**GOVERNMENT OF TAMILNADU**
**DIRECTORATE OF TECHNICAL EDUCATION**
**CHENNAI – 600 025**

**STATE PROJECT COORDINATION UNIT**

**Diploma in Electronics and Communication Engineering**

**Course Code: 1040**

**M – Scheme**

**e-TEXTBOOK**
**on**

# PROGRAMMING IN "C"

**for**
**III Semester DECE**

**Convener for ECE Discipline:**
**Dr.M.JeganMohan M.E., MBA., Ph.D.,(Management), Ph.D(Eng).,M.I.S.T.E.,**
Principal,
138, Government. Polytechnic College,
Uthappanaickanoor,
Usilampatti, Madurai – 625 536

**Team Members for programming in ''c'':**
**Mrs. C. Aruna Vinodhini B.E., M.TECH.,**
HOD (i/c)/Computer Engineering,
178, Bharathiyar centenary memorial government women's polytechnic College,
Ettayapuram -628902.

**Mr.V.S. Raajesh Prabhu M.E.,**
HOD/CSE,
340, KLN Polytechnic College,
Madurai – 625009.

**Mr.P.Jeyasankar M.E.,**
HOD/CSE,
513, Latha Mathavan Polytechnic College,
Madurai – 625301.

**Validated by**
**Dr. S.MohammedMansoorRoomi M.E., Ph.D.,**
Assistant Professor / ECE,
5008, Thiagarajar College of Engineering,
Madurai – 625 015.

# 34033 - PROGRAMMING IN “C”
# DETAILED SYLLABUS

## UNIT- 1
## Program Development & Introduction to C

1.1 Program, Algorithm & flow chart:- Program development cycle- Programming language levels & features. Algorithm – Properties & classification of Algorithm, flow chart – symbols, importance & advantage of flow chart.

1.2 Introduction to C: - History of C – features of C- structure of C program – Compile, link & run a program. Diagrammatic representation of program execution process.

1.3 Variables, Constants & Data types:. C character set-Tokens-Constants- Key words – identifiers and Variables – Data types and storage – Data type Qualifiers – Declaration of Variables – Assigning values to variables- Declaring variables as constants- Declaring variables as volatile- Overflow & under flow of data.

## UNIT - 2
## C OPERATORS, I/O STATEMENT and DECISION MAKING

2.1 C operators:-Arithmetic, Logical, Assignment .Relational, Increment and Decrement, Conditional, Bitwise, Special Operator precedence and Associativity. C expressions – Arithmetic expressions – Evaluation of expressions- Type cast operator

2.2 I/O statements: Formatted input, formatted output, Unformatted I/O statements

2.3 Branching:- Introduction – Simple if statement – if –else – else-if ladder , nested if-else-Switch statement – go statement.

2.4 Looping statements:- While, do-while statements, for loop, break &continue statement.

## UNIT - 3
## ARRAYS and STRINGS FUNCTIONS

3.1 Arrays:- Declaration and initialization of One dimensional, Two dimensional and Character arrays – Accessing array elements – Programs using arrays.

3.2 Strings :- Declaration and initialization of string variables, Reading String, Writing Strings – String handling functions (strlen(),strcat(),strcmp()) – String manipulation programs.

3.3 Built –in functions: -Math functions – Console I/O functions – Standard I/O functions – Character Oriented functions.

3.4 User defined functions:- Defining functions & Needs-, Scope and Life time of Variables, , Function call, return values, Storage classes, Category of function – Recursion.

## UNIT - 4
## STRUCTURES AND UNIONS, DYNAMIC MEMORY MANAGEMENT

4.1 Structures and Unions:- Structure – Definition, initialization, arrays of structures, Arrays with in structures, structures within structures, Structures and functions – Unions – Structure of Union – Difference between Union and structure.

4.2 Dynamic Memory Management:- introduction – dynamic memory allocation – allocating a block memory (MALLOC) – allocating multiple blocks of memory (CALLOC) –releasing the used space: free – altering the size of a block (REALLOC).

## UNIT - 5

**“C” PROGRAMMING**

5.1 Program to find Sum of Series using “while” loop- Program to find Factorial of N numbers using functions- Program to swap the values of two variables.

5.2 Program to implement Ohms Law- Program to find Resonant Frequency of RLC Circuit- Program to find equivalent resistance of three resistances connected in series and parallel- Program to draw the symbol of NPN transistor using Graphics- Program to draw the symbol of diode using Graphics.

## CONTENTS

# Unit 1

## Program Development & Introduction to C

**Session Objectives:**

At the end of this session, the learner will be able to understand:

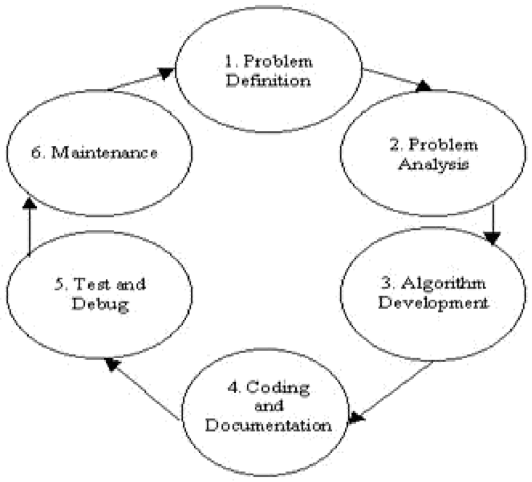
- Program development cycle - Programming language levels & features
- Algorithm- Properties & classification of Algorithm
- Flowchart symbols, importance & advantage of flowchart
- History of C – Features of C
- Structure of C Program
- Compile, link & run a program
- Diagrammatic representation of program execution process
- C character set – Tokens-constants-keywords – identifiers & variables
- Data types and storage – Data type qualifiers
- Declaration of variables – assigning values to variables – declaring variables as constants
- Declaring variables as volatile – overflow and underflow of data

### Program Development & Introduction to C 1.1 Program,Algorithm&flowchart:

#### Program development cycle

The program development cycle is a set of steps that are used to develop a program in any programming language. Generally, program development life cycle contains following 6 steps.

1. Problem Definition
2. Problem Analysis
3. Algorithm Development
4. Coding & Documentation
5. Testing & Debugging
6. Maintenance



Program Development Lifecycle

**1. Problem Definition**

The problem statement and the boundaries of the problem are decided in this stage.We need to understand the problem statement, requirement and the output of the problem solution.

**2. Problem Analysis**

We identify the requirements like variables, functions, etc. to solve the problem. We have to collect the required resources to solve the problem defined in the problem definition stage.

**3. Algorithm Development**

we should develop a step by step procedure to solve the problem using the specification given in the previous stage. It is important for program development.

**4. Coding & Documentation**

We may choose a suitable programming language to write programming instructions for the steps defined in the previous stage. In this stage, we develop actual program using programming languages like C, Java,Visual Basic etc.,.

**5. Testing & Debugging**

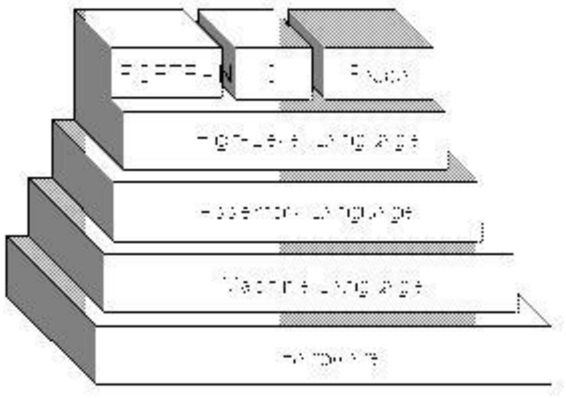
In this step, we verify whether the code written in previous step is solving the specified problem or not. We also test that whether it is providing the expected output or not.

**6. Maintenance**

In this stage, the program is used by the users. If any improvements required, all the phases are to be repeated again to make improvements. Apart from that, if the user encounters any problem ,then we need to repeat all the stages from the beginning.

**Programming language levels & features**

A **programming language** is a special language used by programmers to develop programs (sets of instructions) for computers to execute. It is classified into two types.



1. **Low Level Language**

- Low-level languages are designed to operate and handle the entire hardware and instructions set architecture of a computer directly.

- Low-level languages are considered to be closer to computers. In other words, their prime function is to operate, manage and manipulate the computing hardware and components. Programs and applications written in low-level language are directly executable on the computing hardware without any interpretation or translation.

  **Example**: Machine language and Assembly language.

**Features:**

- Low level languages are machine dependent
- Direct memory management
- Little-to-no abstraction from the hardware
- Register access
- Statements usually have an obvious correspondence with clock cycles
- Excellent performance

### 2. High Level Language

High level languages enable to write instructions using English words and mathematical symbols. Every instruction that the programmer writes in high level language is translated into machine language by using compiler and interpreter. **Example:** Fortran, C,Pascal, etc.,

**Features:**

- High level languages are machine independent.
- These are easy to learn and use because these are just like English language.
- It is easy to locate and correct errors.
- The programs written in high level languages are easier to maintain and modify.
- Writing programs in high level languages require less time or efforts.

## Algorithm

**Definition:** An algorithm is a set of instructions which is used to perform a particular task.

### Properties of an Algorithm

- **Finiteness:** - An algorithm terminates after a finite numbers of steps.
- **Definiteness:** - Each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted and can be performed without any confusion.
- **Input:-** An algorithm accepts zero or more inputs
- **Output:-** It produces at least one output.
- **Effectiveness:-** It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

### Classification of Algorithms

Most common types of algorithms are

1) Brute force
2) Divide and conquer
3) Decrease and conquer
4) Dynamic programming
5) Greedy algorithm
6) Transform and conquer
7) Backtracking algorithm

**Flow chart:**

A flowchart is a visual representation of the sequence of steps for solving a problem. It is a set of symbols that indicate various operations in the program. Once algorithm is written, its pictorial representation can be done using flowchart symbols. A flowchart gives a pictorial representation of an algorithm. The first flowchart was made by John Von Neumann in the year 1945.

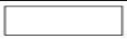
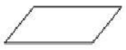
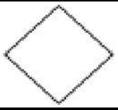
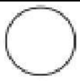
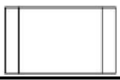
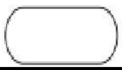
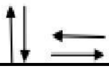
**Importance of a Flowchart:**

- It provides an overview of the program to be developed.
- It shows all elements and their relationships in the program.
- It helps to show the program flow quickly.
- It helps to check the program logic.
- It helps to write code in any language.

**Advantages of flow chart:**

- Flowchart is an important tool in the development of an algorithm itself.
- Easier to understand than a program itself.
- Independent of programming languages.
- Proper documentation
- Proper debugging
- Easy and clear presentation

**Flowchart symbols**: Some of the common symbols used in flowcharts are given below:

| **Symbol** | **Name** | **Function** |
|---|---|---|
|  | Process | Indicates any type of internal operation |
|  | Input/Output | Used for any input/output operation |
|  | Decision | Used to ask a question that can be replied in binary format(yes/no) |
|  | Connector | Used to join two parts of a program |
|  | Predefined process | Used to invoke a subprogram or an interrupt program |
|  | Terminal | Indicates the starting or ending of the program |
|  | Flow lines | Shows direction of flow |

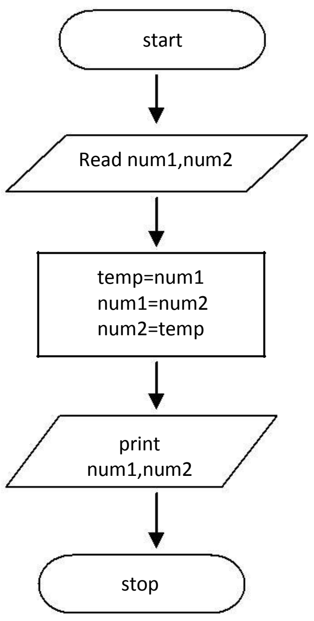
**Example:**

Write an algorithm and flow chart for swapping two numbers.

**Algorithm:**

```
Step1: start
Step2: Input num1, num2
Step3: temp=num1
Step4: num1=num2
Step5: num2=temp
Step6: Output num1, num2
Step7: stop
```

**Flowchart:**



**1.2 Introduction to**

**C <u>History of C:</u>**

C is a general purpose, high level language that was developed by Dennis M.Ritchie at Bell Labs in the year 1972. Later in 1978, Brian Kernighan and Dennis Ritchie made it available for the public usage.

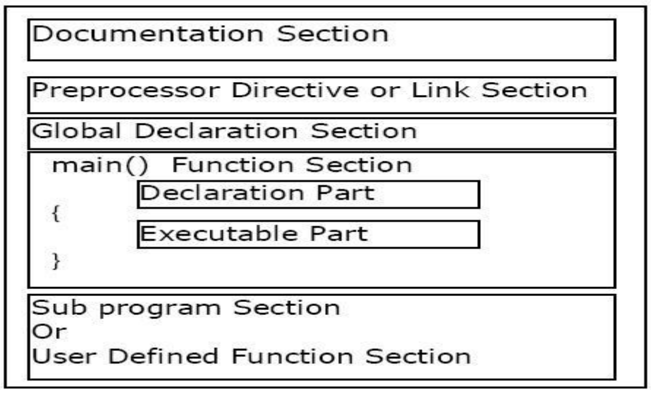
Unix operating system, Unix based applications and C compiler were written in C Language.

**Features of C**

- Easy to learn
- Structured programming language
- Efficient programs can be developed
- It can handle low level activities like assembly language program.
- It is portable. It can be compiled on a variety of computer platforms
- Programs are developed quickly by using built-in functions

**Structure of C Program**

The structure of a C program is given below.



**Sample Program**:

```
#include<stdio.h>
main()
{
        int a,b;
        clrscr();
        printf("Enter two
numbers");
        scanf("%d%d",&a,&b);
sum=a+b; printf("Sum=
%d",sum);
}
```

1. **Documentation section**:

   It consists of a set of comment lines which provides the name of the program,author and other details which the programmer would like to use during maintenance stage. The comments are enclosed in a C Program using /* and */

2. **Preprocessor directive or Link section**:

   It provides instruction to the compiler to link some functions or do some processing prior to the execution of the program. It is also used to define symbolic constants of the program.

   Header file used in the example is a standard input/output file(stdio.h). Programs must contain an #include line for each header file. #include <stdio.h>

3. **Global Declaration section**:

   There are some variables that are used in more than one function. Such variabls are called global variables and are declared in this section.

4. **Main()function section**:

   Every C Program must have one main() function. This section contains two parts,declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. All statements in the declaration part and executable parts must end with a semicolon; is a statement terminator.

5. **Sub program section**:

It contains all the user defined functions that are called in the main () function. User defined functions are generally placed after the main () function, although they may appear in any order.

**Compile,link and run a program:**

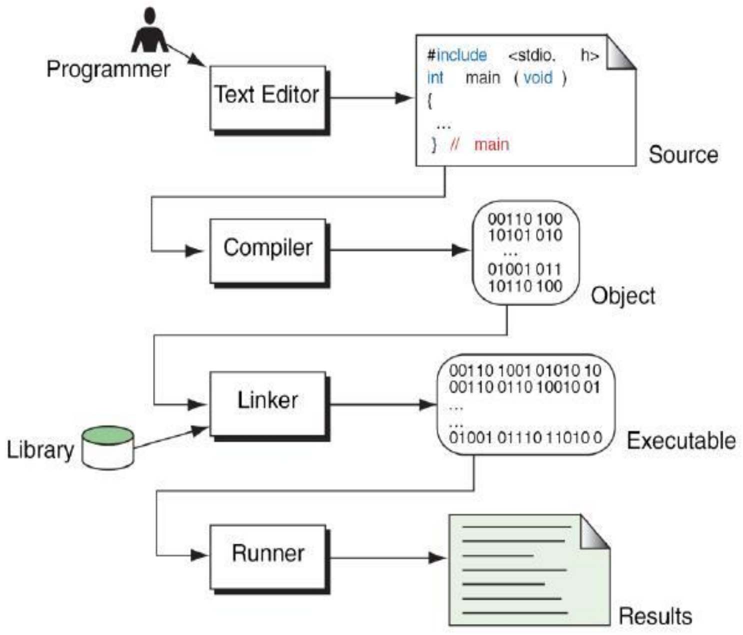
Important steps in creating and running C programs are given below.

1. Writing and editing the program
2. Compiling the program
3. Linking the program
4. Executing the program

**1.Writing and editing the program:**

A text editor helps us to type, edit and store character data. Every compiler comes with a text editor. Some of the features of editors are search, cut, copy, paste and format, etc.,

Once the program is completed, it is saved in a file on the disk. This file is called as "source file" which is given as input to the compiler. Important steps in developing a C program are shown in the following figure.



**2.Compiling the Program:**

The code in a source file on the disk must be translated into machine understandable language. A compiler will translate code in source file into machine language. The C compiler consists of two programs namely **(i)Preprocessor and (ii)Translator**.

The ***preprocessor*** reads the source code and prepares it for the compiler. It will scan for special instructions called as preprocessor commands. These commands inform the preprocessor to take code from libraries and make substitutions in the code. The outcome of preprocessing is called translation unit.

The ***translator*** reads the translation unit and writes resulting object module to a file that can be combined with other precompiled units to form the final program. An object module is the code in machine language. This module is not ready for execution because it does not have all the essential functions.

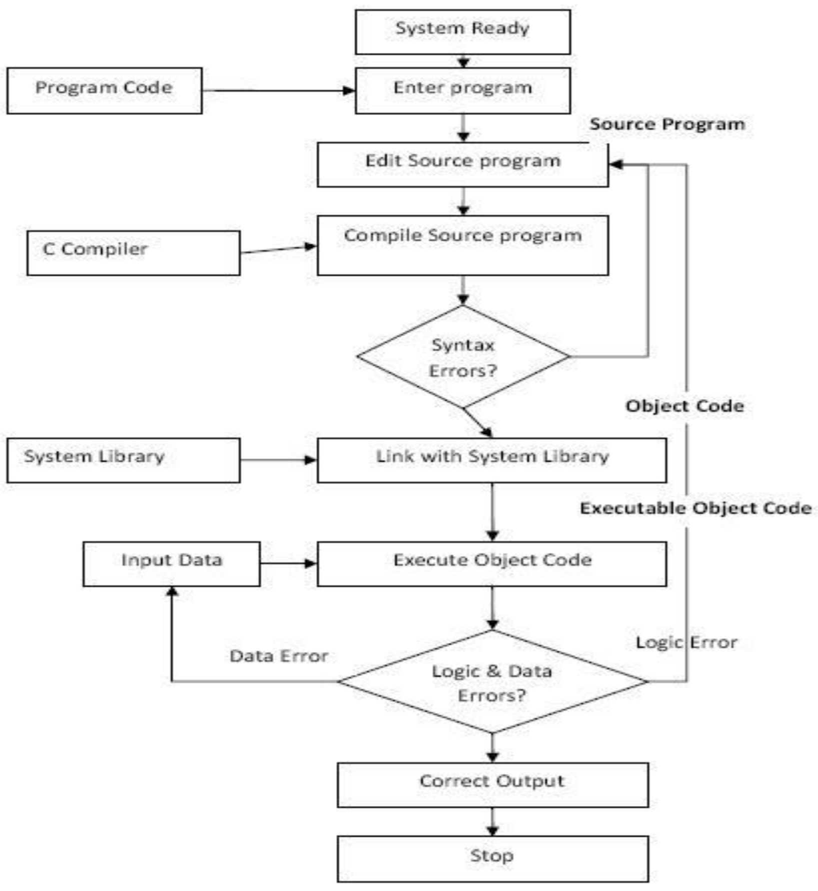
**3.Linking the Program**:

C Programs are made up of several functions like printf(),scanf(),..etc., The linker assembles all the essential functions to make our program into a final executable program.

**4.Executing the Program:**

After linking is over, program is ready for execution. To run a program, we use operating system command, such as run to load the program into main memory and execute it. Obtaining program into memory is the function of an operating system program called Loader. It locates the executable program and reads it into memory. During execution, the program reads data either from keyboard or from file. After the program processes the data, it prepares output. Output is redirected to monitor or to a file. Once the program execution is over, operating system removes the program from memory.

**Diagrammatic representation of program execution process:**



## 1.3 Variables, Constants and Data types

**C Character set**

Character set of C Language is given below

| | | |
|---|---|---|
| 1 | Alphabets | Uppercase A …. Z and lower case a….z |
| 2 | Digits | 0….9 |
| 3 | Special characters | , . ; : ? ' " ! \| \ / ~ _ $ % # & ^ * - + < > {}[]() |
| 4 | White spaces | Blank space, carriage return, horizontal tab, new line and form feed |

**Keywords**

Keywords are already defined and informed to C Compiler. So, it cannot be used as variable names. For example, char is a keyword. It informs the data type of a variable as character. It is also known as Reserve words. Some of the keywords are given below.

| | | |
|---|---|---|
| Auto | Extern | Size of |
| Break | Float | Static |
| Case | For | Struct |
| Char | Goto | Switch |
| Const | If | Typedef |
| Continue | Int | Union |
| Default | Long | Unsigned |
| Do | Register | Void |
| Double | Return | Volatile |
| Else | Short | While |
| Enum | Signed | |

**Identifiers:**

Identifiers are the names given by user to various program elements like variables, constants, functions and arrays.

**General rules :**

- Letters, digits and underscore can be used.
- Must start with an alphabet
- Underscore should not be used in the beginning of identifier
- Keywords cannot be used as identifiers
- Identifiers are case sensitive. For example, salary is different from SALARY

**Some examples of valid identifiers**:

hra, gross_salary,retval,name

**Some examples of invalid identifiers:**

Roll no : blank space not allowed
Emp-code : - hyphen not allowed
1name : first character must be an alphabet

**Constants:**

There are four basic types of constants in C. They are integer constants, floating point constants, character constants and string constants.

**Integer Constants:**

It is an integer valued number. It is classified into three different number systems. They are (i) decimal constants (ii)octal constants (iii)hexadecimal constants.

**General Rules for Integer constants:**

- Constant must have at least one digit
- Must not have decimal point
- Constant can be preceded by minus(-) sign or positive(+) sign
- Commas and blank spaces cannot be included within the constant
- Value of constant should not exceed the minimum and maximum limit. Range of integer constant is -32768 to 32767.

**Decimal Integer constants:**

A decimal integer constant consists of any combination of digits taken from the set 0 to 9.If the constant consists of two or more digits, the first digit must be other than 0.

**Valid decimal integer constants:**

0 1 650 32000 5555

**Invalid decimal integer constants:**

| | | |
|---|---|---|
| 070 | - | First digit cannot be zero |
| 40.3 | - | Illegal character (.) |
| 10,200 | - | Illegal character (,) |

**Octal Integer Constants**

An octal integer constant is a combination of digits taken from the set 0 to 7. In this case, first digit must be 0.

**Valid octal integer constants**

00 03 0431

**Ivalid octal integer constants**

| | | |
|---|---|---|
| 431 | - | First digit is not zero |
| 0831 | - | Invalid digit 8 |
| 03.41 | - | Invalid character. |

**Hexadecimal Integer constants:**

A hexadecimal integer is identified by ox or OX. Hexadecimal integer constant consists of digits taken from the set of 0 to 9 and letters taken from the set A to F (supports both upper and lower case). The letter a to f or A to F represent the decimal values from 10 to 15 i.e. a=10, b=11, c=12, d=13, e=14, f=15.

**Valid hexadecimal integer constants**:

0x0 0x2a 0x1ACF 0x6DDD

**Invalid hexadecimal integer constants:**

| | |
|---|---|
| 342a | First digit is not 0 or 0x |
| 0x14.4a | Invalid character. |
| 0x6bch | Invalid character h |

**Floating point constants**

It is also known as "real constants". The real constant is divided into two types namely fractional form and exponential form.

It is used to represent values like distance, height, temperature, price, etc.,

**Fractional form or Normal form:**

It consists of a number followed by a decimal point and the fractional part.

**Rules :**

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. Default sign is positive.
5. Commas or blanks are not allowed within a real constant

**Valid** Real constants (Fractional): 0.0 -0.1 +123.456 .2 2.

**Invalid** Real constant: -

| | | |
|---|---|---|
| 1 | - | a decimal point is missing |
| 1, 2.3 | - | Illegal character (,) |

**Exponential form or Scientific form:**

In exponential form, the real constant is represented in **two** parts.

**Mantissa** - The part appearing before **e,** the mantissa is either a real number expressed in decimal notation or an integer.

**Exponent** - The part following **e,** the exponent is an integer with an optional plus or minus sign followed by a series of digits. The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase.

**Example:** 0.000342 can be represented in exponential form as 3.42e-4

7500000000 can be represented in exponential form as 7.5e9 or 75E8

**Rules**

1. The mantissa part and the exponential part should be separated by letter E in exponential form
2. The mantissa part may have a positive or negative sign.
3. Default sign of mantissa part is positive.
4. The exponent part must have at least one digit, which must be a positive or negative integer. Default sign is positive.
5. Range of real constants expressed in exponential for is -3.4e38 to 3.4e38.

**Character Constants**:

It is classified into two types (i) Direct character constants (ii) Backslash character constants (i)**Direct character constants**:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
- The maximum length of a character constant is 1 character.

**Valid character constants**
'A'
'e'

**Invalid character constants**

| | |
|---|---|
| A | - It is not enclosed within single quotes |
| E | - It is not enclosed within single quotes |

(ii)**Backslash character constants**:

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

| Backslash character | Meaning |
|---|---|
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \" | Double quote |
| \' | Single quote |
| \\ | Backslash |
| \v | Vertical tab |
| \a | Alert or bell |
| \? | Question mark |

**String constant:**

String constants are the constants which are enclosed in a pair of double-quote marks. Note that a character constant 'A' and string constant "A" are not equal.

**Example:**

| | |
|---|---|
| "Welcome" | String constant |
| "A" | String constant having single character |
| "Hello world\n" | Prints string with newline |
| "" | Null string constant |

## Variables:

A variable is an identifier which is used to store values. Value of variable may change during program execution. Variables are assigned a particular memory location in memory unit where values can be stored.

**Rules :**

- It must begin with a letter
- It must consist of single letter or set of letters,digits or underscore letter
- It is case sensitive. For example, salary,SALARY and Salary are not same.
- Keywords are not allowed
- Blank space is not allowed
- Special characters other than underscore are not permitted

## Datatypes and Storage

C supports following categories of data types:

1. Basic data types
2. User defined data types
3. Derived data types

**Basic data types:**

C supports following fundamental data types.

- Character
- Integer
- Float
- Double

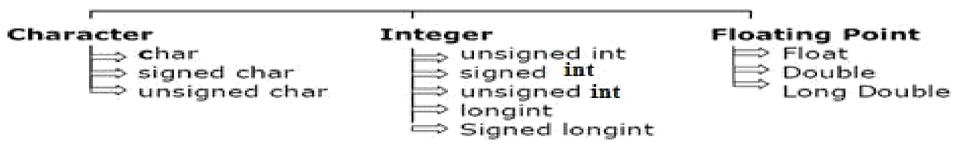
| **Data type** | **Description** | **Memory requirement** |
|---|---|---|
| Char | Single character | 1 byte |
| Int | Integer quantity | 2 bytes |
| Float | Floating point number | 4 bytes |
| Double | Double precision floating point number | 8 bytes |

**Data type Qualifiers**

It is used to change the meaning of basic data types.

List of qualifiers used in c:

- Signed
- Unsigned
- Long
- Short



**Data type qualifiers and their memory requirements**

| Data Types | Modifiers | Memory Requirement | |
|---|---|---|---|
| | | Bits | Byte |
| Character | i) Char | 8 | 1 |
| | ii) signed char | 8 | 1 |
| | iii) unsigned char | 8 | 1 |
| Integer | i) int | 16 | 2 |
| | ii) signed int | 16 | 2 |
| | iii) unsigned int | 16 | 2 |
| | iv) short int | 16 | 2 |
| | v) long int | 32 | 4 |
| | vi) signed long int | 32 | 4 |
| | vii) unsigned long int | 32 | 4 |
| Floating point | i) float | 32 | 4(6 digits ofprecision) |
| | ii) double | 64 | 8(10 digits of precision) |
| | iii) long double | 128 | 16(10 digits of precision) |

**Declaration of variables:**

All variables must be declared before they appear in executable statements. The general form of variable declaration and some examples are given below.

**Syntax:**

```
Data type  'variable name list';
```

**Example**

char name;

int age;

Variables of same data type can be declared in the following method.

**Example:**

int x,y,z;

float a,b,c;

**Assigning values to variables**:

Assignment operator "=" is used to assign values to variables.

**Syntax:**

```
Variable name=value;
```

**Example**:

Age=32

Salary=12500.50

**Initialization of variables**:

The process of giving initial value to the variable is known as initialization of a variable.

**Syntax:**

```
Data type variable name= initial
value;
```

**Example:**

int x=20;

float y=100.45;

**Declaring variables as constants:**

In certain situation, value of a particular variable should not change during program execution. It is possible by declaring the variable with const qualifier at the time of initialization.

**Syntax:**

```
Const data type constant
name=value;
```

**Example:**

Const int price=50;

**Declaring variables as volatile**:

Volatile keyword is a qualifier that is applied to a variable when it is declared. It informs the compiler that the value of the variable may change at any time without any action being taken by the code.

In practice, only three types of variables could change:

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables within a multi-threaded application

**Syntax**:

```
Volatile data type
variable;
```

**Example:**

Volatile int a;

**Overflow and underflow of data:**

This problem occurs when the value of a variable is either too large or too small for the data type. An overflow normally results in the largest possible real value. But an underflow results in zero. A good programmer should take care of choosing correct data types for handling the input/output values.

**Summary:**

In this session, we learnt about:

- Program development cycle - Programming language levels & features
- Algorithm- Properties & classification of Algorithm
- Flowchart symbols, importance & advantage of flowchart
- History of C – Features of C
- Structure of C Program
- Compile,link & run a program
- Diagrammatic representation of program execution process
- C character set – Tokens-constants-keywords – identifiers & variables
- Data types and storage – Data type qualifiers
- Declaration of variables – assigning values to variables – declaring variables as constants
- Declaring variables as volatile – overflow and underflow of data

# Unit -2

## C OPERATORS, I/O STATEMENT and DECISION MAKING

**Session Objectives:**

At the end of this session, the learner will be able to understand:

- C operators – Arithmetic,logical,Assignment,relational,increment&decrement,conditional,bitwise,special operators
- Operator precedence and associativity
- C expressions – arithmetic expressions – evaluation of expressions – type cast operator
- Formatted input&output,Unformatted input&output statements
- Branching statements – simple if – if..else..- else..if ladder – nested if..else – switch statement – goto statement
- Looping statements – while – do..while statements – for loop – break & continue statement

**C Operators,I/O Statement and Decision Making**

**2.1 C operators**: An operator is a symbol that informs the computer to perform a particular arithmetic or logical manipulations.

**Expression**: An expression is a sequence of operands and operators that reduces to single value. Example, 20+30 is an expression whose value is 50.

Different types of C operator are given below.

- Arithmetic
- Relational
- Logical
- Assignment
- Increment and Decrement
- Conditional
- Bitwise
- Special

**Arithmetic Operators**:
C provides all the basic arithmetic operators like + , - , * , / , %

**Integer Arithmetic**: When the operands in an expression are integers then the operation is known as Integer Arithmetic. It always results an integer value. Example, x=40, y=4

| Operator | Purpose | Example | Result |
|---|---|---|---|
| + | Addition | x+y | 44 |
| - | Subtraction | x-y | 36 |
| * | Multiplication | X*y | 160 |

|  |  |  |  |
|---|---|---|---|
| / | Division | x/y | 10 |
| % | Modulus (produces remainder of division) | x%y | 0 |

**Write a program to illustrate the use of all arithmetic operators**

```
 #include<stdio.h>
main()
{
        int add,sub,mul,div,mod,x,y;
        printf("Enter values of x,y: ");
        scanf("%d %d",&x,&y);
        add=x+y;
        printf("Result of addition : %d",add);
        sub=x-y;
        printf("Result of subtraction : %d",sub);
        mul=x*y;
        printf("Result of multiplication : %d",mul);
        div=x/y;
        printf("Result of division : %d",div);
        mod=x%y;
        printf("Result of modulus : %d",mod);
  }
```

**Relational Operator**: These operators can be used to compare arithmetic,logical and character expressions. The value of relational expression can be either one or zero.

The relational operators in C are

| **Operator** | **Meaning** | **Example** | **Result** |
|---|---|---|---|
| < | Less than | 20<40 | 1 |
| <= | Less than or equal to | 20<=20 | 1 |
| > | Greater than | 20>40 | 0 |
| >= | Greater than or equal to | 20>=20 | 1 |
| == | Equal to | 20==20 | 1 |
| != | Not equal to | 20!=20 | 0 |

**Logical Operator**: Logical operators are used when we want to test more than one condition and make decisions. The operands can be constants,variables and expressions. Example, x=1 and y=0

| **Operator** | **Meaning** | **Example** | **Result** |
|---|---|---|---|
| && | AND operator.<br>If both the operands are one,then result is | x&&y | 0 |

| | | | |
|---|---|---|---|
| | 1. | | |
| \|\| | OR operator. If any one of the operand is one,then result is 1. | x\|\|y | 1 |
| ! | NOT operator. If a condition is true,then NOT operator will make it 0. | !(x&&y) | 1 |

**Write a program to illustrate the use of relational and logical operators**

```
#include<stdio.h>
void main()
{
clrscr();
printf("15>5&&15<20 : %3d",15>5&&15<20);
printf("15<5||15==15  :%3d",15<5||15==15);
printf("!(15==15) :%3d",!(15==15))";
}
```

**Result:**

```
15>5&&15<20     :     1
15<5||15==15    :     1
!(15==15) : 0
```

**Assignment Operator:** It is used to assign the results of an expression into a variable. "=" is the assignment operator.

**Short hand assignment operator:**
C provides "short hand" assignment operators in the following form

```
Variable operator = Expression
```

| Short hand Operator | Meaning | Example |
|---|---|---|
| += | It will add both left operand and right side value and result is stored in left operand | x+=1 |
| -= | It will subtract right side value from left operand and result is stored in left operand | x-=1 |
| *= | It will multiply both left operand and right side expression and result is stored in left operand | x*=n+1 |
| /= | It will divide left operand by right side expression and result is stored in left operand | x/=n+1 |
| %= | It will divide left operand by right side expression and remainder is stored in left operand | x%=n+1 |

**Increment and Decrement operators:**

Increment operator (++) adds 1 to its operand and decrement operator (--) subtracts one from its operand.

For example, Y=y+1
is written as Y=y++

Different types of increment and decrement operators:

1. **Prefix increment(++i)** In this case, first increment and then do operation.
2. **Postfix increment(i++)** In this case, first do the operation and then increment.
3. **Prefix decrement(--i)** In this case, first decrement and then do operation.
4. **Postfix decrement(i--)** In this case, first do the operation and then decrement.

**Example: Prefix increment**

```
int i=1;
printf("i=%d \n",i);
printf("i=%d \n",++i);
printf("i=%d \n",i);
```

**Result:**

```
i=1
i=2
i=2
```

**Example: Postfix increment**

```
int i=1;
printf("i=%d \n",i);
printf("i=%d \n",i++);
printf("i=%d \n",i);
```

**Result:**

```
i=1
i=1
i=2
```

**Conditional Operator:**

C has special operator called ternary or conditional operator. If the condition is true then value1 is assigned to the variable, otherwise value2.

**Syntax:**

```
Variable=(condition)?value1:value2
```

**Example :**

```
large=(x>y)?x:y;
```

Above statement is equal to following code:

```
If(x>y)
large=x;
else
large=y;
```

## Bitwise Operators

The Bitwise operators are used to perform bit operations. All the decimal values will be converted into binary values (sequence of bits i.e. 0100, 1100, 1000, 1001 etc) and bitwise operator will work on these bits such as shifting them left to right or converting bit value from 0 to 1 etc. Following table shows the different Bitwise operators in C.

For example, assume x=6 and y=8 and their values in binary form are,

x=00000110

y=00001000

| Bitwise Operators | Meaning | Example |
|---|---|---|

| | | |
|---|---|---|
| & | Bitwise AND | x&y=00000000 |
| \| | Bitwise OR | x\|y=00001110 |
| ^ | Bitwise exclusive OR | x^y=00001110 |
| ~ | Bitwise complement | ~x=00001001<br><br>(Bitwise Not operator will convert all 0 into 1) |
| << | Shift Left | x<<1=00001100<br><br>(Bits will move 1 step left. If we use 2 or 3 then bits shift accordingly |
| >> | Shift right | y>>1=00000100<br><br>(Bits will move 1 step right. If we use 2 or 3 then bits shift accordingly |

**Truth Table of Bitwise Operators**

| **X** | **y** | **x&y** | **x\|y** | **x^y** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Example:** Assume variable a contains 60 and b contains 13

```
a = 60
b = 13
---------------------------------
a in Binary : 0011 1100
b in Binary : 0000 1101
---------------------------------
a & b        : 0000 1100
a | b        : 0011 1101
a ^ b        : 0011 0001
~a           : 1100 0011
---------------------------------
```

**<u>Rules from above table :</u>**

1. If Two bits are same Then Resultant XOR is 0 .
2. If Two bits are different Then Resultant XOR is 1.
3. If any of the bit is 1 then Resultant OR is 1
4. If both bits are 0 then Resultant OR is 0
5. If any of the bit is 0 then Resultant AND is 0.

**Write a program to understand the bitwise operators available in C**

```
#include<stdio.h>
main()
{
Unsigned int a=60;   /*60 = 0011 1100
*/
Unsigned int b=13;   /* 13=0000 1101 */
int c=0;
c=a&b;                          /*12=0000
1100*/
printf("Value of a&b  is %d\n",c);
c=a|b;                    /*61=0011 1101*/
 printf("Value of a|b is %d\n",c);
c=a^b;                          /*49=0011
0001*/
printf("value of a^b is %d\n",c);
c=~a;                     /*-61=1100 0011*/
printf("value of ~a is %d\n",c);
}
```

```
Result:
Value of a&b is 12
Value of a|b  is  61
Value of  a^b is    49
Value of  ~a is     -61
```

**Bitwise Left Shift Operator in C**

1. It is denoted by **<<**
2. Bit Pattern of the data can be **shifted by specified number of Positions to Left**
3. When Data is Shifted Left , trailing zero's are **filled with zero**.

**Overview of Left Shift Operator**

| | |
|---|---|
| **Original Number A** | 0011 1100 |
| **Left Shift** | 1111 00$\boxed{00}$ |
| **Trailing Zero's** | Replaced by 0 |
| **Direction of Movement of Data** | **<<<<<=======Left** |

**Syntax : Bitwise Left Shift Operator**

```
[variable]<<[number of places]
```

**Example** : Bitwise Operator [Left Shift Operator]

```
#include<stdio.h>
int main()
{
int a = 60;
printf("\nNumber is Shifted By 1 Bit : %d",a <<
1);
printf("\nNumber is Shifted By 2 Bits : %d",a <<
2);
printf("\nNumber is Shifted By 3 Bits : %d",a <<
3);
return(0);
}
```

**Output :**

```
Number is Shifted By 1 Bit : 120
Number is Shifted By 2 Bits : 240
Number is Shifted By 3 Bits : 480
```

**Bitwise Right Shift Operator in C**

1. It is denoted by **>>**
2. Bit Pattern of the data can be **shifted by specified number of Positions to Right**
3. When Data is Shifted Right , leading zero's are **filled with zero**.

**Overview of Right Shift Operator**

| | |
|---|---|
| **Original Number A** | 0011 1100 |
| **Right Shift by 2** | 0000 1111 |
| **Leading 2 Blanks** | Replaced by 0 |
| **Direction of Movement of Data** | **Right ========>>>>>>** |

**Syntax : Bitwise Right Shift Operator**

```
[variable]>>[number of places]
```

**Example :** Bitwise Operator [Right Shift Operator]

```
#include<stdio.h>
int main()
{
int a = 60;
printf("\nNumber is Shifted By 1 Bit : %d",a >> 1);
printf("\nNumber is Shifted By 2 Bits : %d",a >> 2);
printf("\nNumber is Shifted By 3 Bits : %d",a >> 3);
return(0);
}
```

**Output :**

```
Number is Shifted By 1 Bit : 30
Number is Shifted By 2 Bits : 15
Number is Shifted By 3 Bits : 7
```

**Special operators:**

| **Operator** | **Meaning** | **Example** |
|---|---|---|
| Sizeof() | It will return the size of a variable | Sizeof(x) |
| & | It will return the address of a variable | &x |
| * | It will be a pointer to a variable | *x |
| ?: | It is conditional operation. | large=(x>y)?x:y; |
| , | Comma operator is used to link the | z=(x=20,y=25,x+y) |

| | | |
|---|---|---|
| | related expressions together. | It will assign 20 to x then assign 25 to y and finally assigns 45(i.e.20+25) to variable z |

**Operator precedence and Associativity**

In mathematics we are given with an expression.

(2 * 3 + 2) - 6 / 2

We know that in maths we use the formula BODMAS - Brackets Of Division or Multiplication and Addition or Subtraction. First thing inside brackets which is 2 * 3 + 2 => 6 + 2 => 8. Expression is now simplified to 8 - 6 / 2 => 8 - 3 => 5. This is called **Operator precedence**.

Take another example
8 - 2 + 1
Here subtraction and addition have same priority in such case evaluated from left to right. 8 - 2 + 1 => 6 + 1 => 7. This is called **Associativity**.

**List of C operators with precedence and associativity**

| Operator | Description | Associativity |
|---|---|---|
| ++ -- | post-increment, post-decrement | left to right |
| ++ --<br>+ -<br>!~<br>&<br>sizeof | pre-increment, pre-decrement<br>Unary + and Unary -<br>Logical not and bit-wise not<br>Address operator<br>Size of variable/expression | right to left |
| * / % | Multiplication, division and remainder | left to right |
| + - | Addition subtraction | left to right |
| << >> | bit-wise left shift, bit-wise right shift | left to right |
| < <=<br>> >= | Less than or less than or equal to<br>greater than or greater than or equal to | left to right |
| == != | Equal, Not Equal | left to right |
| & | Bit-wise AND | left to right |
| ^ | Bit-wise XOR | left to right |
| \| | Bit-wise OR | left to right |
| && | Logical AND | left to right |
| \|\| | Logical OR | left to right |
| ?: | Conditional operator | right to left |
| = | Assignment, Other Assignment short cut | right to left |
| , | Comma | left to right |

**Program to understand operator precedence and associativity**

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

```
int main()
 {
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;
    e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   printf("Value of (a + b) * c / d is : %d\n", e );
   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   printf("Value of ((a + b) * c) / d is : %d\n" ,   e );
   e = (a + b) * (c / d);   // (30) * (15/5)
   printf("Value of (a + b) * (c / d) is : %d\n",   e );
   e = a + (b * c) / d;     // 20 + (150/5)
   printf("Value of a + (b * c) / d is : %d\n" , e );
    return 0;
  }
```

**Result:**

```
Value of (a + b) * c / d is      : 90
Value of ((a + b) * c) / d is    : 90
Value of (a + b) * (c / d) is    : 90
Value of a + (b * c) / d is       : 50
```

**C Expressions:**

An expression is a combination of constants,variables and operators. There are three types of expressions available in C. They are

1. Integer expression
2. Real expression
3. Mixed mode expression

**1.Integer expression:** When both the operands in an expression are integers, it is called as Integer expression. It always produces an integer value. Let us assume, a=20 and b=5 , then we obtain the following results for the different integer expressions.

| | |
|---|---|
| a+b | result is 25 |
| a-b | result is 15 |
| a*b | result is 100 |
| a/b | result is 4 |
| a%b | result is 0 |

**2.Real expression**: An arithmetic expression contains only real(float) operands is called as real expression. Let us assume, a=10.0 and b=4.0, then we obtain the following results for the different real expressions.

| | |
|---|---|
| a+b | result is 14.0 |
| a-b | result is 6.0 |

```
a*b      result is  40.0
a/b      result is  2.5
```

**3.Mixed mode expression:** If one of the operand is real and the other is integer then expression is called as Mixed mode expression. In this case, only the real operation is performed and the result is always a real number.

**Example:**

20/3.0 result is 6.6

**Type Casting**

Changing of one data type to another data type is called type casting. Type cast operator is used to convert one data type to another data type. There are two types of conversion available. They are

1. Implicit conversion
2. Explicit conversion

**1.Implicit conversion:**

If data type are mixed in an expression, then "C" language performs the conversion automatically. It is called as Implicit conversion.

**Example:**

```
float x;
x=20+3.5;
```

During execution, integer value 20 is automatically converted into float data type. So, the result 23.5 will be kept in variable x.

**2.Explicit conversion:**

One data type is converted into another data type explicitly with the help of type cast operator. It is called as explicit conversion.

**Syntax:**

```
(data type) variable;
```

**Example:**

```
float x,y;
int sum;
x=5.25;
y=10.21;
sum=(int)x+(int)y;
printf("sum is : %d",sum);
```

Output of above code is 15, because variables x and y are converted as integer during execution.

**2.2 I/O Statements** All the input and output functions are defined in the "**stdio.h" header file.**

Types of input/ouput statements are given below

- Formatted input/output statements
- Unformatted input/output statements

Formatted Input Statements

**Scanf() function:** The scanf() function is to used to input values for variables of numeric, string and character data types.

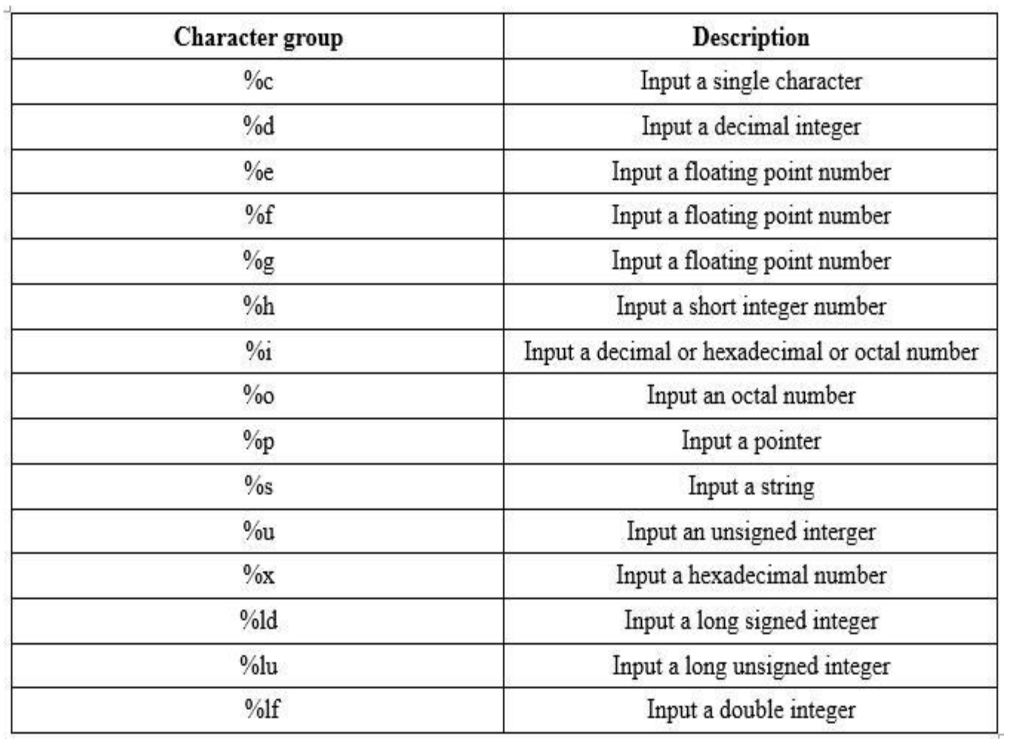
**Syntax:**

```
scanf("string of control", address);
```

**string of control**: Sequence of character groups is entered here. This is usually a combination of one or more conversion characters following % symbol and this describes types of values which are to be specified to the variables.

**address:** This is a list of addresses which represents the address of memory locations where the values we specify for input is stored.

**Table of conversion specifier characters**



| Character group | Description |
|---|---|
| %c | Input a single character |
| %d | Input a decimal integer |
| %e | Input a floating point number |
| %f | Input a floating point number |
| %g | Input a floating point number |
| %h | Input a short integer number |
| %i | Input a decimal or hexadecimal or octal number |
| %o | Input an octal number |
| %p | Input a pointer |
| %s | Input a string |
| %u | Input an unsigned interger |
| %x | Input a hexadecimal number |
| %ld | Input a long signed integer |
| %lu | Input a long unsigned integer |
| %lf | Input a double integer |

**Examples of formatted input functions**

1)**Integer Input :** We input decimal integer data using %d character.

**Example**: scanf("%d", &number1);

**Explanation**: whatever value you input from standard input, e.g. 10, will be stored in a memory location called "number1".

**Example:** scanf("%4d", &number2);

**Explanation:** In the above example, %d is the conversion character and number 4 is known as field width. The number of characters in data value should not exceed the specified field width. We can input any four digit number e.g. 1234, 5678, etc.

2)**Character Input :** We input character data using %c conversion character. The string constant requires %s conversion character.

**Exampe:** scanf("%c", &section) ;

**Explanation:** In the above example, section is a variable of type character. If we input A then corresponding ASCII value associated with this symbol is stored in the memory location and in this example it is 65 as ASCII value corresponding to A is 65.

**Example:**

char name[5];

scanf("%5c", name);

**Explanation:** In the above example, we do not give an ampersand symbol. Here we specify the total number of characters present in the input. we also need to declare a variable name "name" as one dimensional character array.

**Example:**

Char myname[15];

scanf("%s", myname) ;

**Explanation:** In the above example, we describe the way to input string constant. We have to declare myname as a one dimensional character array.

3)**Floating point Input :** We input floating point values by using %f conversion character and in contrast to integer type we usually do not specify the field width for real numbers.

**Example**: scanf("%f", &rate);

**Explanation:** In the above example, if we input 65 then value 65.0 value is stored in the memory location specified by rate.

**Example:** scanf("%lf", &n);

**Explanation:** In the above example, we try to input a double precision number. Here the data type type is double so we use %lf instead of %f.

<u>Formatted Output Function</u> Formatted output function is printf()function. This function displays data on statndard output and that is monitor.

**Syntax:**

| **printf("string of control",** |
|---|

```
list_of_variables);
```

**string of control:** This specifies the data type and format of values which are going to be displayed on standard output

**list_of_variables**: This specifies list of variables associated with values meant to be displayed on monitor

**Examples of formatted output functions**

**1)Integer type:** We use %d conversion character to display the integer type data on standard output. It is similar to formatted input but scanf is replaced by printf and there is no (&) ampersand symbol.

**Example:** printf("%d", number1);

**Explanation:** In the above example, variable number1 is to be displayed on standard output. For instance its value is 10 then number 10 is displayed on monitor as output.

**2)Character type:** We need the same printf function to display a character or a string.

**Example:** printf("%c", section);

**Explanation:** In the above example, character variable is of char type data. For example if we enter A as input then we will see A as output after executing the above statement.

**Example:**

char myname[15];

printf("%s", myname);

**Explanation:** In the above example, we will have a string as ouput which is represented by variable name of type string. In this scenario we will have to declare this variable name before using it say for instance

**3)Floating point type:** Floating point number is displayed either in decimal form or scientific notation. Usually we use %f as conversion character.

**Example:** printf("%f", z);

**Explanation:** In the above example, if variable z stores 25 then 25.000000 will be displayed on screen.

**Example:** printf("%7.2f",z);

**Explanation:** We can control the number of digits by the usage of field width specifier. Above statement is used to print maximum of 7 characters which includes decimal point( two digits after decimal point). For example, if we store 243.1234 in the variable z then it will print 243.12 as output .

**Program to explain formatted input output functions**

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

void main()
{
  int num_1, num_2, num_3, s_um;
  float avg;

   printf("Enter three numbers\n");
   scanf("%d%d%d", &num_1, num_2, num_3,
s_um;
   s_um=num_1+num_2+num_3;
   avg=s_um/3.0;
   printf("Sum=%d, Average=%f", s_um, avg);
   getch();
}
```

Result:

```
Enter three numbers
25 12 67


Sum=104 , Average=34.666668
```

Unformatted Input Function

These functions read character type data from keyboard and we have two functions namely:

- getchar()
- gets()

And both of these functions are included in stdio.h header file.

**1. The getchar() function:** This function is responsible for reading only one character from standard input. We do not need parameter within the brackets.

**Format:** character=getchar();

**Explanation:** Here,**character** is a char type of data variable and it accepts the character assigned.

**2. The gets() function**: This function is responsible for accepting input from keyboard and continues to accept input until enter key or return key is pressed. Whatever we type is stored as string.

**Format:** gets(str);

**Explanation:** Here,**str** represents sequence of characters which is of char data type

Unformatted Output Function

These functions will print vales of character data type on monitor. We have two functions and they are also stored in stdio.h header file:

- putchar()
- puts()

**1. The putchar() function:** This function is responsible for printing only one character on monitor and the supported data type is char.

**Format**: putchar(character);

**Explanation:** Here,**character** is a char data type variable

**2. The puts() function:** This function prints sequence of characters on monitor. The end of string is newline character and it is not displayed on monitor.

**Format**: puts(str);

**Explanation:** Here,**str** represents string of characters

**Program to explain unformatted input output functions**

**Sample program1**

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

void main()
{
 char l;
 l=getchar();
 putchar(l);
 getch();
}
```

```
Result

A
A
```

**Sample program2**

```
include < stdio.h >
main ()
{
char a[50];
clrscr();
printf("Enter the String:");
gets(a);
printf("\n Entered String:");
puts(a);
}
```

**Result**

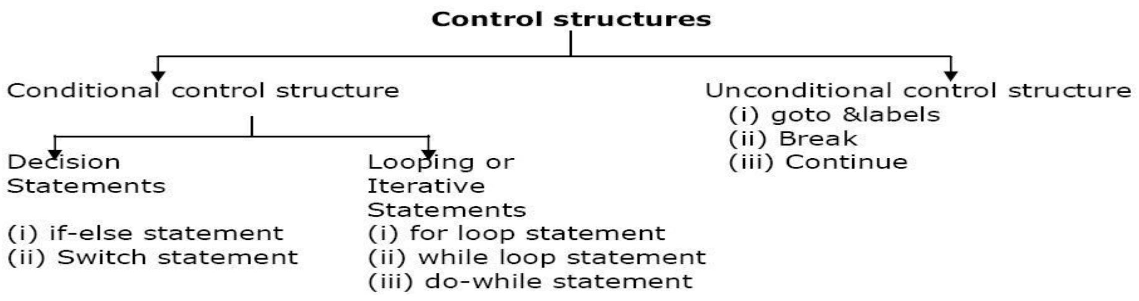
```
Enter the String : hello
Entered String  : hello
```

## 2.3 Branching

### Introduction

A C program requires a logical test to be performed at some point in the program. Based on the outcome of logical test, one of several possible actions can be performed. It is called Branching.

The branching structure which controls the program flow is called control structure. The classification of control structure is shown below.



### (i)Simple if statement

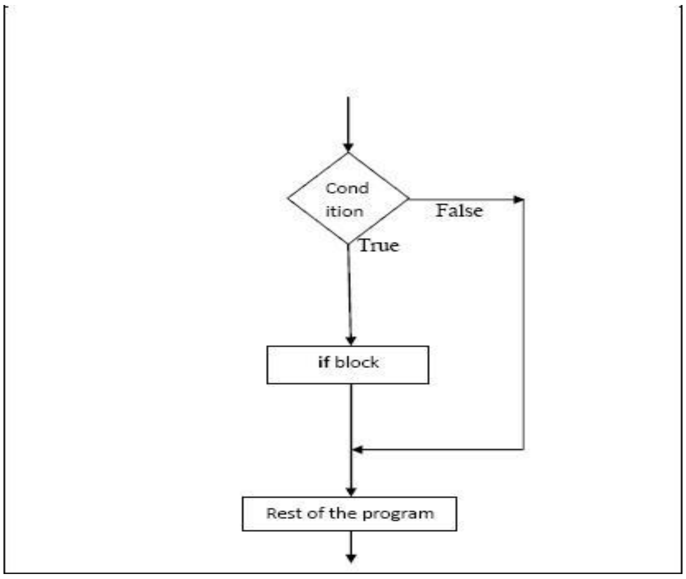
The if statement is a powerful decision making statement and is used to control the flow of execution of statements.

**Syntax:**

```
If (condition)

{

Statement;

}

Rest of the code
```

- It has only one condition. The statement is executed only when the condition is true. If the condition is false, the compiler skips the lines within "if block". The condition is always enclosed within a pair of parenthesis ( ). The statements following "if" statement are enclosed in curly braces { }.

**Flow chart for "if" statement**



**Write a program to check equivalence of two numbers. Use "if" statement.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x,y;
clrscr();
printf("\n Enter two numbers: ");
scanf("%d %d",&x,&y);
if((x-y)==0)
printf("\n Both numbers are
equal");
getch();
}
```

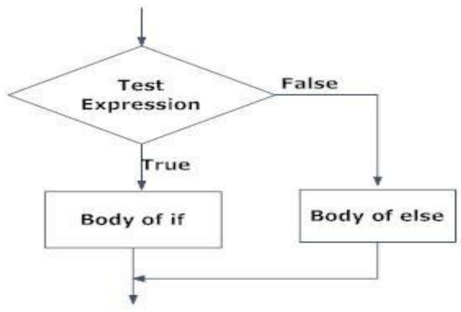
**(ii)if..else statement**

It is observed that the if statement executes only when the condition following if is true. It does nothing when the condition is false. In if..else statement either "true block" or "false block" will be executed and not both. The else statement cannot be used without "if".

**Syntax:**

```
If(condition)
{
Statements;              /* body of if */
}
else
{
Statements;               /* body of else */
}
```

**Flowchart for if..else statement**



**Write a program to print the given number is even or odd**

```
#include<stdio.h>
#include<conio.h>
main()
{
int x;
clrscr();
printf("Enter a number: ");
scanf("%d",&x);
if((x%2)==0)
printf("\n The given number is
EVEN");
else
printf("\n The given number is
ODD");
getch();
}
```

**Result:**

```
Enter a number: 30

The given number is EVEN

Enter a number:15

The given number is ODD
```

**(iii)Nested “if..else” statement:**

Using of one if..else statement in another if..else statement is called as nested if..else statement. When a series of decisions are involved, we may have to use more than one if..else statement in nested form.

**Syntax:**

```
if(condition 1)
{
if(condition 2)
{
Statement 1;
}
else
{
Statement 2;
}
}
else
{
if(condition 3)
{
Statement 3;
}
else
{
Statement 4;
}
}        /*end of outer if..else*/
```

- If condition 1 is true then it checks condition 2 if it is true then statement 1 is executed otherwise statement2 is executed.
- If condition1 is false then it goes outer else block and it checks condition3, if it is true then statement 3 is executed, otherwise statement4 is executed.

**Example: Program to print the largest of three float numbers using nested if ..else statements.**

```
#include<stdio.h>
#include<conio.h>
main()
{
float x,y,z;
printf("Enter three numbers:");
scanf("%f%f%f",&x,&y,&z);
printf("\n Largest value is:");
if(x>y)
{
        if(x>z)
        printf("%f",x);
else
        printf("%f",z);
}
else
{
        if(y>z)
        printf("%f",y);
        else
        printf("%f",z);
}
getch();
}
```

**Result:**

```
Enter three numbers: 10.20 14.32
3.17


  Largest value is: 14.32
```

(i)**The else..if ladder:** It is another method of putting if's together when multiple decisions are involved. A multipath decision is a chain of if's in which the statement associated with each else is an if. Hence, it forms a ladder called else..if ladder.

**Syntax:**

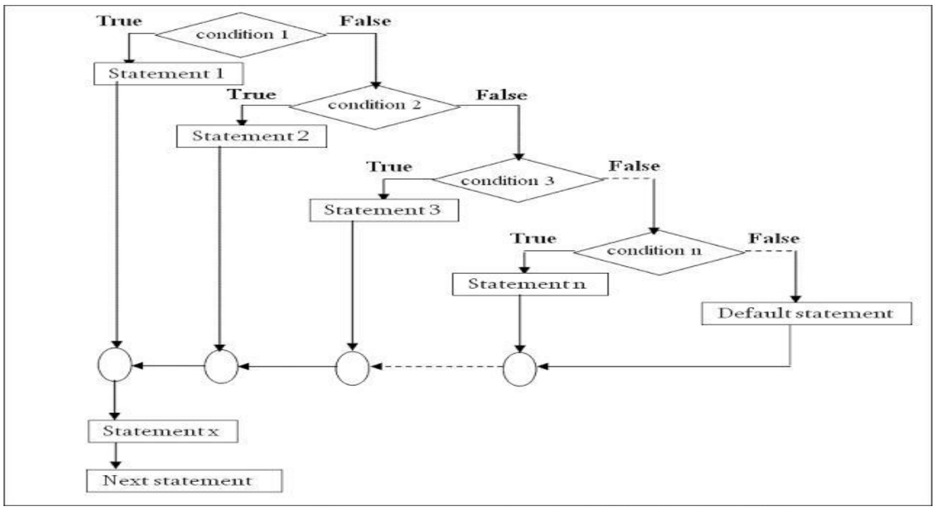
```
if(condition1)
Statement 1
else if (condition2)
Statement 2
else if(condition 3)
Statement 3
…
…
else if(condition n)
Statement n; else

Default statement;

Rest of the program statement –X;
```

Above construction is called as else if ladders. The conditions are evaluated from top to bottom.

As soon as a true condition is met, the statement associated with it is executed and the control is transferred to the rest of the program statement X.

When all the "n" conditions fails, then final else containing the default statement will be executed.

**Flowchart else..if ladder:**



**Write a program to find the largest among three numbers by using "else..if" ladder.**

```
#include<stdio.h>
#include<conio.h>
main()
{
int x,y,z;
clrscr();
printf("Enter first number: ");
scanf("%d",&x);
printf("Enter second number: ");
scanf("%d",&y);
printf("Enter third number: ");
scanf("%d",&z);
if((x>y)&&(x>z))
        printf("Largest number is: %d",x);
else if((y>z))
printf("Largest number is: %d",y);
else
printf("Largest number
is:%d",z); getch();
}
```

**Result:**

```
Enter first number: 50
Enter second number:100
Enter third number:150

Largest number is: 150
```

(v)**Switch case statement**

The switch statement causes a particular group of statements to be chosen from several available groups.The selection is based upon the current value of an expression which is included within the switch statement. It is a multiway branch statement.

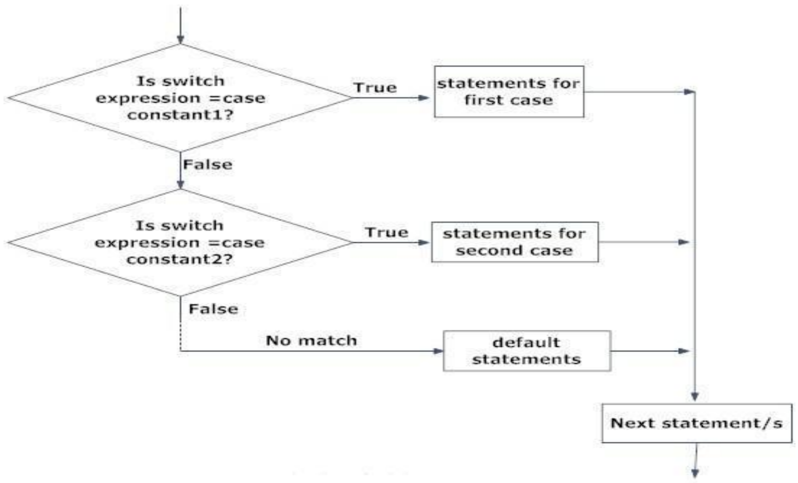
The switch statement requires only one argument of int or char data type,which is checked with number of case options. The switch statement evaluates expression and then looks for its value among the case constants. If the value matches with case constant,then that particular case statement is executed,otherwise default statement is executed.

Every case statement terminates with colon : and each case block should end with break statement.

**Syntax:**

```
switch(variable or expression)

{

case constant value1: statement 1;

                                break;

case constant value2: statement2;

                                 break;

case constant value n: statement n;

                                 break;

default : default statement;


}
```

**Flowchart:**


Is switch expression =case constant1?
True
statements for first case
False
Is switch expression =case constant2?
True
statements for second case
False
No match
default statements
Next statement/s

Program to provide functions 1.Addition 2.Subtraction 3.Multiplication 4.Division 5.Exit using switch statement

```
#include<stdio.h>
#include<conio.h>
main()
{
int x,y,z,ch;
clrscr();
printf("\n \t 1.Addition");
printf("\n\t 2.Subtraction");
printf("\n \t 3.Multiplication");
printf("\n\t 4.Division");
printf("\n \t 5.Exit");
printf("\n\n\t Enter your Choice");
scanf("%d",&ch);
if(ch<=4 && ch>=1)
{
printf("Enter two numbers: ");
scanf("%d %d",&x,&y);
}
switch(ch)
{
Case 1: z=x+y;
        Printf("\n Addition: %d",z);
        Break;
Case 2: z=x-y;
        Printf("\n Subtraction: %d",z);
        Break;
Case 3: z=x*y;
        Printf("\n Multiplication: %d",z);
        Break;
Case 4: z=x/y;
        Printf("\n Division: %d",z);
        Break;
Case 5: exit();
        Break;
Default: printf("\n Wrong choice");
}
getch();
}
```

```
Result:

1.Addition

2.Subtraction

3.Multiplication

4.Division

5.Exit

Enter your choice: 3

Enter two numbers 5 9

Multiplication 45
```

## 2.4 Looping statements

A Loop is defined as a block of statements which are repeatedly executed for certain number of times.

### While loop

It is the simplest of all the looping structures in C.

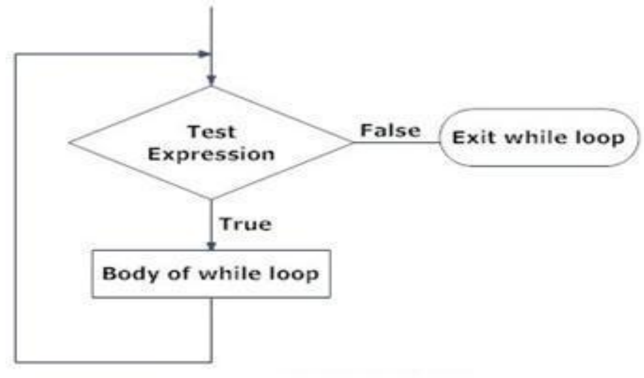
**Syntax:**

```
Initialization expression;

while(condition)

{

        Body of the loop

        Updation expression

}
```

The while is an entry – controlled loop statement. The condition is evaluated first and if the condition is true, then the body of the loop is executed. The execution process is repeated until the test condition becomes false and the control is transferred out of the loop.

On exit, the program continues with the statement immediately after the body of the loop.

**Flowchart of while loop**



**Program to add 10 consecutive numbers starting from 1. Use the while loop.**

```
#include<stdio.h>
#include<conio.h>
main()
{
int a=1,sum=0;
clrscr();
while(a<=10)
{
        sum=sum+a;
        a++;
}
printf("sum of 1 to 10 numbers is : %d", sum);
getch();
}
```

**Result:**

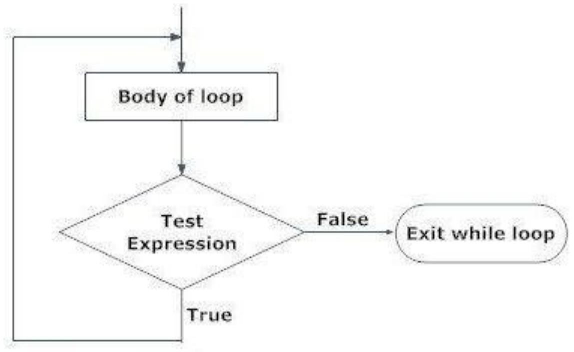
```
Sum of 10 numbers is : 55
```

**Do..while Loop** In do .. while loop, the condition is checked at the end of the loop. So,it will execute at least one time even if the condition is false initially. The do..While loop executes until the condition becomes false.

**Syntax:**

```
Initialization expression

do

{ Body of the loop

        Updation expression;

}while(condition);
```

**Flowchart of do..while loop**



**Program to add 10 consecutive numbers starting from 1. Use the do.. while loop.**

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

{

int a=1,sum=0;

clrscr();

do

{

        Sum=sum+a;

        a++;
} while(a<=10);
printf("sum of 1 to 10 numbers is : %d", sum);
getch();
}
```

```
Result: Sum of 10 numbers is:
55
```

**<u>For loop</u> The** for loop statement comprises of 3 actions. The 3 actions are "initialize expression", "Condition expression" and "Updation expression".

- Expressions are separated by semi-colon ;
- Loop variable should be assigned with a initial and final value
- Each time the updated value is checked by the loop itself
- Increment/decrement is the numerical value added or subtracted to the variable in each round of the loop

**<u>Syntax:</u>**

```
for(initialize expression;test
condition;updation)
{
Statement 1;
Statement 2;
………
Statement n;
}
```

- The initialization sets a loop to an initial value. This statement is executed only once.
- The test condition is a relational expression that determines the number of iterations desired or it decides when to exit from the loop

1. The for loop continues to run as long as conditional test is satisfied.
2. When the condition becomes false the control of the program exits from the body of for loop and executes rest of the code after the body of the loop.

- The updation(increment or decrement operations)decides how to make changes in the loop

**<u>Different methods of using For Loop</u>**

| **Syntax** | **Output** | **Description** |
|---|---|---|
| For (; ;) | Infinite loop | Arguments not required |
| For(x=0;x<=10;) | Infinite loop | "x" is neither increased nor ecreased |
| For(x=0;x<=10;x++) Printf("%d",x) | Displays value from 0 to 10 | "x" is increased from 0 to 10 |
| For(x=10;x<=0;x--) Printf("%d",x) | Displays value from 10 to 0 | "x" is decreased from 10 to 0 |

**Program to display from 1 to 10 using for loop**

```
#include<stdio.h>
#include<conio.h>
main()
{
int x;
clrscr();
for(x=1;x<=10;x++)
Printf("/n%d",x);
getch();
}
```

**Result:**

```
1 2 3 4 5 6 7 8 9 10
```

**<u>Unconditional Control Statements</u>**

**<u>Break Statement:</u>**

A break statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop.

So, we may use break statement to terminate for,while,do..while loops or to exit from a switch.

**<u>Syntax:</u>**

```
break;
```

**Example:**

```
#include<stdio.h>
int main()
{
int x=1;
while(x<10)
{
          Printf("value of x: %d \n",x);
           x++;
          if(x>5)
          {
                    Break;                    /*loop terminated using break
statement*/
}
}
return 0;
}
```

**Result:**

```
Value of x: 1

Value of x: 2

Value of x: 3

Value of x: 4

Value of x: 5
```

**Continue Statement:** The loop does not terminate when a continue statement is met. It will simply skip the remaining statements in that iteration. It will proceed to the next pass in a loop.

Continue statement can be included within a "while", a "do..while", a "for" loops.

**Syntax:**

```
Continue;
```

**Example:**

```
#include<stdio.h>
int main()
{
int x=1;
do
{
if(x==5)
{
        X=x+1;
        Continue;                /*skip the iteration*/
}
printf("value of x: %d\n",x);
X++;
}while(x<10);
return 0;
}
```

**Result:**

```
Value of x:1
Value of x:2
Value of x:3
Value of x:4
Value of x:6
Value of x:7
Value of x:8
Value of x:9
```

**Goto statement** A goto statement provides an unconditional jump from the goto to a labeled statement in the same program.

Most of the programming languages does not encourage the use of goto statement because it is difficult to trace the control flow of a program. It is difficult to understand and modify the program.

**Syntax:**

```
goto label;

…

…

label: statement;
```

**Example:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int a;

clrscr();
printf("Enter a number: ");
scanf("%d",&a);
if(a%2==0)
goto even;
else
goto odd;
even:
printf("\n %d is Even number");
return;
odd:
printf("\n%d is Odd number");
}
```

**Result:**

```
Enter a Number: 20

20 is Even number
```

**Summary:**

In this session, we learnt about:

- C operators – Arithmetic,logical,Assignment,relational,increment&decrement,conditional,bitwise,special operators
- Operator precedence and associativity
- C expressions – arithmetic expressions – evaluation of expressions – type cast operator
- Formatted input&output,Unformatted input&output statements
- Branching statements – simple if – if..else..- else..if ladder – nested if..else – switch statement – goto statement
- Looping statements – while – do..while statements – for loop – break & continue statement

# UNIT III

## ARRAYS and STRINGS FUNCTIONS

**Objectives**

Upon completion of instruction in this section the students shall be able to:

1. Define, describe, and explain the array data structure.
2. Use arrays to store, sort, etc.
3. Declare an one dimensional and two dimensional subscript arrays.
4. Initialize an one dimensional and two dimensional subscript arrays.
5. Reference (read/write to) individual elements of an array.
6. Declare and manipulate Strings and its functions.
7. Use the functions in the C standard library
8. Explain the relationship between the function prototype, function definition, and function call.
9. Explain the relationship between formal parameters in a function definition and arguments in a function call.
10. Construct programs in a modular manner using functions.
11. Define and explain variable scope and storage classes
12. Write and use functions which call themselves. (Recursive Functions).

# ARRAYS and STRINGS FUNCTIONS

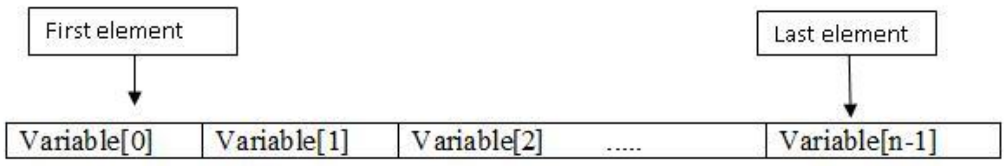
## 3.1 ARRAYS:-

Array is a fixed size sequential collection of elements of same data type. Generally a variable can store only one value at a given time. But an array is used to store a collection of elements. The elements should be of the same type.

Instead of declaring individual variables, such as variable 0, variable 1, ..., and variable n, one array variable can be created as

Array name [size];

Size indicates number of elements in the array. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Array is one of the data structure in C. Arrays are of the following types.

1. one dimensional array
2. two dimensional Array
3. multi dimensional Array

## DECLARATION AND INITIALIZATION OF ONE DIMENSIONAL, TWO DIMENSIONAL AND CHARACTER ARRAYS

### One dimensional array

One Dimensional Array is an array having a single index/subscript value to represent the array elements.

### Declaration

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in the memory. The declaration form of one dimensional array is

data type array name [size];

The data type specifies the type of the element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array

Eg:

To create an array called “a” with 5 integer numbers, the declaration is as follows

```
int a[5];
```

The computer allocates 5 memory locations. As ‘C’ arrays are indexed from 0, the first element of the array is stored at a[0] and the last element of the array at a[4]. Let the 5 integer numbers be {23,34,54,23,11}, then

```
a[0] = 23
a[1] = 34
a[2] = 54
a[3] = 23
a[4] = 11
```

**Initialization**

Arrays could be initialized with elements either at Compile time or Run time.

*Compile Time initialization:*

The elements of arrays can be initialized as the ordinary variables when they are

declared. The general form of initialization of array is:

```
type array name[size]= { list of values };
```

The values in the list are separated by commas.

Example: `int a[3] = { 21, 5, 33};`

The variable ’a’ is declared as an array of size 3 and the values 21, 5 and 33 are assigned to each element a[0],a[1],a[2]. If the number of values in the list is less than the number of elements, then only the given values will be initialized to the elements . The remaining elements will be set to zero automatically.

If the number of values are more than the declared size, the compiler will produce an error.

*Run time Initialization :*

An array can also be explicitly initialized at run time.

Example:

Consider the following segment of a C program.

```
            for(i=0;i<10;i++)
  {
       scanf(" %d ", &x[i] );
  }
```

In the above example, array elements are initialized with the values entered through the keyboard. Looking statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user.

**Program :**

A program to store the elements in the array and to print them from the array

```
#include<stdio.h>
void main()
{
int a[5],i;
printf(“ enter 5 numbers \n”);
for (i=0;i<5;i++)
{
scanf("%d",&a[i]);
}
printf ( “ the elements in the array are \n”);
for (i=0;i<5;i++)
{
    printf("Element stored at a[%d] = %d \n",i,a[i]);
                        }
getch();
}
```

**Input**

```
Enter 5 numbers  11  22  33  44  55
```

**Output**

```
        Elements in the array are

Element stored at a[0] = 11
Element stored at a[1] = 22
Element stored at a[2] = 33
Element stored at a[3] = 44
Element stored at a[4] = 55
```

**Two dimensional array**

Two dimensional arrays are used to store a table of values (matrix). Two subscripts are used in a 2D array. First subscript for the rows of the table and second subscript for the columns of the table.

*Declaration*

Two dimensional arrays are declared using the following syntax

```
type array_name [row_size][column_size];
```

*e.g*:

```
int matrix [3][3];
```

The array contains three rows and three columns, so it is said to be a 3-by-3 array. In general, an array with *m* rows and *n* columns is called an m-by-n array.

*Initializing Two Dimensional Array*

Like the one dimensional array, 2D arrays can be initialized in both the two ways; the compile time initialization and the run time initialization.

***Compile Time initialization***

The two dimensional array may be initialized by the list of initial values enclosed in braces.

Example:

```
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Here the initialization is done row by row. The above statement can also be written as

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
```

or

in the matrix format as

```
int c[2][3] = {
                    { 1, 3, 0 },
                     {-1, 5, 9 }
                 };
```

**Run Time initialization**

In the initialization of one dimensional array, the looping statements were used to set the values of the array one by one. In the similar way two dimensional array are initialized by using the looping structure. To initialize the two dimensional array, the nested loop structure is used; outer For loop for the rows (first sub-script) and the inner For loop for the columns (second sub-script) of the two dimensional array.

```
for(i=0;i<3;i++)
```

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

The above loop creates a two dimensional array ar with 3 rows and 3 columns.

**<u>Program :</u>**

```
# include <stdio.h>
#include <conio.h>
void main()
{
int a[3][3],i,j,count =0;
for(i=0;i<3;i++)
          {
          for (j=0;j<3;j++)
                    {
                    count++;
                    a[i][j]=count;
                    printf("%d \t", array*i+*j+);
                    }
          printf("\n");
          }
           getch();
}
```

**Output**

```
1       2  3
4       5  6
7       8  9
```

**Character arrays**

A string is a sequence of characters. C language does not support string as a separate data type. Strings can be represented as a character array. Strings are one dimensional character array terminated by a null character '\0'.

*Declaring character array :*

```
Char array_name[size];
```

The size determines the number of characters in an array. The size should be the actual number of characters plus one for null character.

Example. :

```
Char name [25];
```

*Initializing character arrays:*

Like numeric arrays, character arrays can be initialized at the time of declaration. C permits the character array to be initialized in either of the following two forms

```
Char dept *4+ =”ece”

Char dept *4+ = , ‘e’, ’c’, ’e’, ’\0’-
```

Character array could also be initialized without specifying the size of the array as

```
Char dept *+ = , ‘e’, ’c’, ’e’, ’\0’-
```

In the above case, the array 'dept' is a 4 element array.

When the size of the array is declared to be much larger than the string size , the other elements are initialized as NULL.

For eg:

Char dept*7+= , 'e', 'c', 'e', '\0'-

| | | | | | | |
|---|---|---|---|---|---|---|
| e | C | e | \0 | \0 | \0 | \0 |

But the following declaration shows error,

Char dept*2+ = , 'e', 'c', 'e', '\0'-

because the size is less than the number of elements in the array.

**ACCESSING ARRAY ELEMENTS**

Array can be accessed using array-name and subscript variable written inside pair of square brackets []. A single element in an array can be accessed by its index or subscript. To access the fourth element in an array named X, the format is

X[4];

**PROGRAMS USING ARRAYS.**

1. Program to find the Sum of two minimum element in an array

```
#include <stdio.h>
void main()
{
int a[10], i, j, sum=0,temp,n;
printf("Enter array size: ");
scanf("%d", &n);
for(i = 0;i < n;i++)
scanf("%d",&a[i]); // input the array elements for(i =
0;i < n-1;i++)
{
for(j = i;j < n;j++)
{
if(a[i] > a[j])
{
temp = a[i];
a[i] = a[j];
a[j]=temp;
}
}
}
sum = a[0] + a[1];
printf("\nsum of minimum two elements are : %d ", sum);
}
```

**Output**

```
Enter array size : 5
1 3 5 7 9
sum of minimum two elements are : 4
```

2. Addition of two matrices in an array

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

```
int i , j, r, c, a[10][10], b[10][10] ;

printf( "Enter the order of matrix up to (10 x 10): \n" ); scanf(

"%d %d " , &r , &c );

printf( "\nEnter the Elements of matrix A :\n" ); for( i =

0; i<r; i++ )

{

for( j = 0; j<c; j++ )

{

scanf( "%d " , &a[i][j] );

}

}

printf( "\nEnter the Elements of matrix B: \n" ); for( i =

0; i < r; i++ )

{

for( j = 0; j < c; j++ )

{

scanf("%d ", &b[i][j] );

}

}

printf("\nMatrix Addition: \n" ); for( i

= 0; i < r; i++ )

{

for( j = 0; j < c; j++ )

{

printf("%4d" , a[i][j] + b[i][j] );

}

printf("\n" );

}

}
```

3. Program to calculate the Sum of elements in an array

```
#include <stdio.h>
void main()
{
int sum=0, a[10], i, n, d;
printf("Enter array size : ");
scanf("%d",&n); printf("Enter
elements : "); for(i = 0;i < n;i++)
scanf("%d",&a[i]);

for(i = 0;i < n;i++)
sum += a[i];
printf("the sum of the elements is %d ",sum);
}
```

4. Program to sort the given numbers in ascending order

```
#include<stdio.h>
void main()
{
int  i,  j,  n,  a[30],  temp  =  0;
printf("\nEnter   a   limit   :   ");
scanf("%d", &n);
printf("\nEnter %d Numbers to sort :",n); for(i =
0;i < n;i++)
scanf("%d", &a[i]);

for(i = 0;i < n-1;i++)

{

for(j = i;j < n;j++)

{

if(a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

printf("\n----Ascending Order----\n");

for(i = 0;i < n;i++)

printf("%d ", a[i]);

}
```

**3.2 STRINGS :-**

A string in C is actually a character array. As an individual character variable can store only one character, we need an array of characters to store strings. Thus, in C string is stored in an array of characters. Each character in a string occupies one location in an array.

In C, strings are terminated by a null character. The null character '\0' is put after the last character. a string not terminated by '\0' is not really a string, but merely a collection of characters. A string may contain any character, including special control characters, such as \n, \t, \\ etc...

There is an important distinction between a string and a single character in C. The convention is that single characters are enclosed by single quotes e.g. '*' and have the type char. Strings, on the hand, are enclosed by double quotes e.g. "name" and have the type array of char.

**DECLARATION AND INITIALIZATION OF STRING VARIABLES:**

As there is no separate data type for strings, Strings are declared as an array of character data type.

**Syntax :**

char String_Variable_name [ SIZE ] ;

The size determines the number of character in the string.

*Eg. :* char s[10];

Strings can be initialized in a number of ways.

*Example:*

1. char c[5] = {'a', 'b', 'c', 'd', '\0'};

2. char c*5+ = “abcd”;

In the first example , string is initialized by listing the elements. Null character is explicitly given in such case. Size of the array is equal to the maximum number of elements in the string plus one for the null character(‘\0’).

3. char c[] = "abcd";

In the above example the size of the array is not given and is determined automatically based on the number of elements initialized.

**READING STRINGS :**

**Using scanf function :**

The **scanf()** functioncan be used to read a string like any other data types. Input function scanf() can be used with %s format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on first white space it encounters. Therefore if you try to read an input string "Programming in C" using scanf() function, it will only read “Programming” and terminate after encountering white spaces.

```
char subject[50];

scanf (“%s”, subject);
```

The address operator(&) is not required in the scanf() function here.

In order to a line of characters including white spaces, C supports a format specification known as edit conversion code %[ . ].

Example:

```
char line[80];

Scanf(“%*^\n+”, line);
```

**Using getchar()**

Getchar() function is a function which is used to accept the single character from the user. This function can be used repeatedly to read successive single characters. The reading is terminated when the newline character (“\n”) is encountered. The null character is then inserted at the end of the string.

**Syntax:**

```
char ch = getchar();
```

**Program to read line of text character by character.**

```
#include <stdio.h>

void main()

{

  char name[30], ch;

  int i = 0;

  printf("Enter name: ");

  while(ch != '\n')  // terminates if user hit enter

  {

    ch = getchar();

    name[i] = ch;

    i++;

  }

  name[i] = '\0';      // inserting null character at end
```

```
   printf("Name: %s", name);

   }
```

In the program above, using the function **getchar()**, getch gets a single character from the user each time.This process is repeated until the user enters return (enter key). Finally, the null character is inserted at the end to make it a string.

**Using gets**

**Gets function r**eads characters from the standard input and stores them as a C string into str until a newline character or the end-of-file is reached. Thus a string of text containing whitespaces can be read easily using gets function.

```
char * gets ( char * str );

     OR

gets( <variable-name> );
```

**Example:**

```
        Char line[80];

        gets(line);

        printf("%s",line);
```

**<u>WRITING STRINGS :</u>**

**Using printf function :**

The printf function is used to display an array of characters termination with null character i.e strings.

```
     Printf("%s", name);
```

The above statement displays the entire content of the string name.

**Example:**

```
#include <stdio.h>

void main()

{

   char name[20];

   printf("Enter subject name: ");

   scanf("%s", name);

   printf("subject name is %s.", name);

   }
```

**Output**

```
Enter name: programming in C

Subject name is programming.
```

Here, program ignores “in C” because, **scanf()** function takes only a single string before the white space, i.e. programming.

**Using putchar ()**

Putchar is used to print a character . it is one of the character handling function.

**Syntax :**

```
int putchar(int c);
```

Example:

```
 Char ch=’a’;

Putchar(ch);  // displays the character ‘a’.
```

**Using puts()**

Puts function is used to print string values.

**Syntax:**

```
Puts(string variable);
```

**Example Program** - **Program to read line of text using gets() and puts()**

```
#include <stdio.h>
 void main()
{
   char name[30];
   printf("Enter name: ");
   gets(name);    //Function to read string from user.
   printf("Name: ");
   puts(name);  //Function to display string.
   return 0;
}
```

**Output**

```
Enter name: Dennis Ritchie
Name: Dennis Ritchie
```

## STRING HANDLING FUNCTIONS (STRLEN(),STRCAT(),STRCMP())

**Strlen():**

**strlen()** is one of the inbuilt string function in c programming which is used to find the length of the given string.

**Syntax**

```
length = strlen(str);
```

**strlen()** accept only one string as a parameter and returns length of string which is of integer type. To use **strlen()** inbuilt string functions in C, we need to declare <string.h> header file.

**Program Using strlen()**

```
#include <stdio.h>

void main()

{

char str1[] = "ThisisString";

printf("Length of str1 = %d ", strlen(str1));

}
```

**Output**

```
Length of str1 = 12
```

The above program prints the length of the string by excluding a null character at the end of the string.

**strcat() :**

**strcat()** is one of the inbuilt string function in c programming which is used to combine two strings to form a single one. The full form of strcat() is string concatenation.

**Syntax:**

```
strcat (string1, string2);
```

**strcat()** accepts two parameters.Both parameters must be a string.To use **strcat()** inbuilt string function in C, we need to declare <string.h> header file. When the function strcat is executed, string2 is appended to string1. The null character present in string1 is deleted and the string2 is added to string1 without any change.

**Program Using strcat()**

```
#include<stdio.h>

#include<string.h>

void main()

{

char str1[30] = "How are ", str2[30] = "you";

strcat (str1, str2);

printf("Str1 : %s ", str1);

}
```

**Output**

```
Str1 : How are you
```

**strcmp()**

**strcmp()** is one of the inbuilt string function in c programming which is used to compare two strings, if the strings are same then the function returns 0. Otherwise it returns a nonzero value.

| Type | Return Value |
|---|---|
| str1 > str2. | Positive value |
| str1 == str2. | 0 |
| str1 < str2. | Negative value |

**Syntax:**

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

**Program Using strcmp()**

```
#include<stdio.h>

#include<string.h>

void main()

{

char str1[20] = "this is strcmp", str2[20] = "this is strcmp";

if(strcmp(str1, str2) == 0)

printf("The strings str1 and str2 are same ");

}
```

**Output**

```
        The strings str1 and str2 are same
```

The above program defines the function strcmp(), which is used to compare two strings. Here, str1 and str2 are same, so it returns 0 and prints the statement inside the condition.

**String manipulation programs.**

1. Program to Sort

```
strings #include<stdio.h>

#include <string.h>

void main()

{

   int i, j;

   char   str[10][50],   temp[50];

   printf("Enter  10  words:\n");

   for(i=0; i<10; ++i)

      scanf("%s[^\n]",str[i]);
```

```
  for(i=0; i<9; ++i)

      for(j=i+1; j<10 ; ++j)

      {

          if(strcmp(str[i], str[j])>0)

          {

              strcpy(temp, str[i]);

              strcpy(str[i], str[j]);

              strcpy(str[j], temp);

          }

      }

  printf("\nIn alphabetical order: \n");

  for(i=0; i<10; ++i)

  {

      puts(str[i]);

  }

 }
```

2. Program to swap two strings

```
#include   <stdio.h>
#include  <string.h>
Void main()
{
  char first[100], second[100], temp[100];

  printf("Enter the first
  string\n"); gets(first);
  printf("Enter the second
  string\n"); gets(second);
  printf("\nBefore Swapping\n");
  printf("First string: %s\n",first);
  printf("Second string:
  %s\n\n",second); strcpy(temp,first);
  strcpy(first,second);
  strcpy(second,temp);
  printf("After Swapping\n");
  printf("First string: %s\n",first);
  printf("Second string: %s\n",second);
}
```

3. Program to change the case of string without using string library functions

```
#include <stdio.h>
 # include <string.h>
void main ()
{
  int c = 0;
  char ch, s[1000];
  printf("Input a string\n");
  gets(s);
   while (s[c] != '\0')
     { ch = s[c];
     if (ch >= 'A' && ch <= 'Z')
        s[c] = s[c] + 32;
     else if (ch >= 'a' && ch <= 'z')
        s[c] = s[c] - 32;
     c++;
  }
   printf("%s\n", s);
 }
```

**3.3 Built –in functions: -**

**Math functions**

1. floor()

floor( ) function in C returns the nearest integer value which is less than or equal to the floating point argument passed to this function.

**Syntax:**

```
double floor ( double x );
```

2. **abs**()

abs( ) function in C returns the absolute value of an integer. The absolute value of a number is always positive. Only integer values are supported in C.

**Syntax:**

```
int abs ( int n );
```

**3. round()**

round( ) function in C returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.

**Syntax:**

```
double round (double a);
float roundf (float a);
long double roundl (long double a);
```

**4. ceil()**

ceil( ) function in C returns nearest integer value which is greater than or equal to the argument passed to this function.

**Syntax:**

```
double ceil (double x);
```

**5.exp()**

exp( ) function is used to calculate the exponential "e" to the xth power. log( ) function is used to calculates natural logarithm and log10( ) function is used to calculates base 10 logarithm.

**6. sqrt()**

sqrt( ) function in C is used to find the square root of the given number.

**Syntax:**

```
double sqrt (double x);
```

**7. pow()**

pow( ) function in C is used to find the power of the given number.

**Syntax:**

```
double pow (double base, double exponent);
```

**8. trunc()**

trunc( ) function in C truncates the decimal value from floating point value and returns integer value.

**Syntax:**

```
double trunc (double
a); float truncf (float a);
long double truncl (long double a);
```

**9.cos()**

This function returns the cosine of a radian angle x.

**Syntax:**

```
double cos(double x)
```

x is the floating point value representing an angle expressed in radians.

**10. sin()**

The sin() function returns the sine of a radian angle x.

**Syntax:**

```
double sin(double x)
```

**CONSOLE I/O FUNCTIONS**

Console input / output functions are defined in conio.h header file.

**LIST OF INBUILT C FUNCTIONS IN CONIO.H FILE:**

| **Functions** | **Description** |
|---|---|
| clrscr() | This function is used to clear the output screen. |
| getch() | It reads character from keyboard |
| getche() | It reads character from keyboard and echoes to o/p screen |
| textcolor() | This function is used to change the text color |
| textbackground() | This function is used to change text background |

**STANDARD I/O FUNCTIONS**

Standard input/output functions are declared in stdio.h header file

| **Function** | **Description** |
|---|---|
| printf() | This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen |
| scanf() | This function is used to read a character, string, numeric data from keyboard. |
| getc() | It reads character from file |
| gets() | It reads line from keyboard |
| getchar() | It reads character from keyboard |
| puts() | It writes line to o/p screen |
| putchar() | It writes a character to screen |
| putc() | writes a character to file |
| putc() | writes a character to file |
| remove() | deletes a file |

**CHARACTER ORIENTED FUNCTIONS**

Character oriented functions are defined in ctype.h header file.

| Functions | Description |
|---|---|
| isalpha() | checks whether character is alphabetic |
| isdigit() | checks whether character is digit |
| isalnum() | Checks whether character is alphanumeric |
| isspace() | Checks whether character is space |
| islower() | Checks whether character is lower case |
| isupper() | Checks whether character is upper case |
| isxdigit() | Checks whether character is hexadecimal |
| iscntrl() | Checks whether character is a control character |
| isprint() | Checks whether character is a printable character |
| ispunct() | Checks whether character is a punctuation |
| isgraph() | Checks whether character is a graphical character |
| tolower() | Checks whether character is alphabetic & converts to lower case |
| toupper() | Checks whether character is alphabetic & converts to upper case |

**isdigit()**

The isdigit() function checks whether a character is numeric character (0-9) or not.

**Syntax:**

```
int isdigit( int arg );
```

Function isdigit() takes a single argument in the form of an integer and returns the value of type **int**.Even though, isdigit() takes integer as an argument, character is passed to the function. Internally, the character is converted to its ASCII value for the check.

**islower()**

The islower() function checks whether a character is lowercase alphabet (a-z) or not.

**Syntax:**

```
int islower( int arg );
```

Function islower() takes a single argument in the form of an integer and returns a value of type **int**.

**isupper()**

The isupper() function checks whether a character is an uppercase alphabet (A-Z) or not.

**Syntax:**

```
int isupper(int argument);
```

**tolower()**

The tolower() function takes an uppercase alphabet and convert it to a lowercase character.If the arguments passed to the tolower() function is other than an uppercase alphabet, it returns the same character that is passed to the function.

**Syntax:**

```
int tolower(int argument);
```

The character is stored in integer form in C programming. When a character is passed as an argument, corresponding ASCII value (integer) of the character is passed instead of that character itself.

**toupper()**

The toupper() function converts a lowercase alphabet to an uppercase alphabet, if the argument passed is an lowercase alphabet.

**Syntax:**

int toupper( int arg );

Function toupper() takes a single argument in the integer form and returns a value of type **int**.Even though, toupper() takes integer as an argument, character is passed to the function. Internally, the character is converted to its corresponding ASCII value for the check.If the argument passed is other than a lowercase alphabet, it returns the same character passed to the function.

**3.4 USER DEFINED FUNCTIONS:-**

A large C program is divided into basic building blocks called C function. Actually, Collection of these functions creates a C program.In C programming, a function is a group of statements that performs a certain task in a program.A program can be defined by using one or more functions. C function contains set of instructions enclosed by ", -" which performs specific operation in a C program. Each function has its own function name, the function can be called using the function name.

There are two types of functions they are,

**Library Functions :**

C provides many pre-defined functions to perform a task, those functions are defined in an appropriate header files.

e.g. printf(), scanf(), strlen(), sqrt() and so on.

**User Defined functions:**

C allows users or programmers to define a function according to their requirement.

**DEFINING FUNCTIONS & NEEDS**

**USES OF C FUNCTIONS:**

- ➢ C functions are used to avoid rewriting same logic/code again and again in a program.
- ➢ There is no limit in calling C functions to make use of same functionality wherever required.
- ➢ We can call functions any number of times in a program and from any place in a program.
- ➢ A large C program can easily be tracked when it is divided into functions.
- ➢ The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

There are 3 aspects in each C function. They are,

| C functions aspects | Syntax |
|---|---|
| function definition | Return_type function_name (arguments list)<br>{ Body of function; } |
| function call | function_name (arguments list); |
| function declaration | return_type function_name (argument list); |

**<u>Function Definition</u>**

The function declaration statement informs the compiler about a function return type, function name and parameters or arguments type.

**<u>Syntax:</u>**

return-type function-name (arguments);

**where**

- Return type is the data type of the value which is given back to the calling function.
- Function name is the name of a function. A function is called by using the function name.
- Parameters or arguments type : C allows programmers to pass information to the called function from the calling function by using parameters. These parameters are variables of data type.

**<u>Format of function:</u>**

```
main()
{
.
.
statements within main function;
function-name (arguments);              //function call
.
}
return-type function-name (arguments)              //function definition
{
.
.
statements within user-defined function;
.
return variablename;
}
```

Example :

Program to add two integers using function.

```
#include <stdio.h>

int add(int ,int );   //function declaration

int main()

{

//main function definition int

a = 5, b = 10;

int sum;

printf("The value of a and b : %d %d ", a, b); sum =

add(a, b); //function call printf("\nsum = %d ",

sum);

}

// function definition int

add(int a, int b)

{

int c;

c = a + b;

return c;   //returns a integer value to the calling function

}
```

**SCOPE AND LIFE TIME OF VARIABLES:**

A scope is a region of the program where the variables can be accessed.A scope contains a group of statements and variables. The variables declared within a block can be accessed only within that block. Programmers can declare the variables both inside and outside of block.

*Scopes are*

- Local scope
- Global scope

*Local Variables or Local Scope*

The variables declared inside a function or a block is known as local variables. Local variables can be accessed only within function or block. Local variables cannot be accessed outside a function or a block.

**Example**

```
#include <stdio.h>

void main()

{

int m1 = 5,m2 = 10;  //Local variable declaration

int mul;  //Local variable declaration

mul = m1 * m2;

printf("Multiplication of %d and %d : %d ", m1, m2, mul);

}
```

**Output:**

```
Multiplication of 5 and 10 : 50
```

The variables are declared inside the main function. So, those cannot be accessed outside the main function.

*Global Scope or Global variables*

The variables declared right before the main function is called as global variables. Global variables can be accessed through out the program.

**Example**

```
#include <stdio.h>

int m1 = 5, m2 = 10;  //global variable declaration

void add();

int main()

{

int mul;  //Local variable declaration

mul = m1 * m2;

printf("\nMultiplication of %d and %d : %d ", m1, m2, mul);

add();

return 0;

}

void add()

{

int sum;

m1 = 2; m2 = 4;  //global variable initialization in add function

sum = m1 + m2;

printf("\nSum of %d and %d : %d ", m1, m2, sum);

}
```

**Output**

```
Multiplication of 5 and 10 : 50
Sum of 2 and 4 : 6
```

The variables m1 and m2 are declared outside all functions. So, those can be accessed in any function within a program.

**FUNCTION CALL**

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name as

function_name (arguments list);

While calling a function, there are two ways in which arguments can be passed to a function.They are,

- Call by value
- Call by reference

**1. CALL BY VALUE:**

In call by value method, the value of the variable is passed to the function as parameter.The value of the actual parameter can not be modified by formal parameter.

Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Actual parameter – This is the argument which is used in function call.

Formal parameter – This is the argument which is used in function definition

**EXAMPLE :**

In this program, the values of the variables “m” and “n” are passed to the function “swap”.

These values are copied to formal parameters “a” and “b” in swap function and used.

```
#include<stdio.h>

// function prototype, also called function declaration

void swap(int a, int b);


int main()

{
    int m = 22, n = 44;

    // calling swap function by value

    printf(" values before swap m = %d \nand n = %d", m, n);

    swap(m, n);

}


void swap(int a, int b)

{

    int tmp;

    tmp = a;

    a = b;

    b = tmp;

    printf(" \nvalues after swap m = %d\n and n = %d", a, b);

}
```

**OUTPUT:**

```
values before swap m =
10 and n = 40
values after swap m = 40
and n = 10
```

**2. CALL BY REFERENCE:**

In call by reference method, the address of the variable is passed to the function as parameter. The value of the actual parameter can be modified by formal parameter.Same memory is used for both actual and formal parameters since only address is used by both parameters.

**EXAMPLE**

In this program, the address of the variables “m” and “n” are passed to the function “swap”. These values are not copied to formal parameters “a” and “b” in swap function but they are just holding the address of those variables. This address is used to access and change the values of the variables.

```
#include<stdio.h>

// function prototype, also called function declaration

void swap(int *a, int *b);


int main()

{

   int m = 22, n = 44;

   // calling swap function by reference

   printf("values before swap m = %d \n and n = %d",m,n);

   swap(&m, &n);

}


void swap(int *a, int *b)

{

   int tmp;

   tmp = *a;

   *a = *b;

   *b = tmp;

   printf("\n values after swap a = %d \nand b = %d", *a, *b);

}
```

**RETURN VALUES**

The return statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the function call. A return statement can also return a value to the calling function.

**Syntax:**

**return** *expression* **;**
**return;**

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is evaluated and then converted to the type returned by the function. If the function was declared with return type void, a return statement containing an expression generates a warning and the expression is not evaluated. If no return statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. In this case, the return value of the called function is undefined. If a return value is not required, declare the function to have void return type; otherwise, the default return type is int.

**CATEGORY OF FUNCTIONS**

All C functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function.

The categories of functions are

C function with arguments (parameters) and with return value.

C function with arguments (parameters) and without return value.

C function without arguments (parameters) and without return value.

C function without arguments (parameters) and with return value.

| **C function categories** | **syntax** |
|---|---|
| 1. With arguments and with return values | **function declaration:**<br>int function ( int );<br><br>**function call:** function ( a );<br><br>**function definition:**<br>int function( int a )<br>{<br>statements;<br>return a;<br>} |
| 2. With arguments and without return values | **function declaration:**<br>void function ( int );<br><br>**function call:** function( a );<br><br>**function definition:**<br>void function( int a )<br>{ |

| | |
|---|---|
| | statements;<br>} |
| 3. Without arguments and without return values | **function declaration:**<br>void function();<br><br>**function call:** function();<br><br>**function definition:**<br>void function()<br>{<br>statements;<br>} |
| 4. Without arguments and with return values | **function declaration:**<br>int function ( );<br><br>**function call:** function ( );<br><br>**function definition:**<br>int function( )<br>{<br>statements;<br>return a;<br>} |

**1. WITH ARGUMENTS & WITH RETURN VALUE:**

Here there is two way communications between functions. The program(function ) that calls the function is called as calling function. The calling function sends data to the

Called function. After processing , the called function returns one required value to the calling function. Thus there is a two way data transfer.

**<u>Program:</u>**

```
#include <stdio.h>

int add(int ,int );

 int main()

{

int a = 5, b = 10;
int sum;

printf("The value of a and b : %d %d ", a, b);

sum = add(a, b);

printf("\nsum = %d ", sum);

}

int add(int a, int b)

{

int c;

c = a + b;

return c;

 }
```

In the above program, the calling function sends parameters a,b to the function add. The add function calculates and returns the sum c to the calling(main) function.

**2. WITH ARGUMENTS & WITHOUT RETURN VALUE:**

In this program, integer value passed as argument to the function. The return type of this function is “void” and no values can be returned from the function. The values of integer, is manipulated and displayed inside the function itself.

Example:

```
#include<stdio.h>

void display(int);

int main()

{

    int a = 20;

    display(a);

    return 0;

}
```

```
void display(int a)

{

    printf(" Integer %d\n\n",a);

}
```

**3. WITHOUT ARGUMENTS & WITHOUT RETURN VALUE:**

In this program, no values are passed to the function “test” and no values are returned from this function to main function.

```
#include<stdio.h>

void test();


int main()

{

   test();

   return 0;

}


void test()

{

   int a = 50, b = 80;

   printf("\nvalues : a = %d and b = %d", a, b);

}
```

**4. WITHOUT ARGUMENTS & WITH RETURN VALUE:**

In this program, no arguments are passed to the function “sum”. But, values are returned from this function to main function. Values of the variable a and b are summed up in the function “sum” and the sum of these value is returned to the main function.

```
#include<stdio.h>


int sum();


int main()

{

   int addition;

   addition = sum();

   printf("\nSum of two given values = %d", addition);

   return 0;

}


int sum()

{

   int a = 50, b = 80, sum;

   sum = a + b;

   return sum;

}
```

Only one value can be returned from a function. if more than one values are given in the return statement, only one value will be returned that appears at the right most place of the return statement. For example, in “return a,b,c” , value for c only will be returned and values a, b won’t be returned to the program.

In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

## STORAGE CLASSES

A storage class defines the scope, visibility and life-time of variables. A scope is defined as the area in which the declared variable is available. Visibility is the way of hiding or showing a variable inside or outside a scope. Visibility is otherwise said to be accessibility.

Life-time is defined as a period of time in which the variable lives. There are three types of life-times in storage classes. They are static life-time, automatic life-time, and dynamic lifetime

There are four types of storage classes in c programming. They are

| **Keyword** | **Storage Classes types** |
|---|---|
| auto | automatic storage class |
| static | static storage class |
| register | register storage class |
| extern | external storage class |

### Auto Storage Class

All variables declared inside a function without any storage class keyword is considered as auto storage class by default. In auto storage class a memory is allocated automatically to the variables when the function is called and deallocates automatically when the function exits.

Variables under auto storage classes are local to the block or function. So, it is also called as local variables. The keyword auto is rarely used because all uninitialized variables are said to have auto storage classes.

**Syntax:**

```
auto int m1;
```

**Program**

```
#include <stdio.h>

void main()

{

auto int x = 10;  // declaration of automatic variable with the keyword auto

int y = 20;   //declaration of automatic variable without the keyword auto
printf("The value of x : %d ", x);

printf("\nThe value of y : %d ", y);

}
```

**Extern Storage Class**

Extern variables are also known as global variables because extern variables are declared above the main function. So, the variables can be accessed by any function. We can also access extern variables of one file to another file. But make sure both files are in same folder.

The keyword for a variable to declared under extern storage class is extern

**Syntax:**

```
extern int m;
```

**Program**

Create a file named **variable.h** .Put all your variables with **extern** keyword which can be used by any program by simply including the file name in it.

*variable.h*

```
extern int num1= 9;

extern int num2 = 1;
```

*program.c*

```
#include <stdio.h>

#include "variable.h"

void main()

{

int add = num1 + num2;
printf("%d + %d = %d ", num1, num2, add);

}
```

**Static Storage Class In C**

A static variable is a variable that tells the compiler to retain the value until the program terminates. They are created once when the function is called, even though the function gets repeated it retains the same value and exists until the program terminates.

The keyword for a variable to declared under static storage class is static

**Syntax:**

```
        static int st;
```

**Program**

```
#include <stdio.h>

void subfun();

void main()

{

subfun();

subfun();

subfun();

}

void subfun()

{

static int st = 1;   //static variable declaration

printf("\nst = %d ", st);

st++;

}
```

**Register Storage Class**

Register variables are similar to auto variables. The only difference is register variables are stored in CPU register instead of memory. Register variables are faster than normal variables. Mostly programmers uses register to store frequently used variables. Programmers can store only few variables in the CPU register because size of register is less. Accessing register variable executes faster than auto variable.

The keyword for a variable to declared under register storage class is register.

**Syntax:**

```
register int m;
```

**Program - register storage class**

```
#include <stdio.h>
void main()
{
int m1 = 5;
register int m2 = 10;
printf("The value of m1 : %d ",m1);
printf("\nThe value of m2 : %d ",m2);
}
```

**RECURSION**

Any function in a C program can be called recursively; that is, it can call itself. The number of recursive calls is limited to the size of the stack. Each time the function is called, new storage is allocated for the parameters so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with **static** storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

*This example illustrates recursive calls:*

```
int factorial( int num );   /* Function prototype */

int main()
{
   int result, number;
   .
   .
   .
   result = factorial( number ); //recursive call
}

int factorial( int num )   /* Function definition */
{
   .
   .
   .
   if ( ( num > 0 ) || ( num <= 10 ) )
      return( num * factorial( num - 1 ) );
}
```

**Program to find factorial of a given number using recursion.**

```
#include<stdio.h>

void main()

{

int f,n;

printf(”enter the number\n”);

scanf(“%d”&n); f=factorial(n);


}



factorial(int n)

{

int fact; if

(n==1)

return(1);

else

fact=n*factorial(n-1);
return(fact);

}
```

**SUMMARY**

- Array is a collection of similar data types that are stored in different memory locations.
- The declaration and initialization of one , two dimensional and character arrays are studied in this chapter with programming examples.
- Passing individual array elements to function are studied in this chapter.
- A function is a segment that groups code in order to perform a specific task
- There is no C program without at least one function
- The main function is the starting point of any C program. This chapter shows how to define one's own functions.
- A function has a type , can have parameters and return a value.
- A recursive function is one that call itself.

# UNIT IV

## STRUCTURES AND UNIONS, DYNAMIC MEMORY MANAGEMENT

**OBJECTIVES :**

- Be able to use compound data structures in programs
- Be able to use compound data structures as function arguments either by value or by reference
- To dynamically allocate memory in your C program using C standard library functions: malloc(), calloc(), free() and realloc()

## 4.1 STRUCTURES AND UNIONS

INTRODUCTION TO STRUCTURE

Structure is a user-defined data type in C for holding data of different datatypes. Structure helps to construct a complex data type in more meaningful way. It is somewhat similar to an Array. The only difference is that array is used to store collection of similar data types while structure can store collection of any type of data.

Structure is used to represent a record. Consider a **Student record** which consists of student name, address, roll number and age. A structure could be defined to hold this information.

### Definition

The **struct** statement is used to define the structure. The struct statement

defines a new data type, with more than one member. The format of the struct statement is as follows.

```
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, one or more structure variables can be specified but it is optional.

Example :

```
    struct Books

    {

      char title[50];
```

```
  char author[50];

  int price;

};
```

Here the **struct Books** declares a structure to hold the details of book which consists of three data fields, namely title,author and price. These fields are called **structure elements or members**. Each member can have different data type, like in this case, **title and author** is of char type and **price** is of int type. **Books** is the name of the structure and is called structure tag.

## ***Declaring Structure Variables***

It is possible to declare variables of a **structure**, after the structure is defined. **Structure** variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in following two ways.

### *1) Declaring Structure variables separately*

```
(i) struct Books

{

  char title[50];

  char author[50];

  int price;

} ;

struct Books Book1,book2; // declaring variables of Books
```

2) Declaring Structure Variables with Structure definition

```
struct Books

{

  char title[50];

  char author[50];

  int price;

}book1,book2;
```

Here **book1** and book**2** are variables of structure **Books**.

## *Accessing Structure Members*

Structure members can be accessed by using dot (.) operator also called **period** or **member**

**access** operator. The member access operator is coded as a dot(.) between the structure variable name and the structure member that is accessed.

Example :

```
    struct Books

    {

      char title[50];

      char author[50];

      int  price;

    }book1,book2;
```

```
book1.price=200;     //book1 is variable of Books type and price is member of Books
```

**Initialization**

*Like any other data type, structure variable can also be initialized at compile time.*

```
    struct Books

    {

      char title[50];

      char author[50];

      int  price;

    };
```

```
struct Books b1 = , “C Programming” ,”xyz”,200 -;  //initialization
```

This assigns the value “C Programming” to b1.title, “xyz” to b1.author, 200 to b1.price. There is a one to one correspondence between the members and the values.

or

```
struct Books b1;
```

```
b1.title = “C Programming”;  //initialization of each member separately

b1.author =”xyz”;

b1.price = 200;
```

We can also use scanf() to give values to structure members through terminal at runtime.

```
scanf(" %s ", b1.name);

scanf(" %d ", &b1.price);
```

## ***Array of Structures***

We can also declare an array of **structures**. Each element of the array represents a s**tructure** variable. Consider the case when the salary of all the employees in a company have to be analyzed.

**Example :**

```
struct employee

{

            char ename[10];

            int sal;

};


Struct employee emp[5];
```

The above code define an array named **emp** of size 5 elements. Each element of array **emp** is of structure type **employee.** The members are accessed as

```
emp*0+.ename=”abc”;
emp[0].sal =20000;
```

**Program to display the details of employees.**

```
#include<stdio.h>

#include<conio.h>

struct employee

{

 char ename[10];

 int sal;
```

```
};


struct employee emp[5];
int i,j;
void main()
{
 clrscr();
 read();
print();
 getch();
}


void read()
            {
      for(i=0;i<3;i++)
 {
  printf("\nEnter %dst employee record\n",i+1);
  printf("\nEmployee name\t");
  scanf("%s",emp[i].ename);
  printf("\nEnter employee salary\t");
  scanf("%d",&emp[i].sal);
 }
}
void print()
{
 printf("\nEmployee
 record\n"); for(i=0;i<3;i++)
 {
  printf("\nEmployee name is %s",emp[i].ename);
  printf("\nSalary is %d",emp[i].sal);
 }
}
```

**Arrays within Structures**

Sometimes, arrays may be a member within structure, this is known as arrays within structure. Arrays can be accessed within structures similar to accessing other members.

For example to use a string value as a structure member, we have to go for array within structure. Similarly single dimension and multidimensional int and float arrays can be used.

**Program to illustrate array within structures and array of structures.**

```
#include <stdio.h>
void main()
{
int i;
struct student {
char   name[30];
int rollno;
} stud[3];
for(i=0; i<3; i++)
{
printf ("\nEnter your RollNo :
"); scanf ("%d",&stud[i].rollno);
printf ("\nEnter your Name : ");
scanf ("%s", stud[i].name);
}
printf ("\nList of all records");
for (i=0; i<3; i++)
{
```

```
printf ("\nRollNo : %d\n Name : %s", stud[i].rollno, stud[i].name);

}

}
```

**Structures Within Structures**

Structures within structures is nothing but Nested structures. One structure can be declared inside other structure.

*Example*

```
        Struct author

                {

                Char name[50];

                Char publisher[50];

                }

        Struct books

                {

                Int id;

                Char name[50]

                Struct author auth; // structure within structure

                }
```

In the above example, author is a structure with name and publisher as members. Books is another structure which has the structure author as a member. This is structures within structures.

**Program**

```
#include <stdio.h>

struct college

    {

      int college_id;

      char college_name[50];

    };
```

```
   struct student
           {
              int id;
             char name[20];
             float percentage;
             struct college clg;  // structure within structure
           }stud;

void main()
{
  struct student stud = {1, "abc", 97.0, 1000, "MS College"};
  printf(" Id is: %d \n", stud.id);
  printf(" Name is: %s \n", stud.name);
  printf(" Percentage is: %f \n\n", stud.percentage);
  printf(" College Id is: %d \n", stud.clg.college_id);
  printf(" College Name is: %s \n", stud.clg.college_name);
 }
```

**<u>Output</u>**

```
 Id is: 1
Name is: abc
Percentage is: 97.0
College Id is: 1000
College Name is: MS college
```

## Structures and functions

C allows programmers to pass each member of a structure or entire structure information to a function. The structure variable may be passed as a value or reference. The function will return the value by using the return statement

**<u>Program to illustrate that structure members are passed as function augments</u>**

```
#include <stdio.h>

int add(int, int) ;   //function declaration

void main()

{

//structures declaration

struct addition{

   int a, b;

   int c;

}sum;

printf("Enter the value of a : \n");

scanf("%d",&sum.a); printf("Enter the

value of b :\n "); scanf("%d",&sum.b);


sum.c = add(sum. a, sum.b); //passing structure members as arguments to function printf("The

sum of two value are :\n ");

printf("%d ", sum.c);

}

//Function definition int
add(int x, int y)

{

int sum1; sum1

= x + y;

return(sum1);

}
```

**<u>Output</u>**

```
Enter the value of a 10
Enter the value of b 20
The sum of two values 30
```

The structure addition have three integer variables a, b, c which are also known as structure members. The variables are read through scanf function. The statement

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

illustrates that the structure members can be passed to the function add. The function add is defined to perform addition operation between two values.

1. **Program to illustrate that entire structure is passed as function arguments**

```
#include <stdio.h>

//structures declaration

  struct addition {

    int a, b;

    int c;

}sum;

void add(sum) ;   //function declaration with struct type sum

void main()

{

Struct sum s1;

printf("Enter the value of a : ");

scanf("%d",&s1.a); printf("\nEnter the

value of b : "); scanf("%d",&s1.b);


add(s1);       //passing entire structure as an argument to function

}

//Function Definition

void add(sum s)
```

```
{

int sum1;

sum1 = s.a + s.b;

printf("\nThe sum of two values are :%d ", sum1);

}
```

**Output**

```
Enter the value of a 10
Enter the value of b 20
The sum of two values 30
```

**2. PROGRAM**

```
 #include<stdio.h>

#include<conio.h>

struct student

{

 char name[10];

 int id;

};

void show(struct student st);

void main()

{

 struct student stud;

 clrscr();

 printf("\nEnter student record\n");

 printf("\nstudent name\t");

 scanf("%s",stud.name);

 printf("\nEnter student roll\t");

 scanf("%d",&stud.id);

 show(stud);
 getch();
}

void show(struct student st)
{
 printf("\nstudent name is %s",st.name);
 printf("\nstudent id is %d",st.id);
}
```

**Output :**

```
student name is XYZ
student id is 001
```

**UNIONS**

**Unions** are conceptually similar to **structures**. A **union** is declared using **union** keyword.The syntax of **union** is also similar to that of structure. The only differences is in terms of storage.

In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member. This implies that although a **union** may contain many members of different types, **it cannot handle all the members at same time**.

```
union stud
{
 int x;
 float y;
 char z;
}s1;
```

This declares a variable **s1** of type union **stud**. This **union** contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for a **union** variable, irrespective of its size. The compiler allocates the storage that is large enough to hold largest variable type in the **union**. In the **union** declared above the member **y** requires 4 bytes which is largest among the members . Other members of **union** will share the same address.

### *Accessing a Union Member*

Syntax for accessing **union** member is similar to accessing structure member,

```
union stud

{

 int x;

 float y;

 char z;

}S1;


S1.x ;  // accessing union member

S1.y;

S1.z ;
```

### ***Program :***

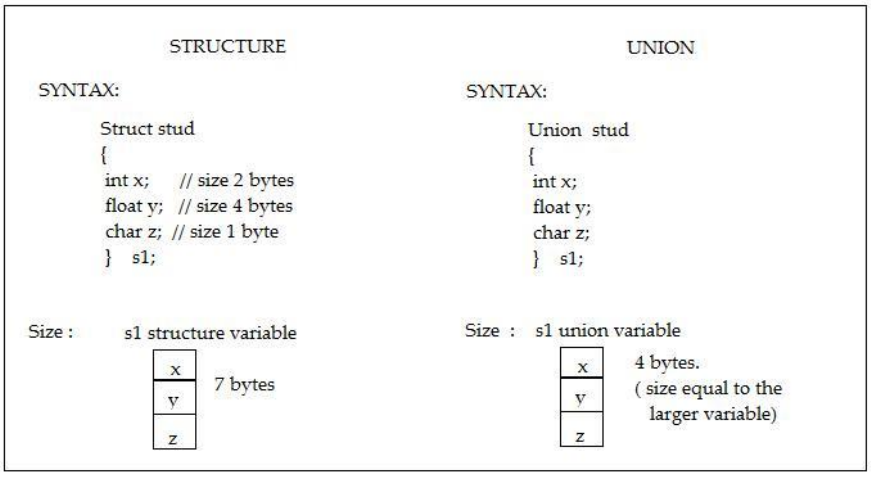
```
#include  <stdio.h>

#include <conio.h>

union demo

{

 int a;

float b;

char c;

};


void main( )

{

 union demo

 d; d.a = 12;

 d.b = 20.2;

 d.c='z';
clrscr();
printf("%d\n",d.a);
printf("%f\n",d.b);
printf("%c\n",d.c);
getch();
}
```

**Output**

```
13178
20.900135
z
```

Here, the values of **a** and **b** get corrupted and only variable **c** prints the expected result. Because in **union**, the only member whose value is currently stored will have the memory.

Difference between Union and structure



STRUCTURE

SYNTAX:

```
Struct stud
{
 int x;     // size 2 bytes
 float y;  // size 4 bytes
 char z; // size 1 byte
 }   s1;
```

Size : s1 structure variable

7 bytes

UNION

SYNTAX:

```
Union  stud
{
 int x;
 float y;
 char z;
 }   s1;
```

Size : s1 union variable

4 bytes.
( size equal to the larger variable)

## 4.2 - Dynamic Memory Management

**Introduction**

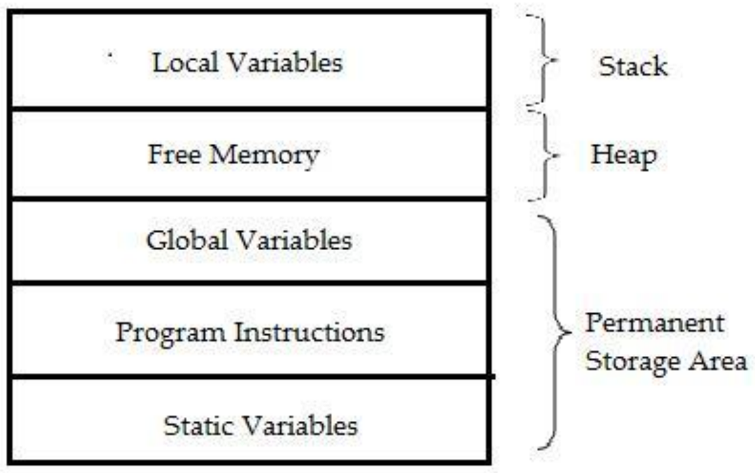
During program execution, the number of data items are dynamic. This leads to increase in the requirement of storage space. Dynamic memory management techniques facilitates allocation of additional memory space or releasing of unwanted memory space. This facility optimizes storage management.

**Dynamic memory allocation**

The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h**.

| Function | Description |
|---|---|
| malloc() | allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space |
| calloc() | allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory |
| Free | releases previously allocated memory |
| Realloc | modify the size of previously allocated space |

**Global** variables, **static** variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in area called **Stack**. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



*Allocating a block memory (MALLOC)*

**malloc()** function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to locate enough space it returns a NULL pointer.

**Syntax**

Pointer variable = (cast-type *) malloc (byte-size);

**Example**

```
int *x;

x = (int*)malloc(50 * sizeof(int));  //memory space allocated to variable x
```

Here memory space equivalent to 50 times the size of integer is reserved. The malloc allocates a block of contiguous bytes. The allocation is not possible when the space in the heap is insufficient.

**Releasing the used space: free**

With dynamic memory allocation, the release of memoryspace should be done when it is not required. The release of memory for future use is done by free function. Free function frees the allocated memory space by malloc(),calloc(),realloc() functions.

```
free(x);                //releases the memory allocated to pointer variable x
```

**Allocating multiple blocks of memory (CALLOC)**

**calloc()** is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**. It allocates multiple blocks of memory If it fails to locate enough space it returns a NULL pointer.

Pointer variable = (cast-type *) calloc (n, elementsize);

The calloc function allocates n blocks of storage, each of same size (elementsize) bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated space is returned

**Example using calloc() :**

```
struct employee

{

 char *name;

 int salary;

};

typedef struct employee emp;
```

emp *e1;

e1 = (emp*)calloc(30,sizeof(emp));

**Altering the size of a block (REALLOC)**

In some situations , the allocated memory is larger or smaller than the required memory space. In such case , the previously allocated memory could be altered with the function realloc. This is reallocation. **realloc()** changes memory size that is already allocated to a variable.

**Example using realloc()**

int *x;

x=(int*)malloc(50 * sizeof(int));

x=(int*)realloc(x,100); //allocated a new memory to variable **x**

Difference between malloc() and calloc()

| **calloc()** | **malloc()** |
|---|---|
| calloc() initializes the allocated memory with 0 value. | malloc() initializes the allocated memory with garbage values. |
| Number of arguments is 2 | Number of argument is 1 |
| **Syntax :** (cast_type *)calloc(blocks , size_of_block); | **Syntax :** (cast_type *)malloc(Size_in_bytes); |

## SUMMARY

- Here in this chapter a complex data type called Structure has been dealt with
- The C structure allows you to wrap related variables with different data types into a single entity that makes it easier to manipulate data in your program.
- Dynamic memory allocations mechanism and how to use the C built –in functions to allocate memory has been studied in this chapter.

# UNIT -5

## "C" PROGRAMMING

**Session Objectives:**

At the end of this session, the learner will be able to develop following programs :

- Program to find sum of series using while loop
- Program to find Factorial of N numbers using functions
- Program to swap the values of two variables
- Program to implement Ohms Law
- Program to find Resonant Frequency of RLC circuit
- Program to find equivalent resistance of three resistances connected in series and parallel
- Program to draw the symbol of NPN transistor using Graphics
- Program to draw the symbol of diode using Graphics

## C PROGRAMMING

### 5.1 Program to find sum of series using while loop

```
/* C Program to find Sum of N Numbers using While
Loop */

#include<stdio.h>
int main()
{
 int Number, i = 1, Sum = 0;

 printf("\nPlease Enter any Integer Value\n");
 scanf("%d", &Number);

 while(i <= Number)
 {
   Sum = Sum +
   i; i++;
 }

 printf("Sum of Natural Numbers = %d", Sum);
 return 0;
}
```

**Result:**

```
Please Enter any Integer Value

10

Sum of Natural Numbers=55
```

**Program to find Factorial of N numbers using functions**

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


void main()

{

int a,f;

int fact(int);

clrscr();

printf(“Enter a number: “);

scanf(“%d”,&a);

f=fact(a);

printf(“factorial=%d”,f);

getch();

}

int fact(int x)

{

int fac=1,i;

for(i=x;i>=1;i--)

fac=fac*i;

return(fac);

}
```

**Result:**

```
Enter a number: 5

factorial=120
```

**Program to swap the values of two variables**

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

{

int x,y,temp;                /*Declaration of variables*/

clrscr();

printf("\n Enter two numbers : ");

scanf("%d %d",&x,&y);            /*input statement*/

printf("\n x=%d \t y=%d",x,y);

temp=x;

x=y;

y=temp;          /*swapping contents of x&y */

printf("\nAfter Swapping \n x=%d \t y=%d",x,y);

getch();

}
```

**Result:**

```
Enter two numbers: 10 20

x=10 y=20

After Swapping

x=20 y=10
```

**5.2 Program to implement Ohms Law**

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

{

int ch;
float voltage , current , resistance ,
result; printf("Ohms law Program.\n");
printf("1. Calculate the voltage.\n");
printf("2. Calculate the current.\n");
printf("3. Calculate the resistance.\n");
printf("4.Exit \n");
scanf("%d",&ch);
switch(ch)

{

case 1:

printf("Enter the current in amps.\n");

scanf("%f",&current);

printf(“Enter the resistance in ohms.\n");

scanf("%f",&resistance);

result = current * resistance;

printf("The voltage is %0.2f volts.\n",result);

break;

case 2:

printf("Enter the voltage in volts.\n");

scanf("%f",&voltage);

printf(“Enter the resistance in ohms.\n");

scanf("%f",&resistance);

result = voltage / resistance;

printf("The current is %0.2f amps.\n",result);

break;
```

```
case 3:

printf("Enter the voltage in volts.\n");

scanf("%f",&voltage);

printf("Enter the current in amps.\n");

scanf("%f",&current);

result = voltage / current;

printf("The resistance is %0.2f ohms.\n",result);

break;

case 4: exit(0);

break;

default : printf("Invalid Choice \n");

break;

}

getch();

}
```

**Result:**

```
Ohms law program
1.Calculate the voltage
2.Calculate the current
3.Calculate the resistance
4.Exit

3
Enter the voltage in volts:
12
Enter the current in amps:
0.6
The resistance is 20.00 ohms
```

**Program to find Resonant Frequency of RLC circuit**

```
#include <stdio.h>

#include <math.h>

/* program to find the resonant frequency of an RLC circuit
*/

void main()

{

   double l;   /* Inductance in millihenrys */

   double c;   /* Capacitance in microfarads */

   double omega;  /* Resonance frequency in radians per
second */

   double f;   /* Resonance frequency in Hertz */

   printf("Enter the inductance in millihenrys: ");

   scanf("%lf", &l);

   printf("Enter the capacitance in microfarads:

   "); scanf("%lf", &c);

   omega = 1.0 / sqrt((l / 1000) * (c /

   1000000)); f = omega / (2 * M_PI);

   printf("Resonant frequency: %.2f\n", f);

  getch();

}
```

**Result:**

```
Enter the inductance in milli henrys: 0.1

Enter the capacitance in microfarads: 0.01

Resonant frequency: 159154.94
```

**Program to find equivalent resistance of three resistances connected in series and parallel**

```
#include<stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
float r1,r2,r3,result;
int ch;
printf("1.Equivalent Series resistance\n");
printf("2.Equivalent Parallel resistance\n" );
printf("3.Exit \n");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("Enter three resistor values: ");
scanf("%f %f %f",&r1,&r2,&r3);
result=r1+r2+r3;
printf("r1=%8.3f, r2=%8.3f and r3=%8.3f ohms\n",r1,r2,r3);
printf("Equivalent series resistance is %8.3f ohms",result);
break;
case 2:
printf("Enter three resistor values: ");
scanf("%f %f %f",&r1,&r2,&r3);
result=1/(1/r1+1/r2+1/r3);
printf("r1=%8.3f, r2=%8.3f and r3=%8.3f ohms\n",r1,r2,r3);
printf("Equivalent parallel resistance is %8.3f ohms\n",result);
break;
case 3: exit(0);
break;
default : printf("Invalid Choice \n");
break;
}
getch();
}
```

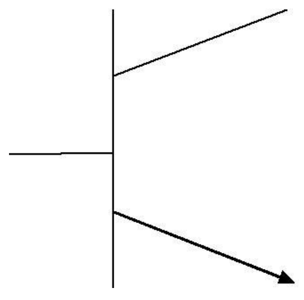
**Result:**

```
1.Equivalent Series resistance

2.Equivalent Parallel resistance

3.Exit

2

Enter three resistor values: 20 30 30

r1= 20.000, r2= 30.000 and r3= 30.000 ohms

Equivalent parallel resistance is  8.571 ohms
```

**Program to draw the symbol of NPN transistor using Graphics**

```
#include<stdio.h>

#include<conio.h>

#include<graphics.h>

void main()

{

  int gd = DETECT, gm;

  initgraph(&gd, &gm, "c:\\tc\\bgi");

  line(100, 100, 100, 200);

  line(70,  150,  100,  150);

  line(100,  125,  150,  90);

  line(100, 175, 150, 210);

  line(140, 190, 150, 210);

  line(130, 210, 150, 210);

  outtextxy(100, 250, "NPN Transistor");

  getch();

  closegraph();
}
```

**Result:**

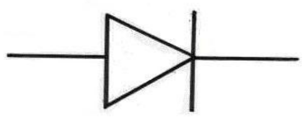


NPN Transistor

**Program to draw the symbol of diode using Graphics**

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

void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"c:\\tc\\bgi");

line(100,100,100,200);
line(70,150,100,150);

line(100,100,150,150);
line(100,200,150,150);

line(150,100,150,200);

line(150,150,250,150);

getch();
closegraph();
}
```

**Result:**



**DIODE**

**Summary:**

In this session, we learnt to develop following programs:

- Program to find sum of series using while loop
- Program to find Factorial of N numbers using functions
- Program to swap the values of two variables
- Program to implement Ohms Law
- Program to find Resonant Frequency of RLC circuit
- Program to find equivalent resistance of three resistances connected in series and parallel
- Program to draw the symbol of NPN transistor using Graphics
- Program to draw the symbol of diode using Graphics