SYLLABUS

Class – B.Com. III Sem.

Subject – C Programming

UNIT – I	Concept of structural programming, algorithm, flowchart, advantages & disadvantages of algorithm & flowchart, making of sequence, selection & iteration, introduction to translator (compiler, assembler, interpreter) linker, loader
UNIT – II	Introduction to C language, history of C language, features of C Language, character sets, keywords, identifier, constant, concept of header file (stdio.h, conio.h math.h string.h), standard functions (print f(), scanf()). Data types in C: fundamental & derived data types, operations in C: airthematic, relational, logical, increment, decrement, bitwise, comound assignment operator, conditional operator.
UNIT – III	Flow of control: Selection statement, if, IF…. Else, nested IF Iteration statement: While loop, for, do-while loop.
UNIT – IV	Functions: Introduction, types of functions, local, global variables, call by value, call by reference, function prototype, recursion technique & example, pointer: concept of pointer, address operator, indirection operator, passing pointer as parameter, pointer air thematic, pointer to array, pointer to function.
UNIT – V	Concept of array: introduction, need of array, type of array (1d, 2d, array), memory representation of array, structure & union: Concept of structure, syntax, reading writing structure variable and array of structure, passing structure in function. Union: concept of union, different between structure & union. Examples of union.

UNIT — I

Computer programming:

Computer programs are written using one of the programming language. A program has a set of instructions written in correct order to get the desired result. The method of writing the instructions to solve the given problem is called programming.

Programming techniques:

There are two types of programming techniques commonly used:

1. Procedural or structural programming
2. Object oriented programming

Object-oriented programming :-

Object-oriented programming (**OOP**) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. Objective-C, Smalltalk, and Java are examples of object-oriented programming languages.

Structured programming :-

Structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops—in contrast to using simple tests and jumps such as the *goto* statement which could lead to "spaghetti code" which is both difficult to follow and to maintain.

At a low level, structured programs are often composed of simple, hierarchical program flow structures. These are sequence, selection, and repetition:

- "Sequence" refers to an ordered execution of statements.
- In "selection" one of a number of statements is executed depending on the state of the program. This is usually expressed with keywords such asif..then..else..endif, switch, or case. In some languages keywords cannot be written verbatim, but must be stropped.
- In "repetition" a statement is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as while, repeat, for or do..until. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).

Advantages of Structured programming:-

1. **Easy to write**
2. **Easy to debug**
3. **Easy to Understand**
4. **Easy to Change**

Algorithm:

- An algorithm is a finite sequence of step by step, discrete, unambiguous instructions for solving a particular problem
  - has input data, and is expected to produce output data
  - each instruction can be carried out in a finite amount of time in a deterministic way
- In simple terms, an algorithm is a series of instructions to solve a problem (complete a task)
- Problems can be in any form
  - Business
    - Get a part from Vancouver to Ottawa by morning
    - Allocate manpower to maximize profit
  - Life
    - I am hungry. How do I order pizza?

- Explain how to tie shoelaces to a five year old child

Algorithmic Representation of Computer Functions

- Input
  - Get information input
- Storage
  - Store information Given/Result
- Process
  - Arithmetic Let (assignment command)
  - Repeat instructions Loop
  - Branch conditionals If
- Output
  - Give information print

Features of Algorithm:-

According to D.E.Knuth, a pioneer in the computer science discipline, an algorithm has five important features they are as follows

1. Finiteness
2. Definiteness
3. Effectiveness
4. Input
5. Output

Advantages of algorithm

- it is a step-by-step rep. of a solution to a given prblem ,which is very easy to understand
- it has got a definite procedure.
- it easy to first develope an algorithm,&then convert it into a flowchart &then into a computer program.
- it is independent of programming language.
- it is easy to debug as every step is got its own logical sequence.

Disadvantages of algorithm

It is time consuming & cubersome as an algorithm is developed first which is converted into flowchart &then into a computer program.

Example :-

Write an algorithm that reads two values, determines the largest value and prints the largest value with an identifying message.

Step 1: *Input* VALUE1, VALUE2
Step 2: *if (*VALUE1 > VALUE2) *then*
MAX ← VALUE1
else
MAX ← VALUE2
endif
Step 3: *Print "The largest value is", MAX*

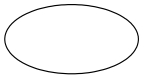
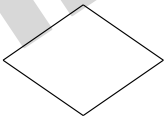
Flowchart:-

A **flowchart** is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows.This diagrammatic representation illustrates a solution to a given problem. Process operations are represented in these boxes, and arrows; rather, they are implied by the sequencing of operations. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.[

- **(Dictionary) A schematic representation of a sequence of operations, as in a manufacturing process or computer program.**
- **(Technical) A graphical representation of the sequence of operations in an information system or program. Information system flowcharts show how data flows from source documents through the computer to final distribution to users. Program flowcharts show the sequence of instructions in a single program or subroutine.**

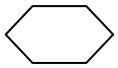
Symbols:-

Different symbols are used to draw each type of flowchart.

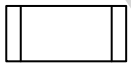
Name	Symbol	Use in Flowchart
Oval		Denotes the beginning or end of the program
Parallelogram		Denotes an input operation
Rectangle		Denotes a process to be carried out e.g. addition, subtraction, division etc.
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes. (e.g. IF/THEN/ELSE)
Hybrid		Denotes an output operation
Flow line		Denotes the direction of logic flow in the program

2. Additional Symbols

Related to more advanced programming



Preparation (may be used with "do loops" explained later)



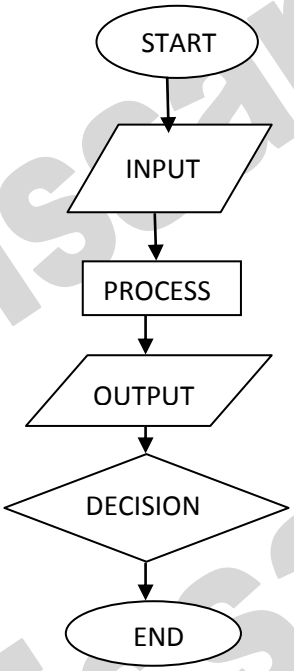
Refers to separate flowchart ("Subprograms"(explained later) are shown in separate flowcharts).

Types of flowchart:-

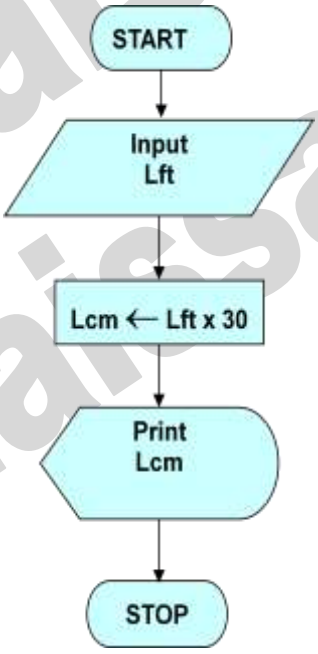
Sterneckert (2003) suggested that flowcharts can be modeled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:

- *Document flowcharts*, showing controls over a document-flow through a system
- *Data flowcharts*, showing controls over a data-flow in a system
- *System flowcharts* showing controls at a physical or resource level
- *Program flowchart*, showing the controls in a program within a system

Program Flowchart
- shows the sequence of instructions in a program or subroutine. These instructions are followed to procedure the needed output.



Write an algorithm and draw a flowchart to convert the length in feet to centimeter



Advantages and limitation of flowchart:-

1. Better communication
2. Proper program documentation
3. Efficient coding
4. Systematic debugging
5. Systematic testing

Limitation of flowchart:-

1. Flowchart are very time consuming and laborious to draw.
2. There are no standards determining the amount of detail that should be included in flowchart.
3. Owing to the symbol-string nature of flowcharting, any change or modification in the program logic will usually require a completely new flowchart.

Making of sequence, selection and iteration:

The concept of structured programming says that any programming logic problem can be solved using an appropriate combination of only three programming structures, none of which are complicated. The three structures are known generally as:

- Sequence
- Selection or decision
  - If statement
  - Switch
- Iteration
  - Looping

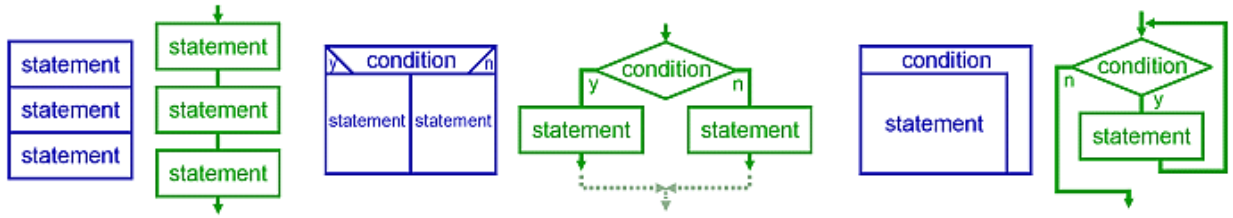
Sequence:-

The general requirement for the sequence structure is that one or more actions may be performed in sequence after entry and before exit.
With the exception discussed below, there may not be any branches or loops between the entry and the exit.
All actions must be taken in sequence.
Enter
Perform one or more actions in sequence
Exit

Selection or decision:-

There is only one entry point and one exit point.
The first thing that happens following entry is that some condition is tested for true or false.
If the condition is true, one or more actions are taken in sequence and control exits the structure.
If the condition is false, **none**, one or more actions are taken in sequence and control exits the structure.
Enter
Test a condition for true or false
On true
Take one or more actions in sequence
On false
Take none, one, or more actions in sequence
Exit

Iteration:-
Perform the test and exit on false
Perform some actions and repeat the test on true
Each action element may be another structure
Need to avoid infinite loops
Enter
Test a condition for true or false
Exit on false
On true
Perform one or more actions in sequence
Go back and test the condition again



Translator:-

Assembler
An **assembler** translates assembly language into machine code. Assembly language consists of mnemonics for machine opcodes so assemblers perform a 1:1 translation from mnemonic to a direct instruction. For example:
LDA #4 converts to 0001001000100100
Conversely, one instruction in a high level language will translate to one or more instructions at machine level.

Advantages of using an Assembler:
- Very fast in translating assembly language to machine code as 1 to 1 relationship
- Assembly code is often very efficient (and therefore fast) because it is a low level language
- Assembly code is fairly easy to understand due to the use of English-like mnemonics

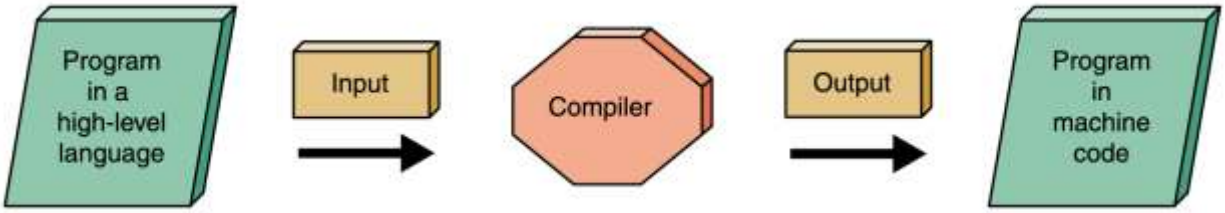
Disadvantages of using Assembler:
- Assembly language is written for a certain instruction set and/or processor
- Assembly tends to be optimised for the hardware it's designed for, meaning it is often incompatible with different hardware
- Lots of assembly code is needed to do relatively simple tasks, and complex programs require lots of programming time

Compiler
A **Compiler** is a computer program that **translates code** written in a high level language to a lower level language, object/machine code. The most common reason for translating source code is to create an executable program (converting from a high level language into machine language).
Advantages of using a compiler
- ource code is not included, therefore compiled code is more secure than interpreted code
- Tends to produce faster code than interpreting source code

- Produces an executable file, and therefore the program can be run without need of the source code

Disadvantages of using a compiler

- Object code needs to be produced before a final executable file, this can be a slow process
- The source code must be 100% correct for the executable file to be produced



Compilation process

Interpreter

An interpreter program executes other programs directly, running through program code and executing it line-by-line. As it analyses every line, an interpreter is slower than running compiled code but it can take less time to interpret program code than to compile and then run it — this is very useful when prototyping and testing code. Interpreters are written for multiple platforms, this means code written once can be run immediately on different systems without having to recompile for each. Examples of this include flash based web programs that will run on your PC, MAC, games console and Mobile phone.

Advantages of using an Interpreter

- Easier to debug(check errors) than a compiler.
- Easier to create multi-platform code, as each different platform would have an interpreter to run the same code.
- Useful for prototyping software and testing basic program logic.

Disadvantages of using an Interpreter

- Source code is required for the program to be executed, and this source code can be read making it insecure
- Interpreters are generally slower than compiled programs due to the per-line translation method

Linker and Loader:-

A **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.

A loader brings object program into memory for execution. System program that performs Loading. Some loaders also do relocation and linking.

Absolute loader:- No linking or relocation. All functions are performed in one pass.

E.g. a Bootstrap Loader

Computer programs typically comprise several parts or modules; all these parts/modules need not be contained within a single object file, and in such case refer to each other by means of symbols. Typically, an object file can contain three kinds of symbols:

- defined symbols, which allow it to be called by other modules,
- undefined symbols, which call the other modules where these symbols are defined, and
- local symbols, used internally within the object file to facilitate relocation.

For most compilers, each object file is the result of compiling one input source code file. When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

Linkers can take objects from a collection called a *library*. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.

The linker also takes care of arranging the objects in a program's address space. This may involve *relocating* code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores.
The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory: every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

Unit II

History of C:-

C language is a structure oriented programming language, was developed at Bell Laboratories in 1972 by Dennis Ritchie
C language features were derived from earlier language called "B" (Basic Combined Programming Language – BCPL)
C language was invented for implementing UNIX operating system
In 1978, Dennis Ritchie and Brian Kernighan published the first edition "The C Programming Language" and commonly known as K&R C
In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

C standards
C89/C90 standard – First standardized specification for C language was developed by American National Standards Institute in 1989. C89 and C90 standards refer to the same programming language.
C99 standard – Next revision was published in 1999 that introduced new futures like advanced data types and other changes.

Features of C language:

- Reliability
- Portability
- Flexibility
- Interactivity
- Modularity
- Efficiency and Effectiveness

Uses of C language:
C language is used for developing system applications that forms major portion of operating systems such as Windows, UNIX and Linux. Below are some examples of C being used.

- Database systems
- Graphics packages
- Word processors
- Spread sheets
- Operating system development
- Compilers and Assemblers
- Network drivers
- Interpreters

We are going to learn a simple "Hello World" program in this section. Functions, syntax and the basics of a C program are explained below.
C Basic Program:

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

```
{
/* Our first simple C basic program */
printf("Hello World! ");
getch();
}
```

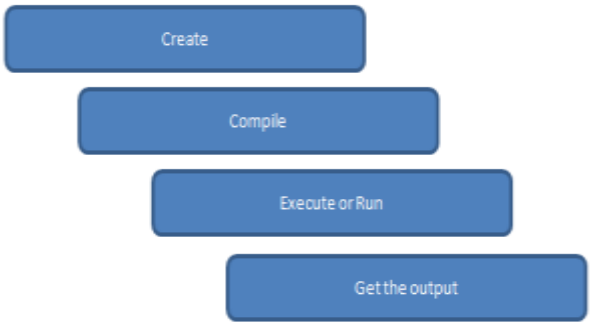
Output:
Hello World!

Explanation for above C basic Program:
Let's see all the sections of the above simple C program line by line.

S.no	Command	Explanation
1	#include <stdio.h>	This is a preprocessor command that includes standard input output header file(stdio.h) from the C library before compiling a C program
2	int main()	This is the main function from where execution of any C program begins.
3	{	This indicates the beginning of the main function.
4	/*_some_comments_*/	whatever is given inside the command "/* */" in any C program, won't be considered for compilation and execution.
5	printf("Hello_World!");	printf command prints the output onto the screen.
6	getch();	This command waits for any character input from keyboard.
7	return 0;	This command terminates C program (main function) and returns 0.
8	}	This indicates the end of the main function.

Steps to write C programs and get the output:

Below are the steps to be followed for any C program to create and get the output. This is common to all C program and there is no exception whether its a very small C program or very large C program.



Creation, Compilation and Execution of a C program:
Prerequisite:

- If you want to create, compile and execute C programs by your own, you have to install C compiler in your machine. Then, you can start to execute your own C programs in your machine.
- You can refer below link for how to install C compiler and compile and execute C programs in your machine.

✶ Once C compiler is installed in your machine, you can create, compile and execute C programs as shown in below link.

Basic structure of C program:

Structure of C program is defined by set of rules called protocol, to be followed by programmer while writing C program. All C programs are having sections/parts which are mentioned below.

1. Documentation section
2. Link Section
3. Definition Section
4. Global declaration section
5. Function prototype declaration section
6. Main function
7. User defined function definition section

Printf and Scanf

C printf():

The printf() function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen.

To display the value of an integer variable, we use printf statement with the `%d` format specifier. Similarly `%c` for character, `%f` for float variable,`%s` for string variable, `%lf` for double, `%x` for hexadecimal variable.

To generate a newline,we use \n in C printf statement.

%d got replaced by value of an integer variable(no),
%c got replaced by value of a character variable(ch),
%f got replaced by value of a float variable(flt),
%lf got replaced by value of a double variable(dbl),
%s got replaced by value of a string variable(str),

C scanf :

scanf() function is used to read a character, string, numeric data from keyboard.

Consider the below example where the user enters a character and assign it to the variable ch and enters a string and assign it to the variable str.

The format specifier %d is used in scanf statement so that the value entered is received as an integer and %s for string.

Ampersand is used before the variable name ch in scanf statement as &ch. It is just like in a pointer which is to point to the variable.

Data Types

✶ C data types are defined as the data storage format that a variable can store a data to perform a specific operation.
✶ Data types are used to define a variable before to use in a program.
✶ Size of variable, constant and array are determined by data types.

C – data types:

There are four data types in C language. They are,

S.no	Types	Data Types
1	Basic data types	int, char, float, double
2	Enumeration data type	enum
3	Derived data type	pointer, array, structure, union
4	Void data type	void

1. Basic data types

Integer data type:

This 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 or 4 or 8 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. Like wise, 4 byte(32 bit) of memory for 32 bit processor and 8 byte(64 bit) of memory for 64 bit processor is allocated for int.

int(2 byte) can store values from -32,768 to +32,767
int(4 byte) can store values from -2,147,483,648 to +2,147,483,647.
If you want to use the integer value that crosses the above limit, you can go for long int and long long int for which the limits are very high.

Character data type:

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

Floating point data type:

Floating point data type consists of 2 types. They are, float and double.

float:

Float data type allows a variable to store decimal values. storage size of float data type is 4. This also varies depend upon the processor in the CPU. We can use upto 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 upto 10 digits after decimal. and the range is from 1E–37 to 1E+37.

C – sizeof() function:

sizeof () operator is used to find the memory space allocated for each C data types.

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6
7      int a;
8      char b;
9      float c;
10     double d;
11     printf("Storage size for int data type:%d \n",sizeof(a));
12     printf("Storage size for char data type:%d \n",sizeof(b));
13     printf("Storage size for float data type:%d \n",sizeof(c));
14     printf("Storage size for double data type:%d \n",sizeof(d));
15     return 0;
16 }
```

Output:

```
Storage size for int data type:4
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8
```

C – Modifiers

The
amount of memory space to be allocated for a variable is derived by modifiers. Modifiers are prefixed with basic data types to modify (either increase or decrease) the amount of storage space allocated to a variable.
For example, storage space for int data type is 4 byte for 32 bit processor.
We can increase the range by using long int which is 8 byte.
We can decrease the range by using short int which is 2 byte.

There are 5 modifiers available in C language. They are,

1. short
2. long
3. signed
4. unsigned
5. long long

Below table gives the detail about the storage size of each C basic data type in 16 bit processor.
Please keep in mind that storage size and range for int and float will vary depend on the CPU processor (8,16, 32 and 64 bit)

S.No	C Data types	storage Size	Range

1	char	1	–127 to 127
2	int	2	–32,767 to 32,767
3	float	4	1E–37 to 1E+37 with six digits of precision
4	double	8	1E–37 to 1E+37 with ten digits of precision
5	long double	10	1E–37 to 1E+37 with ten digits of precision
6	long int	4	–2,147,483,647 to 2,147,483,647
7	short int	2	–32,767 to 32,767
8	unsigned short int	2	0 to 65,535
9	signed short int	2	–32,767 to 32,767
10	long long int	8	–(2power(63) –1) to 2(power)63 –1
11	signed long int	4	–2,147,483,647 to 2,147,483,647
12	unsigned long int	4	0 to 4,294,967,295
13	unsigned long long int	8	2(power)64 –1

2. Enumeration data type:

Enumeration data type consists of named integer constants as a list. It start with 0(zero) by default and value is incremented by 1 for the sequential identifiers in the list.2. Enumeration data type:

Enum syntax in C:

enum identifier [optional{ enumerator-list }]

Enum example :

enum month { Jan, Feb, Mar }; or

3. Derived data type:

Array, pointer, structure and union are called derived data type.

4. Void data type:

Void is an empty data type that has no value. This can be used in functions and pointers.

C – Tokens and keywords

C Tokens:

C Tokens: These are the basic buildings blocks in C language which are constructed together to write a C program.

Each and every smallest individual unit in a C program is known as C Tokens.

C Tokens are of six types.

1) Keywords (eg: int, while),
2) Identifiers (eg: main, total),
3) Constants (eg: 10, 20),
4) Strings (eg: “total”, “hello”),

5) Special symbols (eg: (), {}),
6) Operators (eg: +, /,-,*)

C – Identifiers:
Each program elements in a C program are given a name called identifiers.
Names given to identify Variables, functions and arrays are examples for identifiers. eg. x is a name given to integer variable in above program.

Rules for constructing identifier name:
1. First character should be an alphabet or underscore.
2. Succeeding characters might be digits or letter.
3. Punctuations and special characters aren't allowed except underscore.
4. Identifiers should not be keywords.

C – Keywords
Keywords are pre-defined words in a C compiler.
Each keyword is meant to perform a specific function in a C program.
Since keywords are referred names for compiler, they can't be used as variable name.

C language supports 32 keywords which are given below.

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

C – Constant

C Constant are also like normal variables except their values cannot be modified by the program once they are defined. They refer to fixed values. They are also called as literals
They may be belonging to any of the data type. Please see below table for constants with respect to their data type.

Types of C constant:
1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

S.no	Constant type	data type	Example
1	Integer constants	int unsigned int	53 , 762 , -478 , etc 5000u , 1000U , etc

		long int long long int	1000L , -300L , etc 5555555LL
2	Real or Floating point constants	float, doule	111.22F , 2.22e-2f 111.22 , 4.0 , -0.34565
3	Octal constant	int	013 /* starts with 0 */
4	Hexadecimal constant	int	0×90 /* starts with 0x */
5	character constants	char	'A' , 'B', 'C'
6	string constants	char	"ABCD" , "Hai"

Rules for constructing C constant:

- **Integer Constants**
  1. An integer constant must have at least one digit.
  2. It must not have a decimal point.
  3. It can either be positive or negative.
  4. No commas or blanks are allowed within an integer constant.
  5. If no sign precedes an integer constant, it is assumed to be positive.
  6. The allowable range for integer constants is -32768 to 32767
- **real Constants**
  1. A real constant must have at least one digit
  2. It must have a decimal point
  3. It could be either positive or negative
  4. If no sign precedes an integer constant, it is assumed to be positive.
  5. No commas or blanks are allowed within a real constant.
- **character constants**
  1. A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to left.
  2. For example, 'C' is a valid character constant whereas 'C' is not.
     The maximum length of a character constant is 1 character

Backslash Character Constants:

There are some characters which have special meaning in C language. They should be preceded by back slash symbol to make use of special function of them. Given below is the list of special characters and their purpose.

Character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return

\t	Horizontal tab

How to use constants in a C program?
We can define constants in a C program in the following ways.

1. By "const" keyword
2. By "#define" preprocessor directive

C – Variable

- C variable is a named location in a memory where a program can manipulate the data. This location is used to hold the value of the variable.
- The value of the C variable may get change in the program.
- C variable might be belonging to any of the data type like int, float, char etc.

Rules for naming C variable:

1. Variable name must begin with letter or underscore.
2. Variables are case sensitive
3. They can be constructed with digits, letters.
4. No special symbols are allowed other than underscore.
5. sum, height, _value are some examples for variable name

Declaring & initializing C variable:

- Variables should be declared in the C program before to use.
- Memory space is not allocated for a variable while declaration. It happens only on variable definition.
- Variable initialization means assigning a value to the variable.

S.No	Type	Syntax	Example
1	Variable declaration	data_type variable_name;	int x, y, z; char flat, ch;
2	Variable initialization	data_type variable_name = value;	int x = 50, y = 30; char flag = 'x', ch='l';

There are three types of variables in C program They are,

1. Local variable
2. Global variable
   Environment variable

Operators and Expressions

An operator is a symbol that performs specific mathematical or logical manipulations in program C is flexible and powerful language, because it has rich set of operators.
The symbols which are used to perform logical and mathematical operations are called C operators. These C operators join individual constants and variables to form expressions. Operators, functions, constants and variables are combined together to form expressions.
Example : A + B * 5
where,

- +, * - operators

- A, B - variables
- 5 – constant
- A + B * 5 - expression

Types of C operators:
C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

1) Arithmetic Operators:
These operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

S.No.	Operator	Operation
1	+	Addition
2	-	Subtraction
3	*	multiplication
4	/	Division
5	%	Modulus

Operand 1 Operators Operand 2
Eg. 6 + 4
7 x 3
5 / 2
6 / 2

2) Assignment operators: The symbol = is also an operators, and it is evaluated last. Here the equal TO (=) symbol should not be treated as in mathematics. Here it means that the value of the expression in the right-hand side is taken by the variable to the left side.

eg : a = 2 + 7 * 3

3) Increment and Decrement Operator: C provides two special operators not generally found in other languages. These are the increment (++) and decrement the value of a variable by l. These operators are called unary operators and require only one operand. They are of equal precedence.

These operators can be used both as prefix and postfix.

Eg. ++ variable

Variable ++

-- variable
variable --

4) Relational operators : In C there are 6 operators which are used to compare two values or expressions. The values of two expressions with the relational operator forms a relational expressions. These 6 operators.

These operators are used to find the relation between two variables. That is, used to compare the value of two variables.

Operator	Example
>	x > y
<	x < y
>=	x >= y
<=	x <= y
==	x == y
!=	x != y

5) Logical operators: The logical operators are sued to support the basic logical operations of AND, OR C NOT according to the truth table. For combining expressions into logical expression logical operators are used. These are

Symbol	Name
&&	logical AND
! !	logical OR
!	logical Not

(a) Logical AND operators: This is used to combine two expressions in the form **exp 1 & & exp 2.**
If both exp 1 and exp 2 are true then the whole expression is considered as true. If any one of them is false, the whole expression will considered as false.

(b) Logical OR operator: It also combines the two expressions. If any one of the expression is true the whole expression is considered as true.

Exp1 !! Exp2

(c) Logical NOT operator: It inverts the logical value of an expression.

! Exp

6) Bitwise operators: One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various bitwise operators available in C are :

Operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
<<	shift left
~	one's complement

Bitwise operators can operator upon int and char but not on float and double. All are binary operators except ~ which is unary.

(a) Bitwise AND (&): The bitwise AND (&)operator yields 1 if the corresponeting bits in both values are 1 otherwise, it yields 0.

eg.
```
    1 0 1 0
&   0 1 1 0
  ---------
    0 0 1 0
```

The use of AND operator is to check whether a particular bit of an operand is ON/OFF.

(b) Bitwise OR (!): Another important bitwise operator is OR operator which is represented by ! The rules that given the value of the resulting bit obtained after OR of two bits is shown below :

!	0	1
0	0	1
1	1	1

The result of an inclusive OR is a 1 if either (or both) of the corresponding bits is 1.

eg.
```
  1 0 1 0
  0 1 1 0
  -------
  1 1 1 0
```

Bitwise OR operator is usually used to put ON a particular bit in a number.

(c) Bitwise XOR (^): The XOR operator is represented as ^ and is also called on exclusive OR operator. The OR operator returns 1, when any one of the two bits or bith the bits are 1, whereas XOR returns 1 only if one of the two bits is 1. The truth table for the XOR operator is given below :

^	0	1
0	0	1
1	1	0

(e) Right Shift Operator: Like one's complement operator right shift operator also operates on a single variable. It's represented by >> and it shifts each bit in the operand to right. The number of places the bits are shifted depends on the number following the operand.

Thus, num >>3 would shift all bits three places to the right. Similarly num >> would shift all bits five places to the right.

eg. If num = 110011110 then

num >> 1 given 011001111
num >> 3 gives 000110011
num >> 5 gives 000001100

While bits are shifted to right blanks are created on the left. These left blanks must be filled with zeros.

(e) Left Shift operator: This is similar to the right shift operator the only difference being that the bits are shifted to the left and for each bit shifted, added to the right of the number.

eg. If num =11010111 then

num << 1 gives 10101110
num << 3 gives 10111000
num << 5 gives 11100000

Precedence and order of evaluation of operators :

7) Conditional operators:

Conditional operators return one value if condition is true and returns other value is condition is false.

This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

Example : (A > 100 ? 0 : 1);

Here, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

8. Special Operators:

Below are some of special operators that the C language offers.

S.No.	Operators	Description
1	**&**	This is used to get the address of the variable. Example: &a will give address of a.
2	*****	This is used as pointer to a variable. Example: * a where, * is pointer to the variable a.

3	**Size of ()**	This gives the size of the variable. Example: size of (char) will give us 1.
4	**ternary**	This is ternary operator and act exactly as if … else statement.
5	**Type cast**	Type cast is the concept of modifying variable from one data type to other.

Preprocessor directives

Preprocessor Directives extends the power of C programming language. C Preprocessors Directives is a special set of instruction in program which is processed before handing over the program to the compiler. These instructions are always preceded by a pound sign (#) and NOT terminated by a semicolon. The preprocessor directives are executed before the actual compilation of code begins. Different preprocessor directives (commands) tell the preprocessor to perform different actions. We can categorize the Preprocessor Directives as follows:
File inclusion directives
Macro substitution directives
Conditional compilation directives

Advantages:
It make program development easy
It enhance the readability of the program
It make modification easy
It increase the transportability of the program between different machine architectures.
File inclusion directives
The file inclusion directive is used to include files into the current file. When the preprocessor encounters an #include directive it replaces it by the entire content of the specified file.The file inclusion directive (#include) can be used in following two ways:
#include "file-name"
#include <file-name>

We can see that #include can be used in two ways angular brackets (<>)and inverted commas (""). The only difference between both expressions is the location (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the compiler will look for the in the current directory that includes the file containing the directive. In case if it is not there the compiler will look the file in the default include directories where it is configured to look for the standard header files. When #include is written with <> it means the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

Standard Header Files
It is the set of header files those used for performing various common and standard operations.
stdio.h - Defines core input and output functions
string.h - Defines string handling functions.
time.h - Defines date and time handling functions
math.h - Defines common mathematical functions.
User Defined Header Files

In C Programming Language user can have their own custom header file that provide additional capabilities.

Macro substitution directives

Macro substitution directives are used to define identifier which is being replaced by a pre defined string in program.Macro substitution is process in which preprocessor replaces identifiers by one, or more program statements (like functions) and they are expanded inline.

There are two directives for Macro Definition:

#define – Used to define a macro

#undef – Used to undefine a macro

#define preprocessor

To define preprocessor macros we can use `#define`.

Conditional compilation directives(#ifdef, #ifndef, #if, #endif, #else and #elif)

Conditional Compilation Directives allows you to include or exclude certain part of code only when certain condition met.These macros are evaluated on compile time.

The following directives are included in this category:

- #if
- #elif
- #endif
- #ifdef
- #ifndef

Header Files

A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive**#include** like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source files, specially if we have multiple source file comprising our program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

Include Syntax

Both user and system header files are included using the preprocessing directive **#include**. It has following two forms:

```
#include <file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

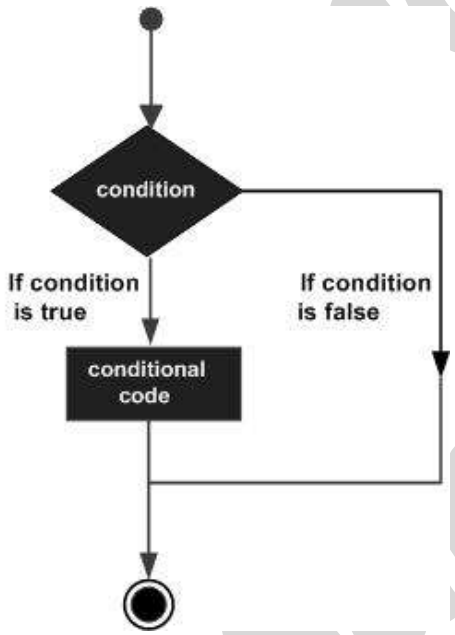
This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can pretend directories to this list with the -I option while compiling your source code.

UNIT-III

DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides following types of decision making statements.

Statement	Description
if statement	An **if statement** consists of a test condition followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**, which executes when the test condition is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one **switch** statement inside another **switch** statement(s).

The if Statement

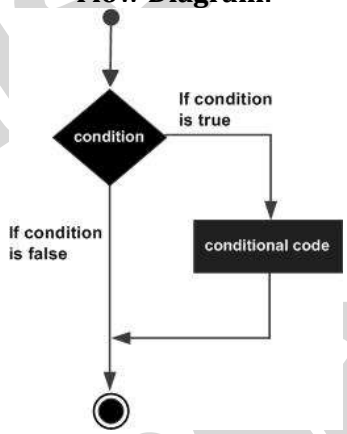
An **if** statement consists of a test condition followed by one or more statements.

Syntax:
The syntax of an if statement in C programming language is:

```
if(test condition )
{
  /* statement(s) will execute if the test condition is true */
}
```

If the test condition evaluates to **true**, then the block of code inside if statement will be executed. If test condition evaluates to **false**, then the first set of code after the end of if statement (after the closing curly brace) will be executed.
C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram:



Example:

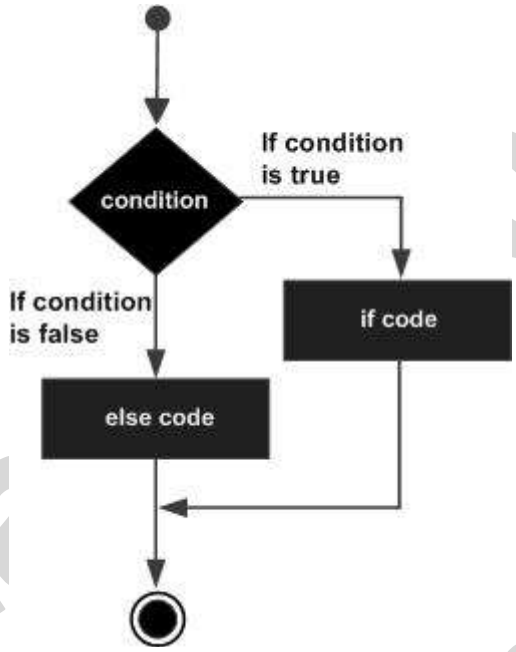
- if(a>b)
- printf("a is greater than b");

The If...Else Statement
An **if** statement can be followed by an optional **else** statement, which executes when the test condition is false.
Syntax:
The syntax of an **if...else** statement in C programming language is:

```
if(test condition )
{
  /* statement(s) will execute if the test condition is true */
}
else
{
  /* statement(s) will execute if the test condition is false */
}
```

If the test condition valuates to **true**, then if **block** of code will be executed, otherwise **else block** of code will be executed.
C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram:



Nested if else

It is always legal in C programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax:

The syntax for a **nested if** statement is as follows:

```
if( test condition  1)
{
  /* Executes when the test condition1 is true */
  if(test condition  2)
  {
    /* Executes when the test condition2 is true */
  }
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

The if else ladder

The if….elseif…else statement is used to select one of among several blocks of code to be executed.

- if(expression1)
- {
- Statement 1;
- }
- else if(expression2)
- {
- Statement 2;

- }
- .
- .
- else
- {
- Statement n;
- }

The Switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax:

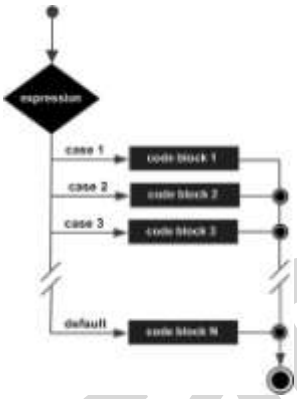
The syntax for a **switch** statement in C programming language is as follows:

```
switch(expression){
   case constant-expression  :
     statement(s);
     break; /* optional */
   case constant-expression  :
     statement(s);
     break; /* optional */
    /* you can have any number of case statements */
   default : /* Optional */
     statement(s);
}
```

The following rules apply to a switch statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will fall through to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram:

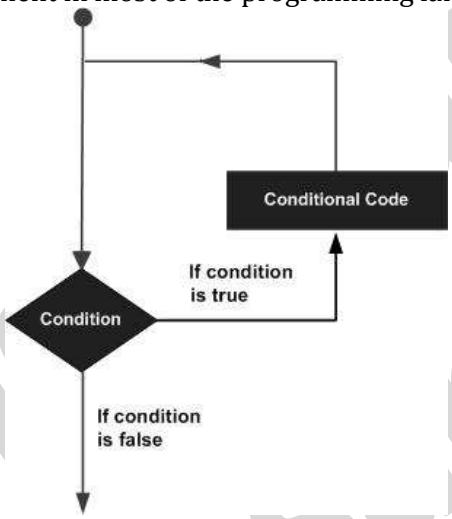


Loop control statements

There may be a situation, when you need to execute a block of code several numbers of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



C programming language provides the following types of loop to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body

nested loops	You can use one or more loop inside any another while, for or do..while loop.

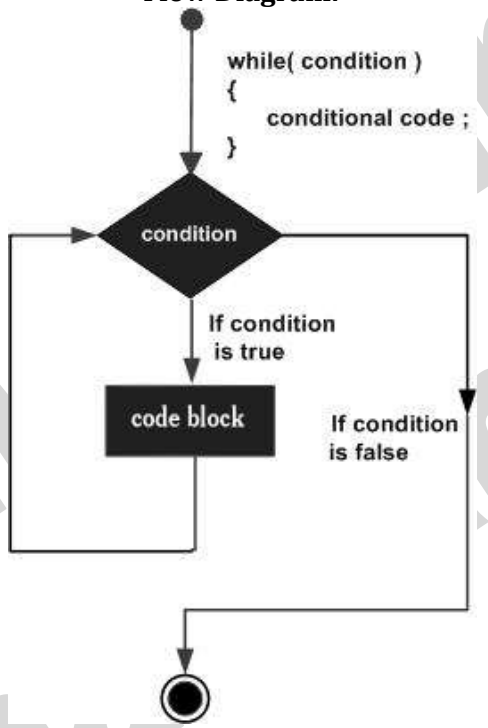
while loop statement:-
A **while** loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:
The syntax of a **while** loop in C programming language is:

```
while(condition)
{
  statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.
When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

```
#include <stdio.h>

int main ()
```

```
{
  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 )
  {
    printf("value of a: %d\n", a);
    a++;
  }

  return 0;
}
```

For

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

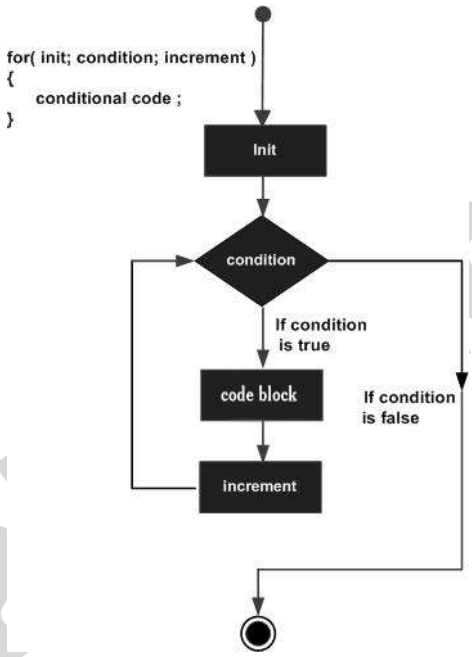
The syntax of a **for** loop in C programming language is:

```
for ( init; condition; increment )
{
  statement(s);
}
```

Here is the flow of control in a for loop:\

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
#include <stdio.h>

void main ()
{
  /* for loop execution */
  for( int a = 10; a < 20; a = a + 1 )
  {
    printf("value of a: %d\n", a);
  }

getch();
}
```

Output:-

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Do While:-

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.
A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

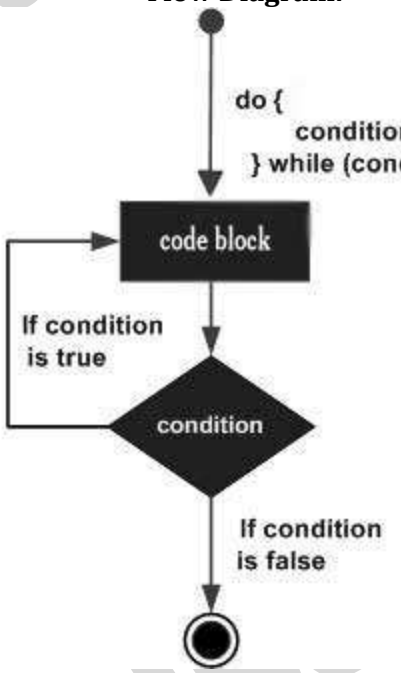
Syntax:
The syntax of a **do...while** loop in C programming language is:

```
do
{
  statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.
If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
#include <stdio.h>

void main ()
{
  /* local variable definition */
  int a = 10;

  /* do loop execution */
  do
```

```
  {
     printf("value of a: %d\n", a);
     a = a + 1;
  }while( a < 20 );

  getch();
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested loop:-

C programming language allows to use one loop inside another loop. Following section shows fewexamples to illustrate the concept.

Syntax:

The syntax for a **nested for loop** statement in C is as follows:

```
for ( init; condition; increment )
{
  for ( init; condition; increment )
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows:

```
while(condition)
{
  while(condition)
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows:

```
do
{
```

```
  statement(s);
  do
  {
    statement(s);
  }while( condition );

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Loop Control Statements:
Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
C supports the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

1. break statement:
Break statement is used to terminate the while loops, switch case loops and for loops from the subsequent execution.
Syntax: break;
2. Continue statement:
Continue statement is used to continue the next iteration of for loop, while loop and do-while loops. So, the remaining statements are skipped within the loop for that particular iteration.
Syntax : continue;
3. goto statements:
goto statements is used to transfer the normal flow of a program to the specified label in the program.
Syntax:

```
{
      …….
      go to label;
      …….
      …….
      Label:
            Statements;
}
```

The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>

int main ()
{

  for( ; ; )
  {
    printf("This loop will run forever.\n");
  }

  return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

Do you know difference between while & do while loop?

S.No	while	do while
1	Control goes into the while loop only when condition is true.	Loop is executed for first time irrespective of the condition. After executing the while loop for first time, condition is being checked.

UNIT IV

Functions in C:
The function is a self contained block of statements which performs a coherent task of a same kind.
C program does not execute the functions directly. It is required to invoke or call that functions. When a function is called in a program then program control goes to the function body. Then, it executes the statements which are involved in a function body. Therefore, it is possible to call function whenever we want to process that functions statements.

Types of functions:

There are 2 types of functions as:
1. Built in Functions
2. User Defined Functions

1. Built in Functions:
These functions are also called as 'library functions'. These functions are provided by system. These functions are stored in library files. e.g.
- scanf()
- printf()
- strcpy
- strlwr
- strcmp
- strlen
- strcat

2. User Defined Functions:
The functions which are created by user for program are known as 'User defined functions'.

Syntax:
```
void main()
{
        // Function prototype
        <return_type><function_name>([<argu_list>]);

        // Function Call
        <function_name>([<arguments>]);
}
// Function definition
<return_type><function_name>([<argu_list>]);
{
        <function_body>;
}
```

Program :
```
#include <stdio.h>
#include <conio.h>
void main()
```

```
{
        void add();
        add();
        getch();
}

void add()
{
        int a, b, c;
        clrscr();
        printf("\n Enter Any 2 Numbers : ");
        scanf("%d %d",&a,&b);
        c = a + b;
        printf("\n Addition is : %d",c);
}
```

Output :

```
Enter Any 2 Numbers : 23 6
Addition is : 29_
```

Advantages :

- It is easy to use.
- Debugging is more suitable for programs.
- It reduces the size of a program.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

A function definition has four components:-

(a) The return- type specifier
(b) The function name
(c) Parameter(argument) list
(d) The body of the function

Syntax

```
Return – type function -name (parameter list) //function header
{
    Variable declarations;
     Statement(s);
}
```

(a) The return type specifier: - the return type specifier identifies the type of value will be returned by the function after performing a task.

(b) The function name :- the function name helps us to uniquely identify and call a function

(c) The parameter(argument) list:-the parameter list identifies
The set of values, which are to be passed to the function from either the main program or a sub program.

(d) Body of the function: - the body of the function includes the variable declarations which are required to perform a task. These variables are local to the function body i.e. they cannot be used outside the function body.

LOCAL AND GLOBAL VARIABLES

Local

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global

These variables can be accessed (i.e. known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

```
/* Demonstrating Global variables */
#include <stdio.h>
int add_numbers( void );

/* These are global variables and can be accessed by functions from this point on */
int  value1, value2, value3;

int add_numbers( void )
{
 auto int result;
 result = value1 + value2 + value3;
 return result;
}

main()
{
 auto int result;
 value1 = 10;
 value2 = 20;
 value3 = 30;
 result = add_numbers();
 printf("The sum of %d + %d + %d is %d\n",
        value1, value2, value3, final_result);
}
```

Sample Program Output
The sum of 10 + 20 + 30 is 60

Call by value and call by reference

1. **Call By Value:**

Creates a new memory location for use within the subroutine. The memory is freed once it leaves the subroutine. Changes made to the variable are not affected outside the subroutine.

Call By Reference:

Passes a pointer to the memory location. Changes made to the variable within the subroutine affects the variable outside the subroutine.

2. In call by value, both the actual and formal parameters will be created in different memory locations whereas if they are called by reference both will be created at the same location.

3. In call by value method, a compiler get a copy of the variable and thus changes made in the value in function
   Will not reflected back to the called function. But in call by reference method, the compiler didn't get any copy, but actually it works on the original copy and thus changes will be reflected back
4. Call by value: call by value means programmer send some value coping from one function to another. At the time of function calling a programmer can send a copy of variable of value. Call by reference: it means sending the address of variable to the called function means a user can send the address of variable.
5. simplest possible method of passing the parameters the actual parameter are evaluated and their r values are passed to the subroutine in location determined by the language implementation
6. Call by value :
   The copy of the argument is passed. e,g., if x and y are arguments and their corresponding values are say 100 and 200.
   c=max(x,y);
   max(int a, int b)

call by reference :
it is sending the address of variables to the called function.

7. In call by value, both the actual and formal parameters will be created in different memory locations whereas if they are called by reference both will be created at the same location and
It is sending the address of variables to the called function .

8 call by value :
call by value means programmer send some value coping from one function to another. At the time of function calling a programmer can send a copy of variable of value.
call by reference :
it means sending the address of variable to the called function means a user can send the address of variable.

9. Call by value method:
Passing the value of variable to the function.

```
void main()
{
int x=10,y=20;
printf("%d%d',x,y);
swap(x,y);
}
void swap(int a,int b)
{
int c;
c=a;//changes here do not affect in values
a=b;//of x and y in main function..
b=c;
}
```

Call by reference method:
Passing the address of variable to the function.

```
swap(&a, &b)
&c=&a;
```

&a=&b;
&b=&c;
changes made in subfunction causes changes in address of variables n thus in main () also..

10. Call By Value :
In call By Value it creates different memory for local variable and actual called Variable. So changes does not affect to the actual variable.

Call By Reference:
In call By Reference Compiler create same memory location for the local variable and actual called variable so changes affect to the actual variable and modify the value of actual called Variable.

Function Declaration and Prototypes
Any C function by default returns an **int** value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type **int**. If we desire that a function should return a value other than an **int**, then it is necessary to explicitly mention so in the calling function as well as in the called function. Suppose we want to find out square of a number using a function. This is how this **simple program would look like:**

```
main( )
{
float a, b ;
printf ( "\nEnter any number " ) ;
scanf ( "%f", &a ) ;
b = square ( a ) ;
printf ( "\nSquare of %f is %f", a, b ) ;
}
square ( float x )
{
float y ;
y = x * x ;
return ( y ) ;
}
```

And here are three sample runs of this program...

```
Enter any number 3
Square of 3 is 9.000000
Enter any number 1.5
Square of 1.5 is 2.000000
Enter any number 2.5
Square of 2.5 is 6.000000
```

Recursion
Recursion is a technique in which a function calls itself, for example in above code factorial function is calling itself. To solve a problem using recursion you must first express its solution in recursive form.
Factorial program in c using recursion

```
#include<stdio.h>
long factorial(int);
int main()
{
  int n;
  long f;
```

```
 printf("Enter an integer to find factorial\n");
 scanf("%d", &n);
 if (n < 0)
  printf("Negative integers are not allowed.\n");
 else
 {
  f = factorial(n);
  printf("%d! = %ld\n", n, f);
 }

 return 0;
}
long factorial(int n)
{
 if (n == 0)
  return 1;
 else
  return(n * factorial(n-1));
}
```

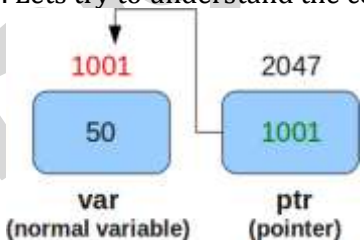
Types of recursion

1. **Infinite Recursion**

• All recursive definitions have to have a non-recursive part
• If they didn't, there would be no way to terminate the recursive path
• Such a definition would cause *infinite recursion*
• This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
• The non-recursive part is often called the *base* case

Eliminating Recursion — Tail
Recursion
A special kind of recursion is tail recursion.
Tail recursion is when a recursive call is the last thing a function does.
Tail recursion is important because it makes the recursion → iteration conversion very easy.
That is, we like tail recursion because it is easy to eliminate.
In fact, tail recursion is such an obvious thing to optimize that some compilers automatically convert it to iteration.

Pointer
What are Pointers?
Different from other normal variables which can store values, pointers are special variables that can hold the address of a variable. Since they store memory address of a variable, the pointers are very commonly said to “point to variables”. Lets try to understand the concept.



As shown in the above diagram:

- A normal variable 'var' has a memory address of 1001 and holds a value 50.
- A pointer variable has its own address 2047 but stores 1001, which is the address of the variable 'var'

How to Declare a Pointer?
A pointer is declared as :
<pointer type> *<pointer-name>

In the above declaration :
pointer-type : It specifies the type of pointer. It can be int, char, float etc. This type specifies the type of variable whose address this pointer can store.
pointer-name : It can be any name specified by the user. Professionally, there are some coding styles which every code follows. The pointer names commonly start with 'p' or end with 'ptr'

An example of a pointer declaration can be :
char *chptr;

In the above declaration, 'char' signifies the pointer type, chptr is the name of the pointer while the asterisk '*' signifies that 'chptr' is a pointer variable.

How to initialize a Pointer?
A pointer is initialized in the following way :
<pointer declaration(except semicolon)> = <address of a variable>
Note that the type of variable above should be same as the pointer type.(Though this is not a strict rule but for beginners this should be kept in mind).
For example :
char ch = 'c';
char *chptr = &ch; //initialize
In the code above, we declared a character variable ch which stores the value 'c'. Now, we declared a character pointer 'chptr' and initialized it with the address of variable 'ch'.
Note that the '&' operator is used to access the address of any type of variable.

How to Use a Pointer?
A pointer can be used in two contexts.

- For accessing the address of the variable whose memory address the pointer stores.

Again consider the following code :

```
char ch = 'c';
char *chptr = &ch;
```

Now, whenever we refer the name 'chptr' in the code after the above two lines, then compiler would try to fetch the value contained by this pointer variable, which is the address of the variable (ch) to which the pointer points. i.e. the value given by 'chptr' would be equal to '&ch'.

For example :
char *ptr = chptr;

The value held by 'chptr' (which in this case is the address of the variable 'ch') is assigned to the new pointer 'ptr'.

- For accessing the value of the variable whose memory address the pointer stores.

Continuing with the piece of code used above :
char ch = 'c';
char t;
char *chptr = &ch;
t = *chptr;

We see that in the last line above, we have used '*' before the name of the pointer. What does this asterisk operator do?
Well, this operator when applied to a pointer variable name(like in the last line above) yields the value of the variable to which this pointer points. Which means, in this case '*chptr' would yield the value kept at address held by chptr. Since 'chptr' holds the address of variable 'ch' and value of 'ch' is 'c', so '*chptr' yeilds 'c'.

When used with pointers, the asterisk '*' operator is also known as 'value of' operator.
An Example of C Pointers

Consider the following code :

CODE :

```
#include <stdio.h>
int main(void)
{
   char ch = 'c';
   char *chptr = &ch;
   int i = 20;
   int *intptr = &i;
   float f = 1.20000;
   float *fptr = &f;
   char *ptr = "I am a string";
   printf("\n [%c], [%d], [%f], [%c], [%s]\n", *chptr, *intptr, *fptr, *ptr, ptr);
   return 0;
}
```

OUTPUT :
$ ./pointers
[c], [20], [1.200000], [I], [I am a string]

To debug a C program, use gdb. The above code covers all the common pointers. The first three of them are very trivial now to understand so lets concentrate on the fourth one. In the fourth example, a character pointer points to a string.
In C, a string is nothing but an array of characters. So we have no staring pointers in C. Its the character pointers that are used in case of strings too.
Now, coming to the string, when we point a pointer to a string, by default it holds the address of the first character of the string. Lets try to understand it better.
The string, 'I am String' in memory is placed as :

1001	1002	1003	1004	1005	1006	1007	1008	1009	1010
I	a	m	S	t	r	i	n	g	\0

Since characters occupy one byte each, so they are placed like above in the memory. Note the last character, its a null character which is placed at the end of every string by default in C. This null character signifies the end of the string.

Now coming back to the point, any character pointer pointing to a string stores the address of the first character of the string. In the code above, 'ptr' holds the address of the character 'I' ie 1001. Now, when we apply the 'value of' operator '*' to 'ptr', we intend to fetch the value at address 1001 which is 'I' and hence when we print '*ptr', we get 'I' as the output.
Also, If we specify the format specifier as '%s' and use 'ptr' (which contains the starting address of the string), then the complete string is printed using printf. The concept is that %s specifier requires the address of the beginning byte of string to display the complete string, which we provided using 'ptr' (which we know holds the beginning byte address of the string). This we can see as the last print in the output above.

The &(address-of)operator
It is a unary operator that return the memory address of its operand. In other words, the & operator is used to get the address of any type of variable(like int, struct, union etc)
Pointer=&variable;

The *(indirection operator)
The asterisk(*) is the indirection operator,which is used to declare pointer variable(s)
For example
Int *ptr,*qty;

Pointer arithmetic
There are four arithmetic operators that can be used on pointers:
++, --, +, -
C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.)
Consider
<datatype> *ptrn; //datatype can be int, long, etc.
Unary Pointer Arithmetic Operators
Operator ++: Adds sizeof(datatype) number of bytes to pointer, so that it points to the next entry of the datatype.
Operator −−: Subtracts sizeof(datatype) number of bytes to pointer, so that it points to the next entry of the datatype.

```
#include <stdio.h>
int main()
{
int *ptrn;
long *ptrlng;
ptrn++; //increments by sizeof(int) (4 bytes)
ptrlng++; //increments by sizeof(long) (8 bytes)
return 0;
}
```

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately. Suppose you have a pointer to a long.
long *ptrlng;
ptrlng+n is valid, if n is an integer. The result is the following
byte address
ptrlng + n*sizeof(long)
and not ptrlng + n.
It advances the pointer by n number of longs.
ptrlng-n is similar.

Consider two pointers ptr1 and ptr2 which point to the same type
of data.
<datatype> *ptr1, *ptr2;
Binary operations between two Pointers
Surprise: Adding two pointers together is not allowed!
ptr1 - ptr 2 is allowed, as long as they are pointing to
elements of the same array. The result is
ptr1 - ptr2
sizeof(datatype)
In other settings, this operation is undefined (may or may not give the correct answer). Why all these special cases? These rules for pointer arithmetic are intended to handle addressing inside arrays correctly.
If we can subtract a pointer from another, all the relational operations can be supported!
Local Oations on Pointers
1 ptr1 > ptr2 is the same as ptr1 - ptr2 > 0,
2 ptr1 = ptr2 is the same as ptr1 - ptr2 = 0,
3 ptr1 < ptr2 is the same as ptr1 - ptr2 < 0,
4 and so on.

Pointer to array
I access the members of the array by using the pointer in a seperate function. I'm wondering because I have an array I have to access in functions other than the one it was defined in, but I need it to be defined in that function, and not globally, because the size is dependant on the user input...
Which is how I came to the pointer to the array idea... that should let me access the data in other functions right? If so, how do I do that and then access the objects from the pointer?
Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];
   for ( i = 0; i < MAX; i++)
  {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }
  for ( i = 0; i < MAX; i++)
  {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

C Function Pointers
Just like pointer to characters, integers etc, we can have pointers to functions.
A function pointer can be declared as :
<return type of function> (*<name of pointer>) (type of function arguments)

For example :
int (*fptr)(int, int)
The above line declares a function pointer 'fptr' that can point to a function whose return type is 'int' and takes two integers as arguments.

Lets take a working example :

```
#include<stdio.h>
int func (int a, int b)
{
   printf("\n a = %d\n",a);
   printf("\n b = %d\n",b);
   return 0;
}

int main(void)
{
   int(*fptr)(int,int); // Function pointer
   fptr = func; // Assign address to function pointer
   func(2,3);
   fptr(2,3);
   return 0;
}
```

In the above example, we defined a function 'func' that takes two integers as inputs and returns an integer. In the main() function, we declare a function pointer 'fptr' and then assign value to it. Note that, name of the function can be treated as starting address of the function so we can assign the address of function to function pointer using function's name. Lets see the output :

```
$ ./fptr
 a = 2
 b = 3
 a = 2
 b = 3
```

UNIT-V

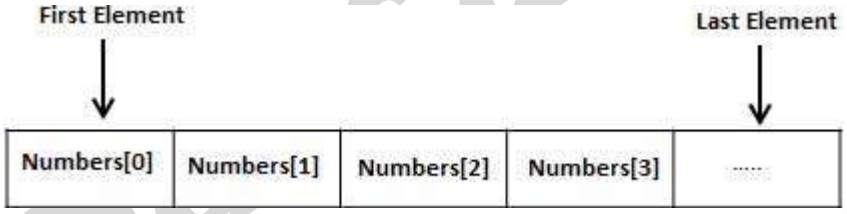
Arrays

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

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [arraySize];

This is called a *single-dimensional* array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

Now *balance* is avariable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

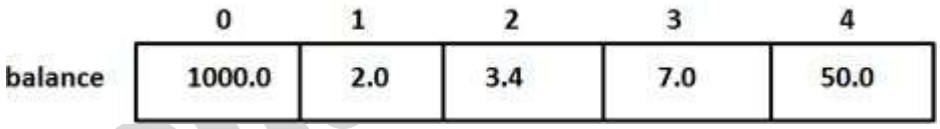
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

balance[4] = 50.0;

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
  int n[ 10 ]; /* n is an array of 10 integers */
  int i,j;

  /* initialize elements of array n to 0 */
  for ( i = 0; i < 10; i++ )
  {
    n[ i ] = i + 100; /* set element at location i to i + 100 */
  }

  /* output each array element's value */
  for (j = 0; j < 10; j++ )
  {
    printf("Element[%d] = %d\n", j, n[j] );
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C Arrays in Detail

Arrays are important to C and should need lots of more details. There are following few important concepts related to array which should be clear to a C programmer:

Concept	Description
Multi-dimensional arrays	C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from a function	C allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Structures

C arrays allow you to define type of variables that can hold several data items of the same kind but**structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

```
struct [structure tag]
{
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

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

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main( )
{
  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */
```

```
  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;

  /* print Book1 info */
  printf( "Book 1 title : %s\n", Book1.title);
  printf( "Book 1 author : %s\n", Book1.author);
  printf( "Book 1 subject : %s\n", Book1.subject);
  printf( "Book 1 book_id : %d\n", Book1.book_id);

  /* print Book2 info */
  printf( "Book 2 title : %s\n", Book2.title);
  printf( "Book 2 author : %s\n", Book2.author);
  printf( "Book 2 subject : %s\n", Book2.subject);
  printf( "Book 2 book_id : %d\n", Book2.book_id);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

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

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

/* function declaration */
void printBook( struct Books book );
```

```
int main( )
{
  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;

  /* print Book1 info */
  printBook( Book1 );

  /* Print Book2 info */
  printBook( Book2 );

  return 0;
}
void printBook( struct Books book )
{
  printf( "Book title : %s\n", book.title);
  printf( "Book author : %s\n", book.author);
  printf( "Book subject : %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

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

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;

  /* print Book1 info by passing address of Book1 */
  printBook( &Book1 );

  /* print Book2 info by passing address of Book2 */
  printBook( &Book2 );

  return 0;
}
void printBook( struct Books *book )
{
  printf( "Book title : %s\n", book->title);
  printf( "Book author : %s\n", book->author);
  printf( "Book subject : %s\n", book->subject);
  printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
```

Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.

C allows us do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit my_int.
C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Unions

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Defining a Union

To define a union, you must use the **union** statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

```
union [union tag]
{
  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

```
union Data
{
  int i;
  float f;
  char  str[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.
The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

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

union Data
{
  int i;
  float f;
  char  str[20];
};

int main( )
{
  union Data data;

  printf( "Memory size occupied by data : %d\n", sizeof(data));

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by data : 20
```

Accessing Union Members
To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

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

union Data
{
  int i;
  float f;
  char  str[20];
};

int main( )
{
  union Data data;

  data.i = 10;
  data.f = 220.5;
  strcpy( data.str, "C Programming");

  printf( "data.i : %d\n", data.i);
  printf( "data.f : %f\n", data.f);
```

```
  printf( "data.str : %s\n", data.str);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that values of **i** and **f** members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

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

union Data
{
  int i;
  float f;
  char  str[20];
};

int main( )
{
  union Data data;

  data.i = 10;
  printf( "data.i : %d\n", data.i);

  data.f = 220.5;
  printf( "data.f : %f\n", data.f);

  strcpy( data.str, "C Programming");
  printf( "data.str : %s\n", data.str);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.