command to compile the program. The **compiler** translates the ‘C’ program into machine language.
3. The ‘C’ **preprocessor** obeys special commands called preprocessor directives which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually consists of including other files in the file to be compiled.
4. The next phase is called **linking** C programs contains references to functions defined else where, such as in the standard libraries or in the private libraries of programmers.
5. A **linker** links the object code with the code for missing functions to produce an executable image
6. The Next phase is called **Loading**. Before a program can be executed, the program must be placed in memory. This is done by the loader. It takes the executable image from the disk and transfers it to memory.

➔ Syntax and Semantics in Compilation

Syntax:

1. It refers to the rules and regulations for writing any statement in a programming language like C/C++.

2. It does not have to do anything with the meaning of the statement.
3. A statement is syntactically valid if it follows all the rules.
4. It is related to the grammar and structure of the language.

Semantics:

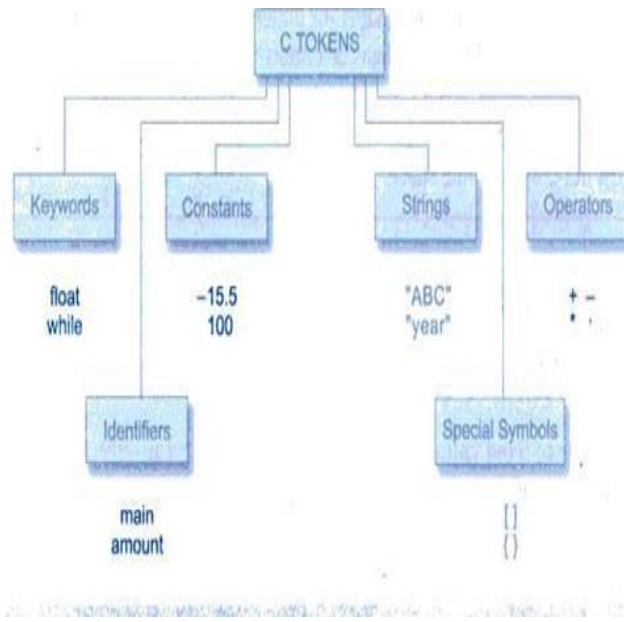
1. It refers to the meaning associated with the statement in a programming language.
2. It is all about the meaning of the statement which interprets the program easily.
3. Errors are handled at runtime.

=====

→C Tokens

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a program the smallest individual units are known as c tokens. C programs are written using tokens and the syntax of the language. C has six types of tokens as shown below:

-
-
1. Keywords.
 2. Identifiers
 3. Constants
 4. Strings
 5. Special symbols
 6. Operators



Keywords and Identifiers: Every C word is classified as either a ‘Keyword’ or an ‘identifier’

All keywords have fixed meanings and these meanings cannot be changed. All keywords must be written in lowercase. There are 32 keywords in C

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

Identifiers: It refers to all names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used.

The underscore character is also permitted in identifiers.

Backslash character constants

C supports some special backslash character constants that are used in output in output functions.

Eg: the symbol `\n` stands for newline character. A list of such backslash character constants is given in table below. These character combinations are known as escape sequences.

Table 2.5 Backslash Character Constants

<i>Constant</i>	<i>Meaning</i>
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

→ Variables

A variable is a data name that may be used to stored a data value unlike constants that remain unchanged during execution of a program, A variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.

variable names may consists of letters digits and the underscore(_) character.

Rules for variables:

1. they must begin with a letter .
2. the length must be of 32 character.
3. Uppercase and lower case are significant.
4. The variable name should not be a keyword.
5. White space is not allowed.

Valid examples	Invalid examples
John	123
Value	%
Count	(sum)
X1	25 th

Declaration of variables**Syntax:**

datatype v1,v2,v3,...vn ;

v1,v2,...vn are the names of the variables

Variables are separated by commas. A declaration statement must end with a semicolon.

Eg: int count;

float number1,number2;

Assigning/Initializing values to variables

The process of giving initial values to variables is called 'initialization'.

Values can be assigned to variables using the assignment operator = as follows:

variables_name=constant;

It is also possible to assign a value to a variable at the time the variable is declared .

It takes the following form:

datatype variable_name=constant;

Eg: int num1=10,num2=20;

char yes='x';

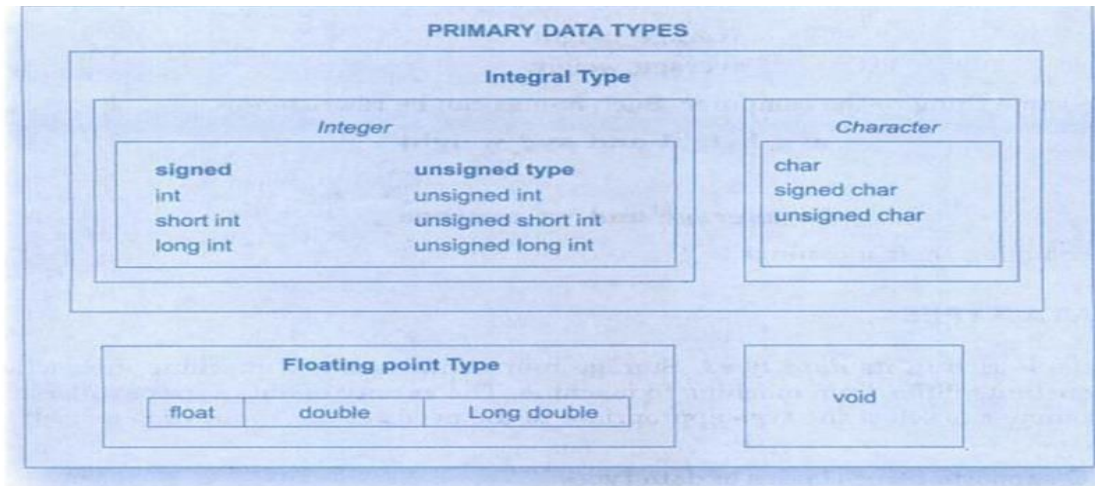
double balance=102.50;

➔Data types

C language is rich in its data types. C data types are defined as the data storage format that a variable can store a data to perform a specific operation.there are three types of data types

1. Primary (or fundamental) data types
2. User_defined data types
3. Derived data types

1. Primary data types



All C compiler support four fundamental data types.

- Integer **int** - integer: a whole number.
- Character **char** - a single character
- Floating point **float** - floating point value: ie a number with a fractional part.
- Double **double** - a double-precision floating point value.
- **void** - valueless special purpose type

INTEGER DATA TYPE:

- Integer data type allows a variable to store numeric values.
- "int" keyword is used to refer integer data type.
- The storage size of int data type is 2 byte.
- It varies depend upon the processor in the CPU that we use. If we are using 16 bit processor, 2 byte (16 bit) of memory will be allocated for int data type.
- int (2 byte) can store values from -32,768 to +32,767

CHARACTER DATA TYPE:

- Character data type allows a variable to store only one character.
- Storage size of character data type is 1. We can store only one character using character data type.

- “char” keyword is used to refer character data type.
- For example, ‘A’ can be stored using char datatype. You can’t store more than one character using char data type.

Note : To store more than one characters in a variable we have to use string .

FLOAT DATA TYPE:

- Float data type allows a variable to store real values.
- Storage size of float data type is 4. This also varies depend upon the processor in the CPU as “int” data type.
- We can use up-to 6 digits after decimal using float data type.
- For example, 10.456789 can be stored in a variable using float data type.

DOUBLE:

- Double data type is also same as float data type which allows up-to 14 digits after decimal.

<i>Data type</i>	<i>Range of values</i>
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

2. User defined datatypes

They are classified into two types:

- Type definition(typedef)
- Enumeration datatype(enum)

Type Definition(typedef)

C supports a feature known as “**type definition**” that allows user to define an identifier that would represent an existing data type. It takes the general form:

typedef type identifier;

type refers to an existing data type

identifier refers to name giving to data type

**eg: typedef int units;
typedef float marks;**

here units symbolizes int and marks symbolizes float.

the main advantage of 'typedef' is that meaningful data type names can be created for increasing the readability of the program.

Enumeration data type is defined as follows:

enum identifier{value1,value2,....value n};

identifier is a user defined enumerated datatype which can be used to declare variables that can have one of the values enclosed within the braces known as enumeration constants.

Eg: enum days{mon,tue,wed,thr,fri,sat,sun};

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants.

3. Derived data type

The datatypes that are derived from the existing primary datatypes is called as derived datatype.

Eg: Arrays

Functions

Structures

Unions

Pointers

→OPERATORS

C supports a rich set of operators. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators are classified into a number of categories. They include:

1. Arithmetic operators
2. Relational operators

3. Logical operators
4. Assignment operators
5. Increment and Decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

ARITHMETIC OPERATORS

The Arithmetic operators are

- + (Addition)
- (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo division)

Eg: 1) $a-b$ 2) $a+b$ 3) $a*b$ 4) a/b 5) $a\%b$

The modulo division produces the remainder of an integer division.

The modulo division operator cannot be used on floating point data.

Note: C does not have any operator for exponentiation.

RELATIONAL OPERATORS

Comparisons can be done with the help of relational operators. The expression containing a relational operator is termed as a relational expression. The value of a Relational Expression is either zero or 1.

Operator	meaning
1) $<$	(is less than)
2) $<=$	(is less than or equal to)
3) $>$	(is greater than)
4) $>=$	(is greater than or equal to)

5) == (is equal to)

6) != (is not equal to)

Eg: a < b or 1 < 20

LOGICAL OPERATORS

C has the following three logical operators.

&& (logical AND)

|| (logical OR)

! (logical NOT)

Eg: 1) if(age > 55 && sal < 1000)

2) if(number < 0 || number > 100)

ASSIGNMENT OPERATORS

The usual assignment operator is '='. In addition, C has a set of 'shorthand' assignment operators of the form, v op = exp;

Eg: x += y + 1;

This is same as the statement

x = x + (y + 1);

INCREMENT AND DECREMENT OPERATORS

C has two very useful operators that are not generally found in other languages. These are the increment and decrement operator:

++ and --

The operator ++ adds 1 to the operands while -- subtracts 1. It takes the following form:

++m; or m++

--m; or m—

CONDITIONAL OPERATOR

A ternary operator pair “?:” is available in C to construct conditional expression of the form:

SYNTAX:

exp1 ? exp2 : exp3;

Here exp1 is evaluated first. If it is true then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false then exp3 is evaluated and its value becomes the value of the expression.

BITWISE OPERATORS

Bitwise operator are used for manipulation of data at bit level. These operators are used for testing the bits or shifting them right or left bitwise operators may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

SPECIAL OPERATORS

C supports some special operators such as

Comma operator

Size of operator

Pointer operators(& and *) and

Member selection operators(. and ->)

→EXPRESSIONS

The combination of operators and operands is said to be an expression.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Ex: $a=x+y$;

EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form

variable = expression;

Eg:1) $x = a * b - c$;

2) $y = b / c * a$;

Rules for Evaluation of Expression

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

→ PRECEDENCE OF ARITHMETIC OPERATORS

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

High priority * / %

Low priority + -

Precedence of operator :

Precedence, in C, is the rule that specifies the order in which certain operations need to be performed in an expression. For a given expression containing more than two operators, it determines which operations should be calculated first.

In C, precedence of arithmetic operators(*, %, /, +, -) is higher than relational operators(==, !=, >, <, >=, <=) and precedence of relational operator is higher than logical operators(&&, || and !).

Example of precedence

$(1 > 2 + 3 \ \&\& \ 4)$

This expression is equivalent to:

$((1 > (2 + 3)) \ \&\& \ 4)$

i.e, $(2 + 3)$ executes first resulting into 5

then, first part of the expression $(1 > 5)$ executes resulting into 0 (false)

then, $(0 \ \&\& \ 4)$ executes resulting into 0 (false)

Output

0

➔ Associativity of a operator :

If two operators of same precedence (priority) is present in an expression, Associativity of operators indicate the order in which they execute.

Example of associativity

$1 == 2 != 3$

Here, operators `==` and `!=` have same precedence. The associativity of both `==` and `!=` is left to right, i.e, the expression on the left is executed first and moves towards the right.

Thus, the expression

above is equivalent to :

$((1 == 2) != 3)$

i.e, $(1 == 2)$ executes first resulting into 0 (false)

then, $(0 != 3)$ executes resulting into 1 (true)

1

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
->	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
-	One's-complement		
-	Unary minus (Negation)		
++	Increment		
--	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
-	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

→Type Conversion in C

Type conversion in C is the process of converting one data type to another. The type conversion is only performed to those data types where conversion is possible.

There are two types of type conversion

1. Implicit Type Conversion

Also known as 'automatic type conversion'.

- . Done by the compiler on its own, without any external trigger from the user.
- . Generally takes place when in an expression more than one data type is present. In such conditions type conversion (type promotion) takes place to avoid loss of data.
- . All the data types of the variables are upgraded to the data type of the variable with the largest data type.

bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

2. Explicit Type Conversion

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

All the data types of the variables are converted from larger data type to the lower data type.

Eg:

(type) expression

Type indicated the data type to which the final result is converted.

float a = 1.5;

(int)a;

Classification of languages:

A language is the main medium of communicating between the Computer systems and the most common are the programming languages. There are basically two types of computer languages:

- 1) Low Level Language
- 2) High Level Language

1. Low Level Language:

Low level languages are computer instructions or machine code very easily understandable by a computing machine. The main function of the Low level language is to operate, manage and manipulate the hardware and system components.

Machine Language is one of the low-level programming languages which is the first generation language developed for communicating with a Computer.

It is written in machine code which represents 0 and 1 binary digits inside the Computer string which makes it easy to understand and perform the operations.

Assembly Language

Instead of using numbers like in Machine languages here we use words or names in English forms and also symbols.

An assembly language is the most basic programming language available for any processor.

LOAD, STR, ADD

2. High Level Language:

A high-level language is a programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer.

Each instruction in the high-level language is translated into many machine language instructions that the computer can understand.

=====

→ EXPRESSIONS

The combination of operators and operands is said to be an expression.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Ex: $a = x + y$;

EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form

$\text{variable} = \text{expression};$

Eg:1) $x = a * b - c$;

2) $y = b / c * a$;

Rules for Evaluation of Expression

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

PRECEDENCE OF ARITHMETIC OPERATORS

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

High priority * / %

Low priority + -

Precedence of operator :

Precedence, in C, is the rule that specifies the order in which certain operations need to be performed in an expression. For a given expression containing more than two operators, it determines which operations should be calculated first.

In C, precedence of arithmetic operators(*, %, /, +, -) is higher than relational operators(==, !=, >, <, >=, <=) and precedence of relational operator is higher than logical operators(&&, || and !).

Example of precedence

`(1 > 2 + 3 && 4)`

This expression is equivalent to:

`((1 > (2 + 3)) && 4)`

i.e, `(2 + 3)` executes first resulting into 5

then, first part of the expression `(1 > 5)` executes resulting into 0 (false)

then, `(0 && 4)` executes resulting into 0 (false)

Output

0

Associativity of a operator :

If two operators of same precedence (priority) is present in an expression, Associativity of operators indicate the order in which they execute.

Example of associativity

$1 == 2 != 3$

Here, operators `==` and `!=` have same precedence. The associativity of both `==` and `!=` is left to right, i.e, the expression on the left is executed first and moves towards the right.

Thus, the expression

above is equivalent to :

$((1 == 2) != 3)$

i.e, $(1 == 2)$ executes first resulting into 0 (false)

then, $(0 != 3)$ executes resulting into 1 (true)

Output

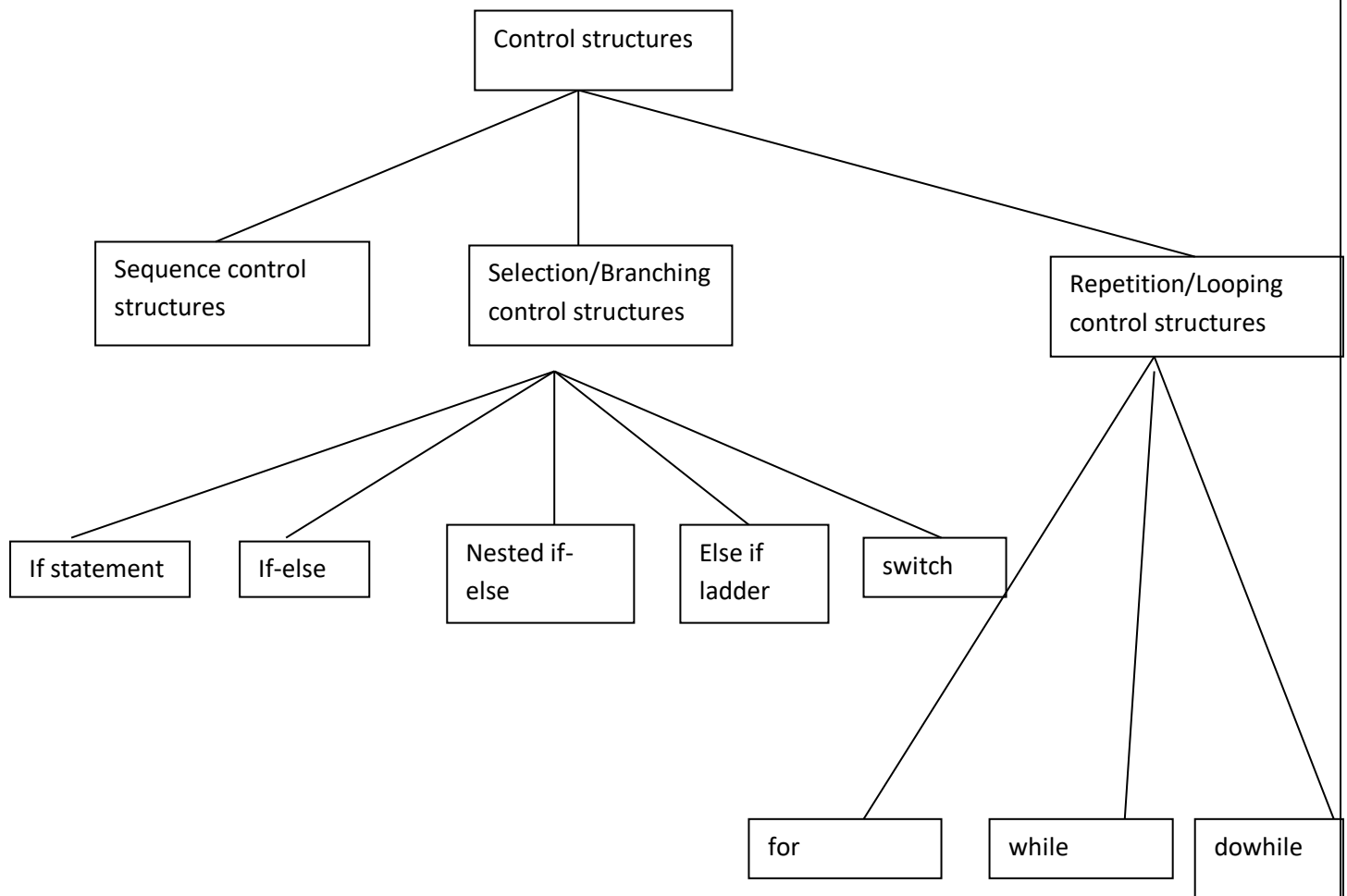
1

	Operator	Associativity	Precedence
() [] . ->	Function call Array subscript Dot (Member of structure) Arrow (Member of structure)	Left-to-Right	Highest 14
! - - ++ -- & * (type) sizeof	Logical NOT One's-complement Unary minus (Negation) Increment Decrement Address-of Indirection Cast Sizeof	Right-to-Left	13
* / %	Multiplication Division Modulus (Remainder)	Left-to-Right	12
+ -	Addition Subtraction	Left-to-Right	11
<< >>	Left-shift Right-shift	Left-to-Right	10
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left-to-Right	8
== !=	Equal to Not equal to	Left-to-Right	8
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

→Control structures:

All programs are written in terms of the following control structures, namely

1. Sequence control structures
2. Selection/Branching control structures.
3. Repetition/Looping control structures

**Sequence Control structures:**

Control structure is essentially built into 'c' Unless directed, otherwise the computer automatically executes. 'c' Statements one after the other in the order in which they are written

Selection control structures/Branching statements/Decision Making statement:

C language possesses decision making capabilities (selection control statements) by supporting the following statements

1. if statement
2. if/else statement
3. switch statement
4. goto statement

→If- statement: The 'if' statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

if(test expression)

{

Statement-block;

}

Statement-x;

→if-else statement: The 'if..else' statement is an extension of the simple 'if' statement

if(test-expression)

{

True-block statements

}

else

{

False-block statements


```
}
```

Statement-x;

If the test expression is true, then the true-block statements, immediately following the if- statement are executed; otherwise, the false-block statements are executed. In either case, either true block or false-block will be executed, not both

Else-if Ladder: There is another way of putting 'ifs' together multipath decisions are involved.

A multipath decision is a chain of 'ifs' in which the statement associated with each 'else' is an 'if'. It takes the following general form:

if(condition 1)

statement-1;

else if(condition 2)

statement-2;

else if(condition-3)

statement-3;

else if(condition n)

statement-n;

else

default-statement;

statement-x;

This construct is known as the 'else-if' ladder. The conditions are evaluated from the top, downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x. when all the n-conditions become false, then the final else containing the default statement will be executed.

Switch statement: C has a built-in multiway decision statement known as a 'switch'. The 'switch' statement tests the value of a given variable(or expression) against a list of 'case' values and when a match is found, a block of statements associated with that 'case' is executed. The general form of the 'switch' statement is as shown below:

```
switch(expression)
{
    case value-1:
        block-1
        break;
    case value-2:
        block-2
        break;
    .....
    .....
    default:
        default-block
        break;
}
statement-x;
```

Rules for 'switch' statement:

- The 'Switch' expression must be an integral type.
- Case labels must be constants or constant expression.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.
- The break statement transfers the control out of the 'switch' statement
- The break statement is optional.

- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
- The default may be placed anywhere but usually placed at the end.
- It is permitted to nest switch statements

The GOTO statement:

- C supports the 'goto' statement to branch unconditionally from one point to another in the program.
- The 'goto' requires a label in order to identify the place where the branch is to be made. A 'label' is any valid variable name, and must be followed by a colon.
- The label: can be anywhere in the program either before or after the 'goto' label; statement.

SYNTAX

goto label;

- A 'goto' breaks the normal sequential execution of a program. If the label: is before the statement 'goto' label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a **backward jump**.
- On the other hand, if the label: is placed after the goto label; some statement will be skipped and the jump is known as **forward jump**

Forward jump

goto label;

.....

.....

label:

Statement;

Backward Jump

label:

Statement;

.....

.....

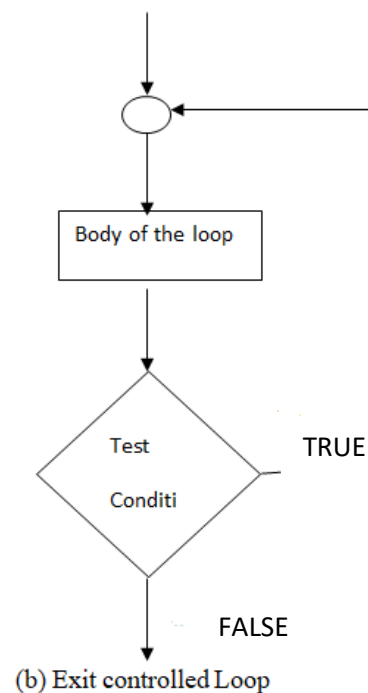
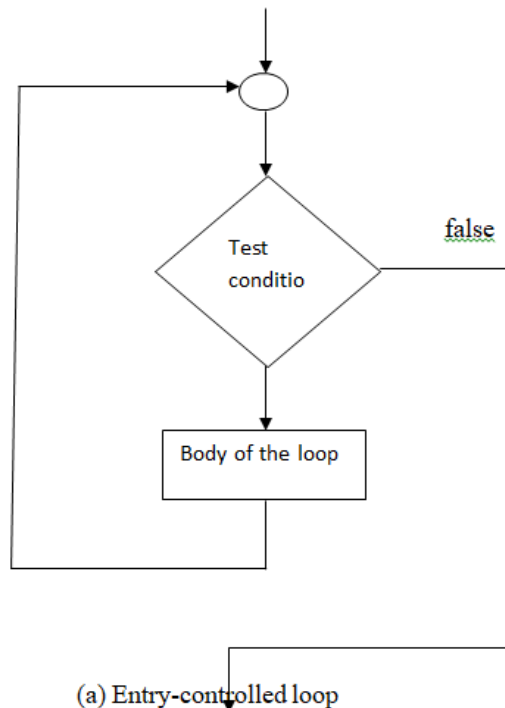
goto label;

→ Repetition control structure/Looping Statements:

In Looping, a sequence of Statements are executed until some conditions for termination of the loop are satisfied. Depending on the position of the control Statement in the loop, a control structure may be classified either as the

1. Entry-controlled loop

2. Exit controlled Loop



In the 'entry-controlled loop', the control conditions are tested before the start of the loop executions. If the conditions are not satisfied, then the body of the loop will not be executed. It is also known as 'pre-test loop'

- In the 'exit –controlled loop', the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. It is also known as 'post-test loop'.

A looping process, in general would include the following four steps:

1.Initialization of a condition variable

2.Test for a specified value of the condition variable for execution of the loop

3.Incrementing or updating the condition variable.

The C-language provides three constructs for performing loop operations. They are

- 1.The while statement
- 2.The do statement
- 3.The for statement

The 'while' statement: The simplest of all the looping structures in c is the while statement. The basic format of the while statement is

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

The while is an entry-controlled loop. The test-condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again.

The 'do-while' statements: In some cases, it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do.. while statements .This takes the form:

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

On reaching the 'do' statements, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statements is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement

Since the test-condition is evaluated at the bottom of the loop, the do...while construct provides an exit –controlled loop and therefore the body of the loop is always executed at least once

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

for(initialization;test-condition;increment)

{

body of the loop

}

In the ‘for’ loop all the three actions, namely initialization, testing and incrementing are placed in the ‘for’ statements itself, thus making them visible to the programmers and users, in one place.

Nesting of for loops: Nesting of loops, that is, one ‘for’ statement within another ‘for’ statement, is allowed in C

The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each ‘for’ statement.

ANSI C allows up to 15 levels of nesting.

→Arrays

An array is a group of related data items that share a common name and common type.

eg: An array name is ‘salary’ used to represent a set of salaries of a group of employees.

- A particular value is indicated by writing a number ‘index’ or ‘subscript’ in brackets after the array name.

Eg:salary[10]

Advantage: The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables to develop concise and efficient programs.

One-dimensional Arrays:

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

Declaration of Arrays:

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

SYNTAX:

datatype variable-name[size];

type —specifies the type of element that will be contained in the array, such as int, float or char

size—indicates the maximum number of elements that can be stored inside the array

eg: float student[50];

declares the 'student' to be an array containing 50 real elements. Any subscript 0 to 49 are valid

Initialization of Arrays:

The elements of arrays are initialized in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

SYNTAX:

datatype array-name[size]={list of values};

the values in the list are separated by commas.

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

Eg: float total[5]={0.0,15.75,-10};

Will initialize the first three elements to above values and the remaining two elements to zero.

The 'size' may even be omitted. In such cases, the compiler allocates enough space for all initialized elements.

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

For example:

int mark[5] = {19,10,8,17,9};

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Two- Dimensional Arrays:

Two-Dimensional Arrays are declared as follows:

SYNTAX:

datatype array-name[row-size][column-size]

Two-Dimensional Arrays can be used to store tables of values or representing matrices.

Initializing Two-Dimensional Arrays:

Like the one-Dimensional Arrays, two-Dimensional Arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

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

Initializes the elements of the first row to zero and the second row to one.

Initialization is done row by row

The above statement is equivalently written as

Eg: `int table[2][3]={ {0,0,0},{1,1,1} };`

Example: `int x[3][3];`

	Col 0	Col 1	Col 2
row 0	x[0][0]	x[0][1]	x[0][2]
row 1	x[1][0]	x[1][1]	x[1][2]
row 2	x[2][0]	x[2][1]	x[2][2]

Example: `int x[3][3]={1,50,2,75,8,4,9,33,77};`

	Col 0	Col 1	Col 2
row 0	1	50	2
row 1	75	8	4
row 2	9	33	77

Multi-dimensional arrays:

‘C ’ allows arrays of three or more dimensions, the exact limit is determined by the compiler.

General form is :

SYNTAX:

data type array-name [s1] [s2] [s3].....[sm];

Where Sm---- size of mth dimension.

Egs : `int survey [3] [5] [12];`

`float table [5] [4] [5] [3] ;`

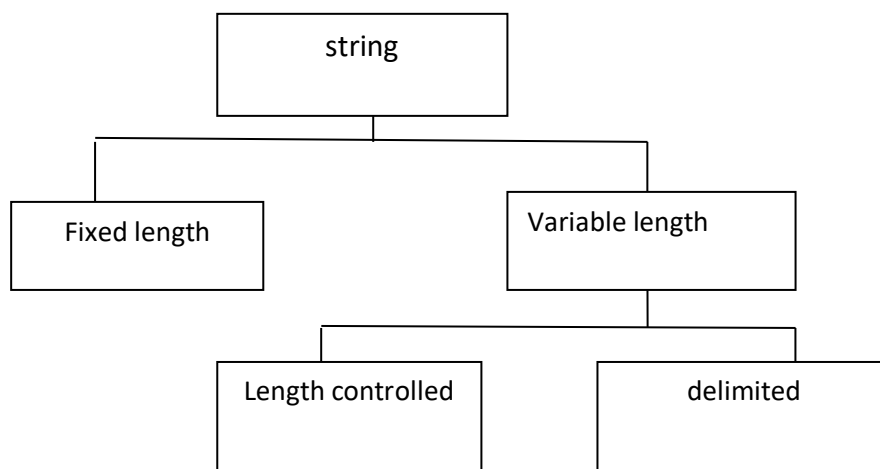
‘survey’ is a three- dimensional array declared to contain 180 integer type elements. Similarly ‘table’ is a four- dimensional array containing 300 elements of floating- point type.

➔Strings

String concepts:

In general, a String is a series of character treated as a unit.

The String taxonomy is shown in the figure below:



1. Fixed length String: when implementing a fixed length String format, the first decision is the size of the variable. If it is too small, all data can't be stored. If it is too big, memory gets wasted. A problem associated with storing data in fixed length String is how to differ the data from non-data. A common solution is to add spaces, at the end of the data.
2. Variable length String: It is a structure that can expand and contract to accommodate the data. This flexibility does not come without a cost. There must be some way to tell the end of the data.

C Strings /ARRAY OF CHARACTERS:

A String is a sequence of characters that is treated as a single data item. It is an array of characters. Any group of character(except double quote sign) defined between double quotation marks is a constant String

Eg: “sweet is the fruit of labour!”

Declaring and initializing string variables:

A String variable is any valid 'C' variable name and is always declared as an array. The general form of declaration of a String variable is

SYNTAX:

```
char stringname[size];
```

when the compiler assigns a character String to a character array, it automatically supplies a null character ('\0') at the end of the v therefore, the size should be equal to the maximum number of characters in the String plus one.

Characters arrays may be initialized when they are declared. 'C' permits a character array to be initialized in either of the following two forms:

```
char city[10]="HYDERABAD";
```

```
char city[10]={'H','Y','D','E','R','A','B','A','D','\0'};
```

The reason 'city' had to be 10 elements long is that the String HYDERABAD contains 9 characters and one element space is provided for the null terminator

'C' also permits to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized.

```
Eg; char string[]={'G','R','E','A','T','\0'};
```

Defines the array String as a five element array.

String Manipulations Functions :

There are several string library functions used to manipulate string and the prototypes for these functions are in header file "string.h". Some commonly used string handling functions are:

strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

strcat() : it joins two strings together. It takes following form

strcat(string1,string2);

This function is used to append a copy of a string at the end of the other string.

If the first string is "very" and second string is "good" then after using this function the string becomes "verygood".

The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected. A pointer to the first string str1 is returned by the function.

strcmp() This function is used to compare two strings.

If the two string match, strcmp() return a value 0 otherwise it return a non-zero value.

It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

It takes following form

strcmp(string1,string2);

strlen() This function return the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string.

It takes following form

n= strlen("string");

example

- l= strlen("good");

It return the value 4 .

User defined string

These functions are not predefined in the Compiler.

These functions are created by users as per their own requirements.

User-defined functions are not stored in library files.

There is no such kind of requirement to add a particular library.

Execution of the program begins from the user-define function.

Example: sum(), fact(),...etc.

reverse a string without using library function

read a string from the user and convert the string into lowercase and uppercase without using library function.

read and compare two strings using case and ignoring case without using library function.

Function in C: Function Declaration, Function Definition, Function call, Types of Function(User Defined, Library) , Passing Parameters to function (Call By Value, Call By Reference). Recursion function: Recursion definition, Iteration vs Recursion, Example programs: GCD, Factorial, sum of digits and Fibonacci.

➔ Functions

A function is a block of code that performs a specific task.

Dividing complex problem into small components makes program easy to understand and use.

Types of functions

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

- [Standard library functions](#)
- [User defined functions](#)

Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "`stdio.h`" header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

Note : see the table given in unit one

User-defined functions

As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

How user-defined function works

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

```
void functionName()
```

```
{
```

```
... ..
```

```
... ..
```

```
}
```

```
void main()
```

```
{
```

```
... ..
```

```
.. .. .
```

```
functionName();
```

```
.. .. .
```

```
.. .. .
```

```
}
```

The execution of a C program begins from the `main()` function.

When the compiler encounters `functionName();` inside the main function, control of the program jumps to

```
void functionName()
```

And, the compiler starts executing the codes inside the user-defined function.

The control of the program jumps to statement next to `functionName();` once all the codes inside the function definition are executed.

Remember, function name is an identifier and should be unique.

Need for user defined functions (or) Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

Elements of user defined functions

- 1.function definition
2. function call
3. function declaration or prototype

Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

It is divided into two parts namely **function head, function body**.

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Note: We should not use semicolon ; in function definition() head same like main() function.

The general syntax of a function definition is as follows:

```
return_type function_name(parameter list) /*.....function head.....*/
```

```
{
    local_variables declaration; /*.....function body.....*/
    executable statement(s);
return_value;
}
```

Example:

```
int add( int a, int b )
{
    int sum;
    sum = a + b;
```

```
return sum; }
```

Syntax of return statement

```
return (expression);
```

For example,

```
return a;
```

```
return (a+b);
```

The type of value returned from the function and the return type specified in function prototype and function definition must match.

Function Call

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, function call is made using `addNumbers(n1,n2);` statement inside the `main()`.

Function Declaration :

- Like variables function must be declared before they are used.
- A function prototype gives information to the compiler that the function may later be used in the program.

It consists of 4 parts.

- **Function type**
- **Function name**
- **Parameter list**
- **Terminating semi colon**

SYNTAX:

returntype functionname(type1 argument1, type2 argument2,...);

Example:

int mul(int a,int b);

int mul(int,int);

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables `n1` and `n2` are passed during function call. These are called actual parameters.

The parameters `a` and `b` accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

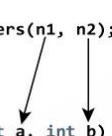
How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```



The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If `n1` is of char type, `a` also should be of char type. If `n2` is of float type, variable `b` also should be of float type.

A function can also be called without passing an argument.

Category of functions:

A function depending on whether the arguments are present or not and whether a value is returned or not, may belong to one of following categories

1. Function with no argument and no Return value
2. Function with no argument and with Return value
3. Function with argument and No Return value
4. Function with argument and Return value

1. Function with No argument and No Return value

In this category, the function has no arguments. It does not receive any data from the calling function. Similarly, it doesn't return any value. The calling function doesn't receive any data from the called function. So, there is no communication between calling and called functions.

In this program, We are going to calculate the Sum of 2 integer values and print the output from the user defined function itself.

➔ Parameter passing in functions:

There are different ways in which parameter data can be passed into and out of methods and functions.

First some important terminology:

Formal Parameter

A variable and its type as they appear in the prototype of the function or method.

Actual Parameter

The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Call by value or Pass by Value

Using "pass by value", the data associated with the actual parameter is copied into a separate storage location assigned to the formal parameter. Any modifications to the formal parameter variable inside the called function or method affect only this separate storage location and will therefore *not* be reflected in the actual parameter in the calling environment.

Example:

```
#include<stdio.h>

/* function declaration */
void swap(int x,int y);
void main ()
{
/* local variable definition */
int a =100, b =200;
printf("Before swap, value of a : %d\n", a );
```

```

printf("Before swap, value of b : %d\n", b );

/* calling a function to swap the values */

swap(a, b);

printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );

}

/* function definition to swap the values */

void swap(int x,int y)
{
int temp;
temp= x; /* save the value of x */
    x = y; /* put y into x */
    y = temp; /* put temp into y */
return;
}

```

Output:

```

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200

```

It shows that there are no changes in the values, though they had been changed inside the function.

Call by Reference or Pass by Reference

Using "pass by reference", the formal parameter receives a reference (or pointer) to the actual data in the calling environment, hence any changes to the formal parameter *are* reflected in the actual parameter in the calling environment.

Example:

```
#include<stdio.h>

/* function declaration */

void swap(int*x,int*y);

void main ()
{
/* local variable definition */

int a =100;
int b =200;

printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );

/* calling a function to swap the values.

    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b. */

swap(&a,&b);

printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
}

/* function definition to swap the values */

void swap(int*x,int*y)
{
int temp;

temp=*x; /* save the value at address x */

*x =*y; /* put y into x */

*y = temp; /* put temp into y */

return;
```

```
}
```

Output:

```
Before swap, value of a :100
```

```
Before swap, value of b :200
```

```
After swap, value of a :200
```

```
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

RECURSION

Recursion is the process of a function calling itself repeatedly till the given condition is satisfied. A function that calls itself directly or indirectly is called a recursive function and such kind of function calls are called recursive calls.

In C, recursion is used to solve complex problems by breaking them down into simpler sub-problems. We can solve large numbers of problems using recursion in C. For example, factorial of a number, generating Fibonacci series, generating subsets, etc.

The basic syntax structure of the recursive functions is:

```
type function_name (args) {  
    //function statements  
  
    // base condition  
  
    // recursion case (recursive call)  
}
```

Fundamentals of C Recursion

The fundamental of recursion consists of two objects which are essential for any recursive function. These are:

1. Recursion Case

2. Base Condition

```
int nSum(int n)
{
    if (n==0) {
        return 0;
    }
    int res = n + nsum(n-1);
    return res;
}
```

Base condition

Recursive case

Base Condition and Recursion Case for nSum() Function

1. Recursion Case

The recursion case refers to the recursive call present in the recursive function. It decides what type of recursion will occur and how the problem will be divided into smaller subproblems.

The **recursion case defined in the nSum()** function of the above example is:

```
int res = n + nSum(n - 1);
```

The recursion case is often represented mathematically as a recurrence relation. For the above case:

$$f(N) = N + f(N - 1);$$

This recurrence relation is used for the complexity analysis of the method.

2. Base Condition

The base condition specifies when the recursion is going to terminate. It is the condition that determines the exit point of the recursion.

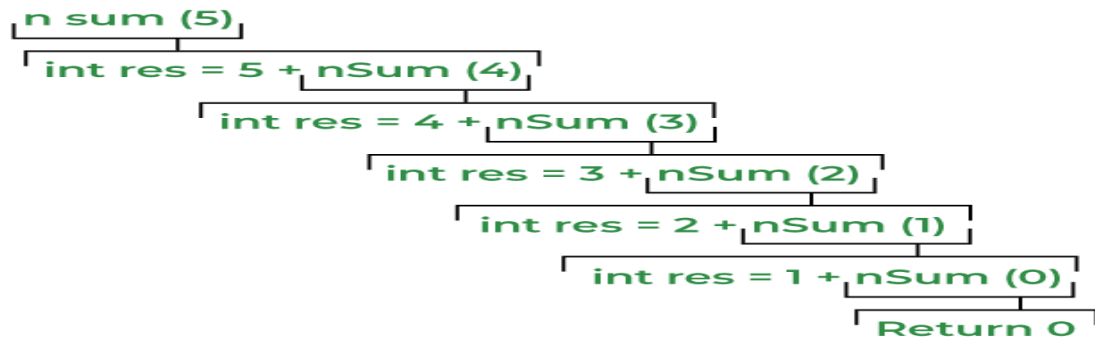
***Note:** It is important to define the base condition before the recursive case otherwise, the base condition may never be encountered and recursion might continue till infinity.*

Considering the above example again, the **base condition defined for the nSum()** function:

```
if (n == 0) {
    return 0;
}
```

```
}
```

the sum of the first 5 natural numbers.



What is Iteration?

When there is a repetition or loop, it is known as iterative. In Iteration, we prefer loops to perform the group of instructions repetitively until the state of the iteration becomes wrong.

Difference between Recursion and Iteration

S.No.	Recursion	Iteration
1	Recursion is the method of calling a function itself.	There is a repeated implementation of the bunch of instructions.
2	In case of recursion, we need to define the termination condition within the recursive function.	Here, we need to define the termination condition and the definition of the loop.
3	The code size is smaller than the iteration's code size.	The code size is large as compared to recursion.
4	The speed of recursion is slow.	It is fast as compared to recursion.

5	We mostly prefer recursion when there is no concern about time complexity and the size of code is small.	We prefer iteration when we have to manage the time complexity and the code size is large.
6	It has high time complexity.	The time complexity is lower as compared to recursion.
7	It requires more memory.	It requires less memory.

GCD of Two Numbers using Recursion

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

```
int hcf(int n1, int n2);
```

```
int main() {
```

```
    int n1, n2;
```

```
    printf("Enter two positive integers: ");
```

```
    scanf("%d %d", &n1, &n2);
```

```
    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
```

```
    return 0;
```

```
}
```

```
int hcf(int n1, int n2) {
```

```
    if (n2 != 0)
```

```
        return hcf(n2, n1 % n2);
```

```
    else
```

```

    return n1;
}

```

Output:

Enter two positive integers: 366

60

G.C.D of 366 and 60 is 6.

Factorial of a Number Using Recursion

The factorial of a positive number n is given by:

factorial of n ($n!$) = $1 * 2 * 3 * 4 * \dots * n$

The factorial of a negative number doesn't exist. And the factorial of 0 is 1.

```

#include<stdio.h>

```

```

long int multiplyNumbers(int n);

```

```

int main() {

```

```

    int n;

```

```

    printf("Enter a positive integer: ");

```

```

    scanf("%d",&n);

```

```

    printf("Factorial of %d = %ld", n, multiplyNumbers(n));

```

```

    return 0;

```

```

}

```

```

long int multiplyNumbers(int n) {

```

```

    if (n>=1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}

```

Output

Enter a positive integer: 6

Factorial of 6 = 720

Suppose the user entered 6.

Initially, multiplyNumbers() is called from main() with 6 passed as an argument.

Then, 5 is passed to multiplyNumbers() from the same function (recursive call). In each recursive call, the value of argument n is decreased by 1.

When the value of n is less than 1, there is no recursive call and the factorial is returned ultimately to the main() function.

```
/* C program to calculate sum of digits using recursion */
```

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

```
/* Function declaration */
```

```
int sumOfDigits(int num);
```

```
int main()
```

```
{
```

```
    int num, sum;
```

```

printf("Enter any number to find sum of digits: ");
scanf("%d", &num);
sum = sumOfDigits(num);
printf("Sum of digits of %d = %d", num, sum);

return 0;
}

/* Recursive function to find sum of digits of a number*/
int sumOfDigits(int num)
{
    // Base condition
    if(num == 0)
        return 0;

    return ((num % 10) + sumOfDigits(num / 10));
}

```

Output:

Enter any number to find sum of digits: 1234

Sum of digits of 1234 = 10

The **Fibonacci series** is the sequence where each number is the sum of the previous two numbers of the sequence. The first two numbers are 0 and 1 which are used to generate the whole series.

Example

Input: $n = 5$

Output: 0 1 1 2 3

Explanation: The first 5 terms of the Fibonacci series are 0, 1, 1, 2, 3.

Input: $N = 7$

Output: 0 1 1 2 3 5 8

Explanation: The first 7 terms of the Fibonacci series are 0, 1, 1, 2, 3, 5, 8.

// C Program to print the Fibonacci series using recursion

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

// Recursive function to print the fibonacci series

```
void fib(int n, int prev1, int prev2) {
```

// Base Case: when n gets less than 3

```
    if (n < 3) {
```

```
        return;
```

```
    }
```

```
    int curr = prev1 + prev2;
```

```
    prev2 = prev1;
```

```
    prev1 = curr;
```

```
    printf("%d ", curr);
```

```
    return fib(n - 1, prev1, prev2);
```

```
}
```

// Function that handles the first two terms and calls the

// recursive function

```
void printFib(int n) {
```

// When the number of terms is less than 1

```
    if (n < 1) {
```

```
        printf("Invalid number of terms\n");
```

```

    }

    // When the number of terms is 1
    else if (n == 1) {
        printf("%d ", 0);
    }

    // When the number of terms is 2
    else if (n == 2) {
        printf("%d %d", 0, 1);
    }

    // When number of terms greater than 2
    else {
        printf("%d %d ", 0, 1);
        fib(n, 0, 1);
    }

    return;
}

int main() {
    int n = 9;

    // Printing first 9 fibonacci series terms
    printFib(n);

    return 0;
}

```

Output

0 1 1 2 3 5 8 13 21

UNIT4

Structures: Definition and Accessing Structured data, Array of Structures, Passing structure to function, nested structure, Difference between structures & Unions.

Union: Introduction to Union and Operations on Unions.

Preprocessor Directives: Macros defines if elf.

Structures: Definition

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations.

C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the following syntax:

Syntax

```
struct structure_name {  
    data_type member_name1;  
    data_type member_name1;  
    ....  
    ....  
};
```

The above syntax is also called a structure template or structure prototype and no memory is allocated to the structure in the declaration.

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {  
    data_type member_name1;  
    data_type member_name1;  
    ....  
    ....  
}variable1, variable2, ...;
```

2. Structure Variable Declaration after Structure Template

```
// structure declared beforehand
struct structure_name variable1, variable2, .....;
```

Access Structure Members

We can access structure members by using the (.) dot operator.

Syntax

```
structure_name. member1;
structure_name. member2;
```

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

Initialize Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct Point
{
    int x = 0; // COMPILER ERROR: cannot initialize members here
    int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

```
struct Point
{
    int x;
    int y;
};
```

```
struct Point p = {0}; // Both x and y are initialized to 0
```

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
```

```
str.member2 = value2;
str.member3 = value3;
.
.
.
```

2. Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

3. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the [C99 standard](#).

```
struct structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };
```

The Designated Initialization is only supported in C but not in C++.

// C program to illustrate the use of structures

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

// declaring structure with name str1

```
struct str1 {
    int i;
    char c;
    float f;
    char s[30];
};
```

// declaring structure with name str2

```
struct str2 {
    int ii;
    char cc;
    float ff;
} var; // variable declaration with structure template
```

```

// Driver code

int main()
{
    // variable declaration after structure template
    // initialization with initializer list and designated
    // initializer list
    struct str1 var1 = { 1, 'A', 1.00, "isl" },
        var2;
    struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };

    // copying structure using assignment operator
    var2 = var1;

    printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var1.i, var1.c, var1.f, var1.s);
    printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var2.i, var2.c, var2.f, var2.s);
    printf("Struct 3:\n\ti = %d, c = %c, f = %f\n", var3.ii,
        var3.cc, var3.ff);

    return 0;
}

```

Output

Struct 1:

i = 1, c = A, f = 1.000000, s = isl

Struct 2:

i = 1, c = A, f = 1.000000, s = isl

Struct 3

i = 5, c = a, f = 5.000000

Array of Structures

An array whose elements are of type structure is called array of structure. It is generally useful when we need multiple structure variables in our program.

Need for Array of Structures

Suppose we have 50 employees and we need to store the data of 50 employees. So for that, we need to define 50 variables of struct Employee type and store the data within that. However, declaring and handling the 50 variables is not an easy task. Let's imagine a bigger scenario, like 1000 employees.

So, if we declare the variable this way, it's not possible to handle this.

```
struct Employee emp1, emp2, emp3, ... .. emp1000;
```

For that, we can define an array whose data type will be struct Employee so that will be easily manageable.

Declaration of Array of Structures

```
struct structure_name array_name [number_of_elements];
```

Initialization of Array of Structures

We can initialize the array of structures in the following ways:

```
struct structure_name array_name [number_of_elements] = {  
    {element1_value1, element1_value2, ....},  
    {element2_value1, element2_value2, ....},  
    .....  
    .....  
};
```

The same initialization can also be done as:

```
struct structure_name array_name [number_of_elements] = {  
    element1_value1, element1_value2 .....,  
    element2_value1, element2_value2 .....,  
};
```

```
//program for Student Information
```

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

```
struct student {  
    char firstName[50];  
    int roll;
```

```

float sub1marks,sub2marks,sub3marks,avg;

} s[];

int main() {
    int i;
    float totalmarks,avg,grade;
    printf("Enter information of students:\n");

    // storing information
    for (i = 0; i < 5; ++i) {
        s[i].roll = i + 1;
        printf("\nFor roll number%d,\n", s[i].roll);
        printf("Enter first name: ");
        scanf("%s", s[i].firstName);
        printf("Enter subject1 marks: ");
        scanf("%f", &s[i].sub1marks);
        printf("Enter subject2 marks: ");
        scanf("%f", &s[i].sub2marks);
        printf("Enter subject3 marks: ");
        scanf("%f", &s[i].sub3marks);

    }
    printf("Displaying Information:\n\n");

    // displaying information
    for (i = 0; i < 5; ++i) {
        printf("\nRoll number: %d\n", i + 1);
        printf("First name: ");
        puts(s[i].firstName);
    }
}

```

```

printf("subject1 Marks: %f\n", s[i].sub1marks);
printf("subject2 Marks: %f\n", s[i].sub2marks);
printf("subject3 Marks: %f\n", s[i].sub3marks);
printf("\n");
s[i].avg=s[i].sub1marks+s[i].sub2marks+s[i].sub3marks/3;
printf("average marks of student:%f\n",s[i].avg);
}
return 0;
}

```

C – Passing struct to function

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

Passing structure to function in C:

It can be done in below 3 ways.

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global

Example program – passing structure to function in C by value:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.


```

#include <stdio.h>
#include <string.h>

struct student
{
    int id;
    char name[20];
    float percentage;
};

void func(struct student record);

int main()
{
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    func(record);
    return 0
}

void func(struct student record)
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}

```

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

Example program – Passing structure to function in C by address:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

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

struct student
{
    int id;
    char name[20];
    float percentage;
};

void func(struct student *record);

int main()
{
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    func(&record);
```

```

        return 0;
    }

void func(struct student *record)
{
    printf(" Id is: %d \n", record->id);
    printf(" Name is: %s \n", record->name);
    printf(" Percentage is: %f \n", record->percentage);
}

```

Output:

```

Id is: 1
Name is: Raju
Percentage is: 86.500000

```

Example program to declare a structure variable as global in C:

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

C

```

#include <stdio.h>
#include <string.h>

struct student
{
    int id;
    char name[20];
    float percentage;
};

struct student record; // Global declaration of structure

void structure_demo();

```

```

int main()
{
    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    structure_demo();
    return 0;
}

void structure_demo()
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}

```

Output:

```

Id is: 1
Name is: Raju
Percentage is: 86.500000

```

A **nested structure** in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

Syntax:

```

struct name_1
{
    member1;
    member2;
    .
    .
    membern;

    struct name_2
    {

```

```

    member_1;
    member_2;
    .
    .
    member_n;
}, var1
} var2;

```

The member of a nested structure can be accessed using the following syntax:

Variable name of Outer_Structure.Variable name of Nested_Structure.data member to access

Example:

- Consider there are two structures **Employee (depended structure)** and another structure called **Organisation(Outer structure)**.
- The structure Organisation has the data members like organisation_name, organisation_number.
- The Employee structure is nested inside the structure Organisation and it has the data members like employee_id, name, salary.

For accessing the members of Organisation and Employee following syntax will be used:

```

org.emp.employee_id;
org.emp.name;
org.emp.salary;

org.organisation_name;
org.organisation_number;

```

Here, org is the structure variable of the outer structure Organisation and emp is the structure variable of the inner structure Employee.

Different ways of nesting structure

The structure can be nested in the following different ways:

1. **By separate nested structure**
2. **By embedded nested structure.**

1. By separate nested structure: In this method, the two structures are created, but the dependent structure(Employee) should be used inside the main structure(Organisation) as a member. Below is the C program to implement the approach:

- C

```

// C program to implement
// the above approach

```

```

#include <stdio.h>
#include <string.h>

// Declaration of the
// dependent structure
struct Employee
{
    int employee_id;
    char name[20];
    int salary;
};

// Declaration of the
// Outer structure
struct Organisation
{
    char organisation_name[20];
    char org_number[20];

    // Dependent structure is used
    // as a member inside the main
    // structure for implementing
    // nested structure
    struct Employee emp;
};

// Driver code
int main()
{

```

```

// Structure variable
struct Organisation org;

// Print the size of organisation
// structure
printf("The size of structure organisation : %ld\n",
        sizeof(org));

org.emp.employee_id = 101;
strcpy(org.emp.name, "Robert");
org.emp.salary = 400000;
strcpy(org.organisation_name,
        "GeeksforGeeks");
strcpy(org.org_number, "GFG123768");

// Printing the details
printf("Organisation Name : %s\n",
        org.organisation_name);
printf("Organisation Number : %s\n",
        org.org_number);
printf("Employee id : %d\n",
        org.emp.employee_id);
printf("Employee name : %s\n",
        org.emp.name);
printf("Employee Salary : %d\n",
        org.emp.salary);
}

```

Output:

The size of structure organisation : 68

Organisation Name : GeeksforGeeks

Organisation Number : GFG123768

Employee id : 101

Employee name : Robert

Employee Salary : 400000

2. By Embedded nested structure: Using this method, allows to declare structure inside a structure and it requires fewer lines of code.

Case 1: Error will occur if the structure is present but the structure variable is missing.

- C

```
// C program to implement
// the above approach
#include <stdio.h>

// Declaration of the outer
// structure
struct Organisation
{
    char organisation_name[20];
    char org_number[20];

    // Declaration of the employee
    // structure
    struct Employee
    {
        int employee_id;
        char name[20];
        int salary;

        // This line will cause error because
        // datatype struct Employee is present ,
```



```
// but Structure variable is missing.
```

```
};
```

```
};
```

```
// Driver code
```

```
int main()
```

```
{
```

```
// Structure variable of organisation
```

```
struct Organisation org;
```

```
printf("%ld", sizeof(org));
```

```
}
```

Differences between Structure and Union

Differences between Structure and Union are as shown below in tabular format as shown below as follows:

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

UNION

Introduction to Union:

The Union is a user-defined data type in C language that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

Syntax of Union in C

The syntax of the union in C can be divided into three steps which are as follows:

C Union Declaration

In this part, we only declare the template of the union, i.e., we only declare the members' names and data types along with the name of the union. No memory is allocated to the union in the declaration.

```
union union_name {  
    datatype member1;  
    datatype member2;  
    ...  
};
```

Different Ways to Define a Union Variable

We need to define a variable of the union type to start using union members. There are two methods using which we can define a union variable.

1. With Union Declaration
2. After Union Declaration

1. Defining Union Variable with Declaration

```
union union_name {  
    datatype member1;  
    datatype member2;  
    ...  
} var1, var2, ...;
```

2. Defining Union Variable after Declaration

```
union union_name var1, var2, var3...;
```

where *union_name* is the name of an already declared union.

Access Union Members

We can access the members of a union by using the [\(.\) dot operator](#) just like structures.

```
var1.member1;
```

where *var1* is the **union variable** and *member1* is the **member of the union**.

The above method of accessing the members of the union also works for the nested unions.

var1.member1.memberA;

Here,

- *var1* is a union variable.
- *member1* is a member of the union.
- *memberA* is a member of member1.

Initialization of Union in C

The initialization of a union is the initialization of its members by simply assigning the value to it.

var1.member1 = some_value;

One important thing to note here is that **only one member can contain some value at a given instance of time.**

// C Program to demonstrate how to use union

#include <stdio.h>

// union template or declaration

union un {

int member1;

char member2;

float member3;

};

// driver code

int main()

{

// defining a union variable

union un var1;

// initializing the union member

```

var1.member1 = 15;

printf("The value stored in member1 = %d",
      var1.member1);

return 0;
}

```

Output:

The value stored in member1 = 15

Size of Union

The size of the union will always be equal to the size of the largest member of the array. All the less-sized elements can store the data in the same space without any overflow.

Pre-processor Directives in C

Pre-processor is placed in the source program before the main line, it begins with the symbol "#" in column one and does not require a semicolon at the end.

The commonly used pre-processor directives are –

- #define
- #undef
- #include
- #ifdef
- #endif
- #if
- #else

The pre-processor directives are divided into three categories –

- Macro substitution directives.
- File inclusion directives.
- Compiler control directives.



Macro Substitution Directives

This helps us to define macros that act as placeholders or shortcuts for code or constants and expressions. The pre-processor scans the source code and replaces each appearance of macros with its corresponding text or value before compilation begins.

Syntax

Given below is the syntax for the macro substitution directive –

```
#define identifier string
```

Example

Following is an example of a simple macro –

```
// Following is a sample macro
```

```
#define MAX 500
```

```
// Following is a macro with arguments
```

```
#define sqrt(x) x*x
```

```
// Following is a nested macro
```

```
#define A 10
```

```
#define B A+1
```

With macro arguments, we can perform operations like calculations on input data directly, which makes the code more efficient.

File Inclusion Directives

File Inclusion is used for adding content of one file to another when its pre-processing. This helps us when we need to include external libraries or our own header files.

Syntax

Given below is the syntax for the file inclusion directive –

```
#include "filename"
```

or,

```
#include <filename>
```

Example

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

```
#include "FORM.C"
```

File inclusion directive helps us to include files that contain declarations and function definitions. The angle brackets include standard library files, and double quotes are used for user-defined files. This feature allows code to be reused across different programs.

Compiler Control Directives

These are used to control the compiler actions. C pre-processor offers a feature called conditional compilation, which can be used to switch on or off based on a particular line or group of lines in a program.

For example, **#ifdef** and **#ifndef** can be used to include certain parts of the code only if a condition is true. This helps when dealing with platform-specific code or debugging parts of a program without affecting the entire code-base.

In more complex programs, conditional compilation gives control over the sections of code that are compiled and ensures that certain parts only run in specific environments or configurations.

Example

Let's say we have a code that runs differently on Windows and Linux:

```
#if def _WIN32
    printf("This is Windows.");
#else
    printf("This is Linux.");
#endif
```

In this code, if the program is compiled on Windows, it will print "This is Windows." If compiled on Linux or any other system, it will print "This is Linux."

UNIT-5

Pointers: Define Pointers, Pointer Arithmetic, Array of pointers, pointers to Array

Null pointer, generic pointer, double pointers, passing pointer to function: call by address, Accessing structure using pointers, Self-referential structure and dynamic memory allocation, operation on pointers

File Handling: I/O streams, file operations, file modes

Sequential/random accessing files, command lines arguments

Pointers: A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the (*) dereferencing operator in the pointer declaration.

datatype * *ptr*;

where

- *ptr* is the name of the pointer.
- datatype is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. Pointer Declaration
2. Pointer Initialization
3. Pointer Dereferencing

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the (*) dereference operator before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the (&: ampersand) addressof operator to get the memory address of a variable and then store it in the pointer variable.

Example

```
int var = 10;  
int * ptr;  
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called pointer definition as the pointer is declared and initialized at the same time.

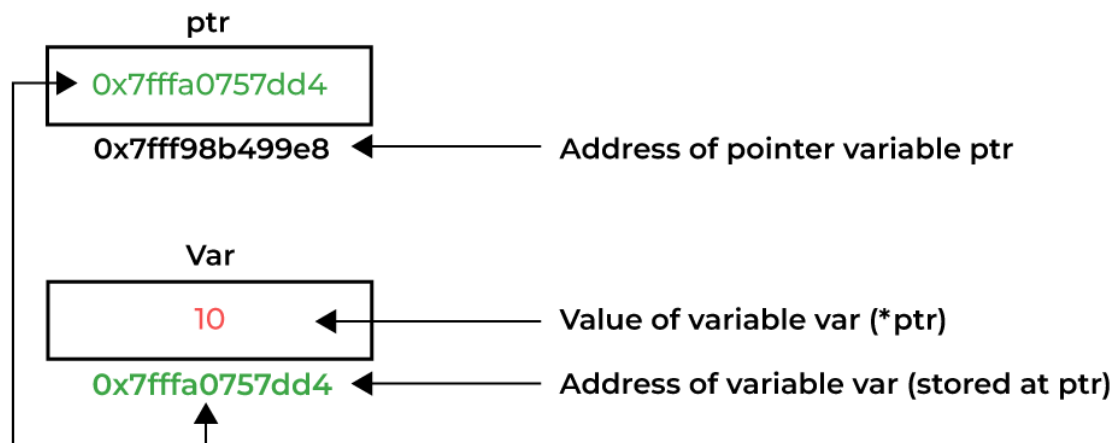
Example

```
int *ptr = &var;
```

Note: It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.

3. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) dereferencing operator that we used in the pointer declaration.



Dereferencing a Pointer in C

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The [pointer](#) variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

1. Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores address 1000 is incremented, then it will increment by 2(size of an int), and the new address will point to 1002. While if a float type pointer is incremented then it will increment by 4(size of a float) and the new address will be 1004.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores address 1000 is decremented, then it will decrement by 2(size of an int), and the new address will point to 998. While if a float type pointer is decremented then it will decrement by 4(size of a float) and the new address will be 996.

2. Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

For Example:

Consider the same example as above where the ptr is an integer pointer that stores 1000 as an address. If we add integer 5 to it using the expression, $ptr = ptr + 5$, then, the final address stored in the ptr will be $ptr = 1000 + \text{sizeof}(\text{int}) * 5 = 1020$.

Array of Pointers in C

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Note: It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing *array_name in the parenthesis will mean that array_name is a pointer to an array.

Pointer to Arrays

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as [Pointer to Arrays](#). We can create a pointer to an array using the given syntax.

Syntax

```
char *ptr = &array_name;
```

we have a pointer *ptr* that points to the 0th element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array.

Array of Pointers in C

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

// C program to demonstrate the use of array of pointers

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

```
int main()
{
    // declaring some temp variables
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;

    // array of pointers to integers
    int* ptr_arr[3] = { &var1, &var2, &var3 };

    // traversing using loop
    for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }

    return 0;
}
```

NULL Pointer

The [Null Pointers](#) are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

```
data_type *pointer_name = NULL;
```

or

```
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

Generic Pointer

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.

void pointers cannot be dereferenced.

The [C standard](#) doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of the void as 1.

Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or [pointers-to-pointer](#). Instead of pointing to a data value, they point to another pointer.

Syntax

```
datatype ** pointer_name;
```

Accessing structure using pointers

A structure pointer is defined as the [pointer](#) which points to the address of the memory block that stores a [structure](#) known as the structure pointer. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

Accessing the Structure Member with the Help of Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. With the help of (*) asterisk or indirection operator and (.) dot operator.
2. With the help of (->) Arrow operator.

// C Program to demonstrate Structure pointer

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

```
#include <string.h>
```

```
// Creating Structure Student
```

```
struct Student {  
    int roll_no;  
    char name[30];  
    char branch[40];  
    int batch;  
};
```

```
// variable of structure with pointer defined  
struct Student s, *ptr;
```

```

int main()
{
    ptr = &s;
    // Taking inputs
    printf("Enter the Roll Number of Student\n");
    scanf("%d", &ptr->roll_no);
    printf("Enter Name of Student\n");
    scanf("%s", &ptr->name);
    printf("Enter Branch of Student\n");
    scanf("%s", &ptr->branch);
    printf("Enter batch of Student\n");
    scanf("%d", &ptr->batch);

    // Displaying details of the student
    printf("\nStudent details are: \n");

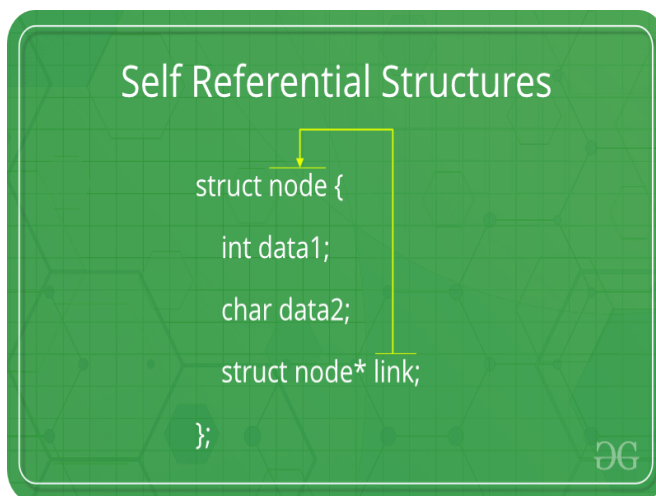
    printf("Roll No: %d\n", ptr->roll_no);
    printf("Name: %s\n", ptr->name);
    printf("Branch: %s\n", ptr->branch);
    printf("Batch: %d\n", ptr->batch);

    return 0;
}

```

Self Referential Structures

Self Referential structures are those [structures](#) that have one or more pointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self-referential in nature.

C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under `<stdlib.h>` header file to facilitate dynamic memory allocation in C programming. They are:

1. `malloc()`
2. `calloc()`
3. `free()`
4. `realloc()`

C `malloc()` method

The “**malloc**” or “**memory allocation**” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

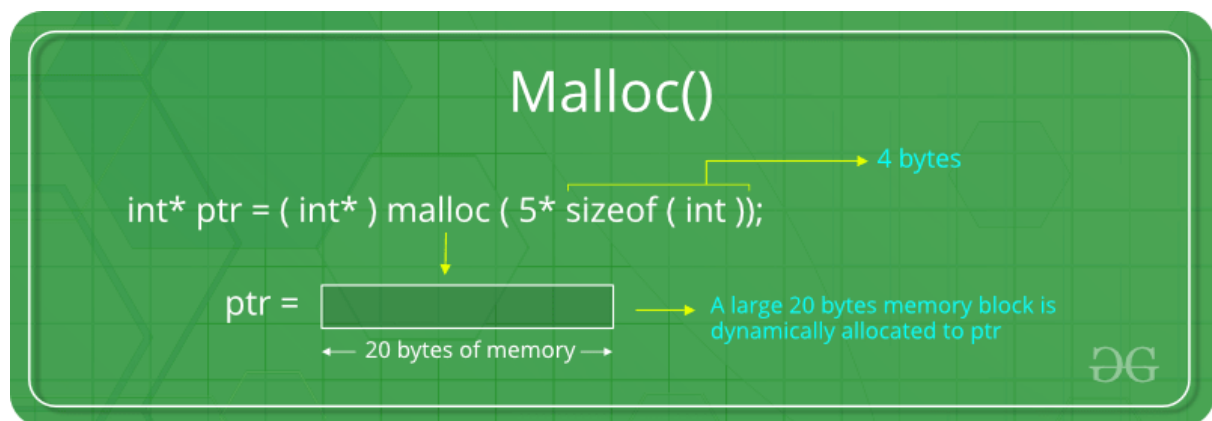
Syntax of `malloc()` in C

`ptr = (cast-type*) malloc(byte-size)`

For Example:

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

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



If space is insufficient, allocation fails and returns a NULL pointer.

C `calloc()` method

1. “**calloc**” or “**contiguous allocation**” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to `malloc()` but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to `malloc()`.

Syntax of `calloc()` in C

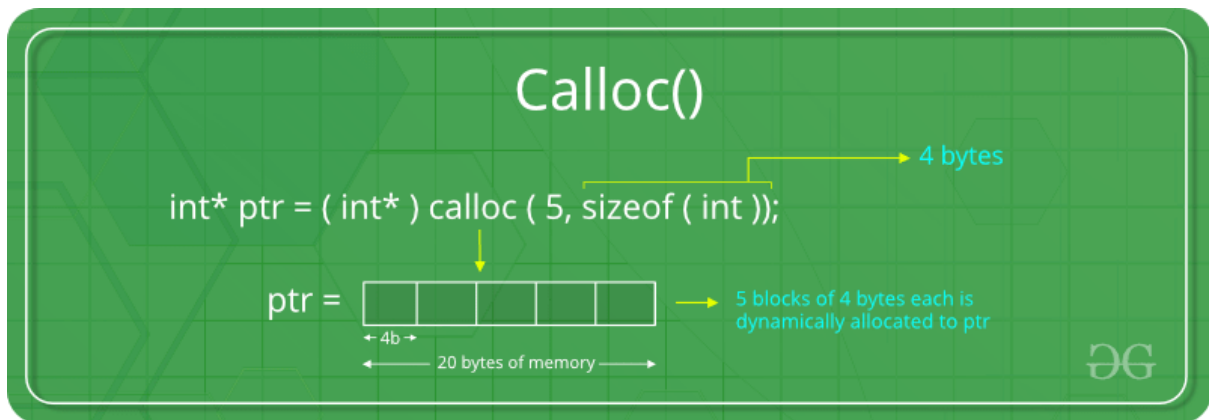
`ptr = (cast-type*)calloc(n, element-size);`

here, n is the no. of elements and element-size is the size of each element.

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



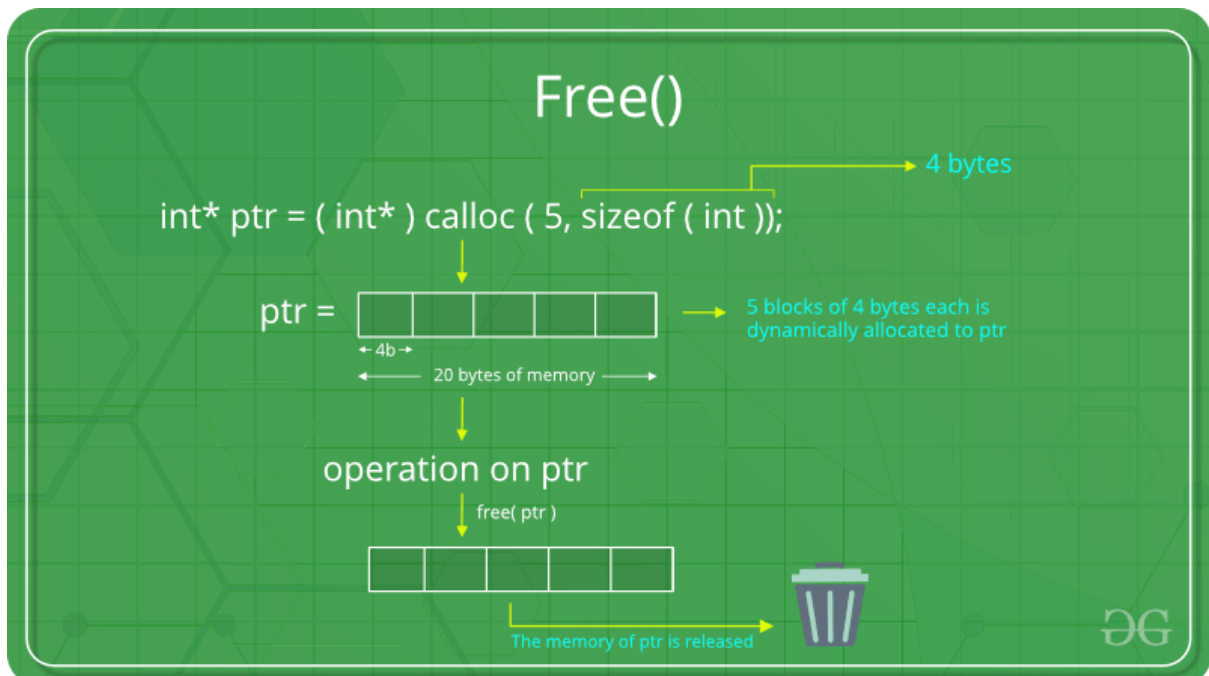
If space is insufficient, allocation fails and returns a NULL pointer.

C free() method

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions `malloc()` and `calloc()` is not de-allocated on their own. Hence the `free()` method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```



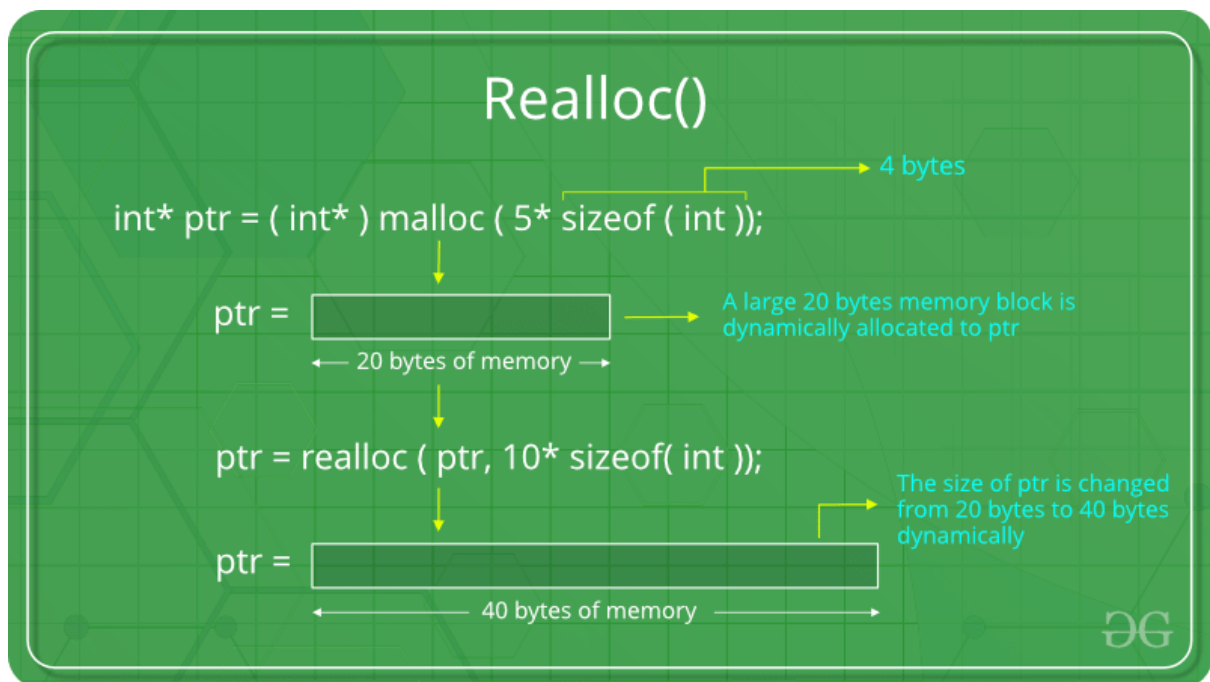
C realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

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

where ptr is reallocated with new size 'newSize'.



If space is insufficient, allocation fails and returns a NULL pointer.

File Handling

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as `fopen()`, `fwrite()`, `fread()`, `fseek()`, `fprintf()`, etc. to perform input, output, and many different C file operations in our program.

Why do we need File Handling in C?

- **Reusability:** The data stored in the file can be accessed, updated, and deleted anywhere and anytime providing high reusability.
 - **Portability:** Without losing any data, files can be transferred to another in the computer system. The risk of flawed coding is minimized with this feature.
 - **Efficient:** A large amount of input may be required for some programs. File handling allows you to easily access a part of a file using few instructions which saves a lot of time and reduces the chance of errors.
 - **Storage Capacity:** Files allow you to store a large amount of data without having to worry about storing everything simultaneously in a program.
-

I/O Streams

runtime environment must provide three streams viz. standard input, standard output and standard error to every C program. These streams are `stdin`, `stdout` and `stderr` and these are pointers to `FILE` structure.

stdin is default to read from a device which is mostly a keyboard,

stdout writes default to a output device which is mostly a terminal or user screen.

stderr writes error messages default to terminal or user screen. **perror()** function writes its error messages to a place where `stderr` writes its own.

C File Operations

C file operations refer to the different possible operations that we can perform on a file in C such as:

1. Creating a new file – **fopen()** with attributes as “a” or “a+” or “w” or “w+”
2. Opening an existing file – **fopen()**
3. Reading from file – **fscanf()** or **fgets()**
4. Writing to a file – **fprintf()** or **fputs()**
5. Moving to a specific location in a file – **fseek()**, **rewind()**
6. Closing a file – **fclose()**

Create a File in C

The `fopen()` function can not only open a file but also can create a file if it does not exist already. For that, we have to use the modes that allow the creation of a file if not found such as `w`, `w+`, `wb`, `wb+`, `a`, `a+`, `ab`, and `ab+`.

```
FILE *fptr;  
fptr = fopen("filename.txt", "w");
```

Reading From a File

The file read operation in C can be performed using functions `fscanf()` or `fgets()`. Both the functions performed the same operations as that of `scanf` and `gets` but with an additional parameter, the file pointer. There are also other functions we can use to read from a file. Such functions are listed below:

Function	Description
<u>fscanf()</u>	Use formatted string and variable arguments list to take input from a file.
<u>fgets()</u>	Input the whole line from the file.
<u>fgetc()</u>	Reads a single character from the file.
<u>fgetw()</u>	Reads a number from a file.
<u>fread()</u>	Reads the specified bytes of data from a binary file.

Write to a File

The file write operations can be performed by the functions `fprintf()` and `fputs()` with similarities to read operations. C programming also provides some other functions that can be used to write data to a file such as:

Function	Description
<u>fprintf()</u>	Similar to <code>printf()</code> , this function use formatted string and variable arguments list to print output to the file.
<u>fputs()</u>	Prints the whole line in the file and a newline at the end.
<u>fputc()</u>	Prints a single character into the file.
<u>fputw()</u>	Prints a number to the file.
<u>fwrite()</u>	This functions write the specified amount of bytes to the binary file.

Closing a File

The `fclose()` function is used to close the file. After successful file operations, you must always close a file to remove it from the memory.

Syntax of `fclose()`

```
fclose(file_pointer);
```

where the *file_pointer* is the pointer to the opened file.

Sequential/random accessing files

The following methods exist in C for accessing data that is saved in a file:

1. Sequential Access
2. Random Access

Sequential Access

Sequential access is not the optimum method for reading the data in the centre of the file if the file size is too large. A *sequential file* is a kind of file organization in which we can keep the data continuously, and every record is stored after the other and can be accessed the data sequentially. In this file, the data is *read* from the *start* of the file and processed in the sequence it comes in the file. It is mainly suitable for those applications where the data is processed sequentially and doesn't access randomly.

Random Access

A *random access file* in C is a kind of file that enables us to *write or read* any data in the disk file without having to read or write each section of data before it. In this file, we may instantly find for data, modify it, or even delete or remove it. We may open and close random access files in C in the same way we can sequential files, but we require a few more functions to do so

Functions of Random Access file

Consequently, there are mainly three functions that assist in accessing the random access file in C:

- *ftell()*
- *rewind()*
- *fseek()*

ftell() function:

The *file pointer's position* is relative to the file's beginning and can be determined using the *ftell()* function.

Syntax:

It has the following syntax:

```
ftell(FILE *fp)
```

rewind() function:

The *file pointer* can be moved to the *file's beginning* using the *rewind()* function. When a file needs to be updated, it is useful.

Syntax:

It has the following syntax:

```
rewind(FILE *fp);
```

fseek() function:

The *fseek()* function is used to move the *file position* to a given location.

Syntax:

The syntax is:

```
int fseek(FILE *fp, long displacement, int origin);
```

Constant

Value

Position

SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

Command Line Argument

Command-line arguments are the values given after the name of the program in the command-line shell of Operating Systems. Command-line arguments are handled by the main() function of a C program.

To pass command-line arguments, we typically define main() with two arguments: the first argument is the **number of command-line arguments** and the second is a **list of command-line arguments**.

Syntax

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

Here,

- **argc (ARGument Count)** is an integer variable that stores the number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, the value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non-negative.
- **argv (ARGument Vector)** is an array of character pointers listing all the arguments.
- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- argv[0] is the name of the program, After that till argv[argc-1] every element is command-line arguments.

Properties of Command Line Arguments in C

1. They are passed to the main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control programs from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[argc-1] points to the last argument.

Note: You pass all the command line arguments separated by a space, but if the argument itself has a space, then you can pass such arguments by putting them inside double quotes "" or single quotes ' '.

The below program demonstrates the working of command line arguments.

C

```
// C program to illustrate
// command line arguments
#include <stdio.h>
```

```
int main(int argc, char* argv[])
```

```

{
    printf("Program name is: %s", argv[0]);

    if (argc == 1)
        printf("\nNo Extra Command Line Argument Passed "
            "Other Than Program Name");

    if (argc >= 2) {
        printf("\nNumber Of Arguments Passed: %d", argc);
        printf("\n----Following Are The Command Line "
            "Arguments Passed----");
        for (int i = 0; i < argc; i++)
            printf("\nargv[%d]: %s", i, argv[i]);
    }
    return 0;
}

```

Output in different scenarios:

1. Without argument: When the above code is compiled and executed without passing any argument, it produces the following output.

Terminal Input

\$./a.out

Output

Program Name Is: ./a.out

No Extra Command Line Argument Passed Other Than Program Name

2. Three arguments: When the above code is compiled and executed with three arguments, it produces the following output.

Terminal Input

\$./a.out First Second Third

Output

Program Name Is: ./a.out

Number Of Arguments Passed: 4

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First

argv[2]: Second

argv[3]: Third

3. Single Argument: When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

Terminal Input

\$./a.out "First Second Third"

Output

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----

```
argv[0]: ./a.out  
argv[1]: First Second Third
```

4. A single argument in quotes separated by space: When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

Terminal Input

```
$ ./a.out 'First Second Third'
```

Output

```
Program Name Is: ./a.out
```

```
Number Of Arguments Passed: 2
```

```
----Following Are The Command Line Arguments Passed----
```

```
argv[0]: ./a.out
```

```
argv[1]: First Second Third
```