UNIT II

Problem formulation & Problem Solving

Effective problem formulation is fundamental success of all analysis, particularly in command-and-control assessment because the problems are often ill defined and complex, involving many dimensions and rich contents. Problem formulation involves decomposition of the analytic problems into appropriate dimensions such as structures, functions and mission areas.

Problem formulation is an interactive process that evolves over the course of study. Itis essential even for small studies as where time is short, It will save time later help ensure quality.

The problem formulation phase should identify the context of the study and aspects of the problem related issues. There is no universal acceptance approach to problem formulation. However, practices exist that can be applied. First find out what the question is then find out what the real question is.

Problem Solving:

When we start reading these and wants to learn how to solve a problem by using computers, it is first of all important to understand what the problem is. We need to read all the problem statements a number of times to ensure that is understands what is asked before attempting to solve t problem.

Method of problem solving:

1. Recognize and understand the problems
2. Accumulate facts
3. Select appropriate theory
4. Make necessary assumptions

5. Solve the problems
6. Verify the results

Performing step 5 solves the problem may involve a computer.The 5 steps in using a computer as a problem-solving tool

1. Develop an algorithm and flowchart
2. Write a program in computer language
3. Enter the program in to computer
4. Test and debug the program
5. Run the program, input data, and get the results from computer.

C-INTRODUTION

- Communicating with a computer involves the language the computer understands
- Which immediately rules out English as the language of communication with computers
- C is one of the most popular programming languages.

History of 'C'

- C is developed by '**DENNIS RITCHE'** at **AT & T Bell Laboratories** at USA in **1972.**
- It is the upgraded version of two languages called BCPL and B which were developed at bell laboratories.
- It is seamed to abstract too general another language Called Basic Computer Programming Language (BCPL)was developed by martin Richards at Cambridge university with some additional features than CPL.
- At the same time, a language called B was developed by Ken Thomsonat AT & T Bell Labs.
- But like BCPL and B turned out to be very specific, Dennis Ritche developed a language with some additional features of BCPL and B which is very simple, relatively good programming efficiency and relatively good machine efficiency called 'C' language.
- Consequently, the ANSI has begun to work on a standardized definition of the 'C' Language to make it still powerful.

Features of 'C':

- C is a general-purpose language
- C is a structural Language
- C is middle level language ie., it supports both the low- and high-level languagefeatures
- C is flexible and more powerful language
- C programs are fast and efficient
- C is most suitable for writing system software as well as application software's
- Machine independent and portable
- C has the ability to extend itself, we can continuously add our own functions to the existing library functions
- C is the robust language
- C is widely available, commercial C compilers are available on most PC's
- Commands may be inserted anywhere in a program
- C has rich set of operators.
- C language allows reference to memory location with the help of pointers, which holds the address of the memory locations

STRUCTURE OF C PROGRAM

DOCUMENTATION SECTION
PREPROCESSOR SECTION
DEFINITION SECTION
GLOBAL DECLARATION SECTION

main()
{
DECLA
RAION PART
EXECUTION
PART
}

```
Sub program section
{       Body of the sub program
}
```

i. **Documentation Section:**

It consists of set of command lines used to specify the name of the program, the author of the program and other details etc..

Comments:

Comments are very helpful in identifying the program features and underlying logi of the program. The lines with '/*' and ending with '*/' are known as comment lines. These are not executable; the compiler is ignored anything in between /* and */

ii. **Preprocessor Section:**

It is used to link system library files, for defining the macros and for defining the conditional inclusion.

Ex : include <stdio.h>

iii. **DefintionSection:**

The definition section defines all symbolic constants.

Ex: # define pi 3.14

iv. **Global Declaration Section:**

The variable that are used in more than one function throughout the program are called global variable and are declared outside of all the function. Ie.m main () function.

v. **Main Function:**

Every C program must have one main function, which specify the starting of C program

Declaration Part:

This part is used to declare all the variables that are used in the executable part of the program and these are called local variables

Executable Part:

- It contains at least one valid C statements
- The execution of a program begins with opening brace '{'and ends with '}'

Rules for writing C program:

- All the statements should be in lower case letters.
- Upper case letters are only used for symbolic constants.
- Blank spaces may be inserted between two words. It is not used when declaring variables, keywords, constants and functions.
- The program statements can write anywhere between the two braces following the declaration part
- The user can also write one or more statements in one line separating them with semicolon (;)

Executing C Program

Execution is the process of running the program, to execute a 'C' program, we need to follow the steps given below.

i. Create the program
ii. Compiling the program
iii. Linking the program with system library
iv. Executing the program

i. Creating the program:

Creating the program means entering and editing the program in statndard C editors and save the program with an extension .c

ii. Compiling the program

- This is the process of converting the high-level language program into machine understandable form. For this purpose, compiler is used. Usually this can be done in C language by pressing ALT+F9 or compile from compile menu
- Here there are possibility to show errors ie., syntax errors, means the statements written in program are not in proper syntax

iii. Linking the program with system library

- C language program is the collection of predicted functions.
- These functions are already written in some standard C header files,
- Therefore, before executing a C program, we need to link system library
- This can be done automatically at the time of execution

iv. Executing the program:

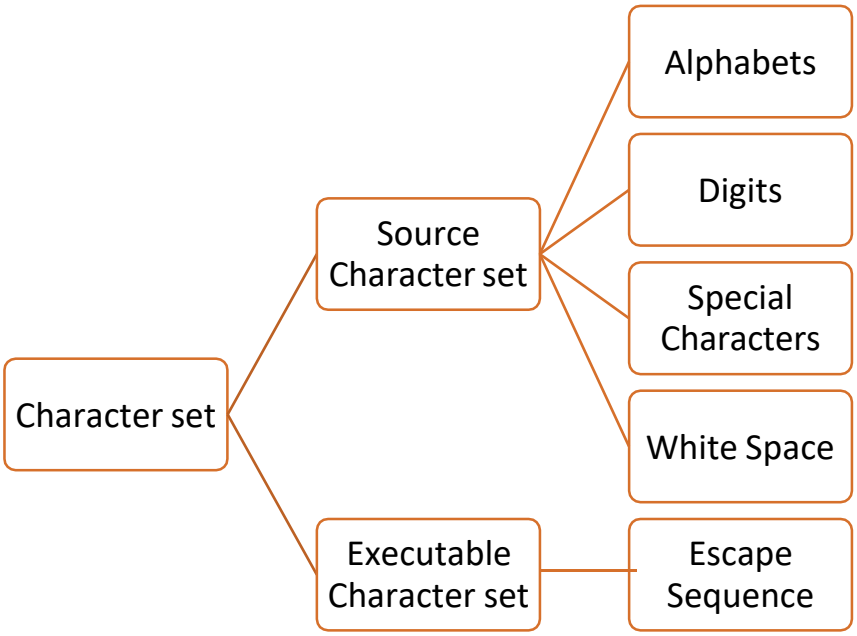
- This is the process of running and testing the program with the sample data
- At this time there is a possibility show two type of errors given below
  - Logical Error:
    - These are the errors, in which conditional and control statements cannot end their match after some sequential execution
  - Data error:
    - These are the errors, in which the input data given, if not in a proper syntax as specified in input statements.

CHARACTER SET

The character set is the fundamental raw material of any language and they are used to represent information. ***The set of characters used in a language is knownas its character set.*** These characters can be represented in the computers.

C programs are basically of two types, namely

1. Source Character set
2. Executable Character set



1. **Source character set**

They are used to construct the statements in the source programs

Alphabets	-	A to Z,a to z.
Decimal Digits	-	0 to 9
White Spaces	-	Blank space, horizontal tab, vertical tab, new line
Special characters	-	- +,*,, ; ,', /,?,[,{,@,#,%,&,(,<,=,>,},],_,-.

Trigraph Characters:

??= -> # ??(-> [??) ->] ??< -> { ??> -> } ??! -> |

2. **Executable character set**

\a – Beep \b – Back space \t – Horizantal tab

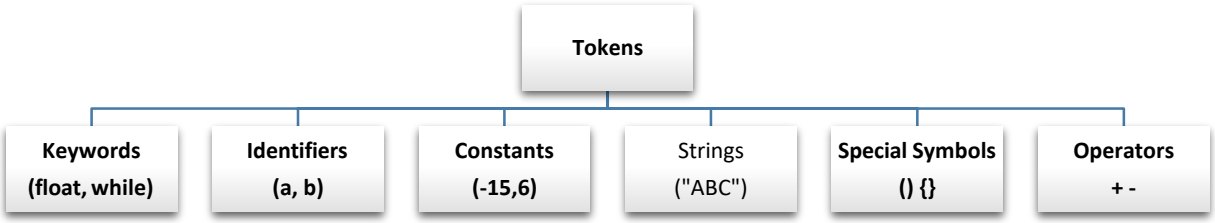
\\ - Back slash \0 – Null \n – next line

\v –vertical tab \f – form feed \r – carriage return

\' – Single Quote \"- Double quote

TOKENS

The tokens are usually referred as individual text and punctuation in a passage of text. C tokens has following types.



Identifiers and Keywords

In C Language every world is classified into either a keyword or an identifier.

Identifiers:

- Identifiers are names given to various program elements, such as a variable, functions and arrays etc..
- Identifiers are user defined names
- It consists of sequence of letters and digits.

Example of valid Identifiers:

```
Length, Area, volume, SuM,_Average
```

Invalid identifiers:

```
Length of line, Stum: , Year's, 2a.,
```

Rules for writing Identifiers:

1. It contains letters and digits.
2. '_' can also be used.
3. First character must be a letter or _
4. Contain only 31 characters.
5. No space and special symbols are allowed.
6. It cannot be a keyword.

KEYWORD

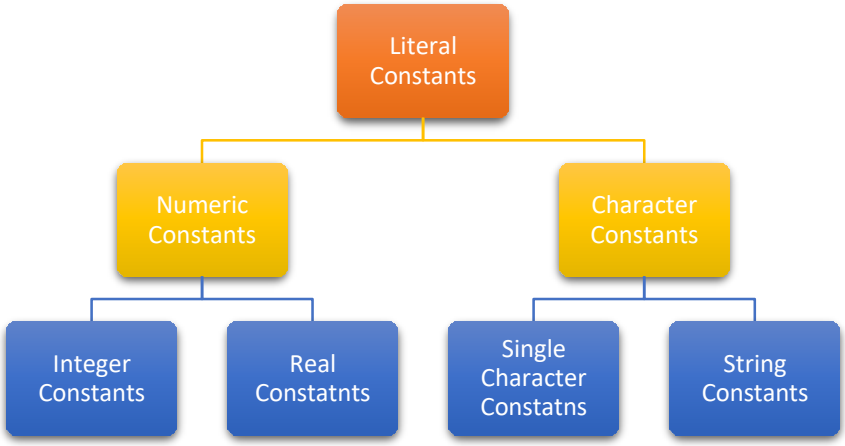
- These are reserved words that have standard and predefined meaning in C language.
- It cannot be changed.
- They can't be used as a variable name.
- For utilizing the keyword in a program, no header files are included.

- The C support 32 keywords

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

CONSTANTS

- The item whose values cannot be changed during the execution of program called constants.
- Three types of constants:
  - Literal Constants
    - A literal constant is a value that you put directly in your code for example
  - Symbolic Constants
    - A symbolic constant is a constant that has a name.
    - Example: #define PI 3.14
  - Qualifier Constant:
    - The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed
    - Example: const float pi=3.14;
- Several types of literal constants available.



Numeric constants:

Integer Constants:

- The constants are represented with whole numbers
- They require a minimum of 2 bytes and a maximum of 4 byte of memory

Rules for Constructing Integer Constants

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive.
- No commas or blanks are allowed within an integer constant.
- The allowable range for integer constants is -32768to 32767

Valid Examples: 426 +782 -8000 -7605
Invalid example: 2.3, .235, $76, 3,600

Real Constants:

- Real Constants are often known as floating point constants
- Real Constants can be represented in exponential form or floating-point form.

Rules for constructing Real Constants:

- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within a real constant.

Ex.: +325.34 426.0 -32.76 -48.5792

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large.

In exponential form of representation is as follows :

MANTISSA e EXPONENT.

Rules for constructing Exponential form:

- The mantissa part and the exponential part should be separated by a letter e.
- The mantissa part may have a positive or negative sign.
- Default sign of mantissa part is positive.
- The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

Range of real constants expressed in exponential form is

-3.4e38 to 3.4e38.

Ex.: +3.2e-5 4.1e8 -0.2e+3 -3.2e-5

Character Constant:

Single Character Constants:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas (‘’).
- The maximum length of a character constant can be 1 character

Ex.: 'A' 'I' '5' '=

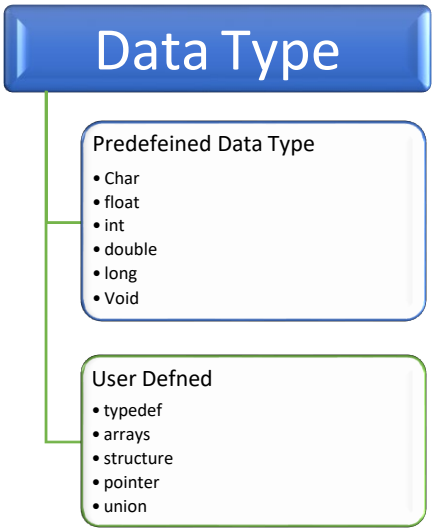
String Constant:

- String Constants are sequence of characters within double quote marks (“”)
- The string may be combination of all kinds of symbols

Ex.: “Hello”, “India”, “444”, “a”, “”

DATA TYPES

- Data type is the type of data going to be process within the program.
- C supports different data types have predefined Memory requirement.
- Generally data is represented using numbers or character



Integer Datatype:

(i) **Short Integer:**

- It occupies 2 bytes of memory
- Range is from -32768 to 32767
- Program runs faster
- Format specifier is %d or %c

- Example: int a=2; short int a=3;

(ii) **Long Integer:**

- It occupies 4 bytes of memory
- Range -2147483648 to -2147483647
- Program runs slower
- Format specifier %ld
- Example: long int a=12;
- When a variable is declared without a short or long keyword the default is "short signed int"

(iii) **Signed Integer:**

- It occupies 2 bytes of memory
- Range -32768 to 32767
- Format specifier is %d or %c
- By default signed int is "short signed int "
- Long signed integer occupies 4 bytes of memory
- Example: signed int a=-2;

(iv) **Unsigned unteger:**

- It occupies 2 bytes of memory
- Range 0 to 65535
- Format specifier is %u
- By default unsigned int is "short unsigned int"
- Long unsigned integer occupies 4 bytes of memory
- Example: unsigned long int b=45;

Character datatype:

(i) **Signed character:**

- It occupies 1 byte of memory
- Range is from -128to -127
- Format specifier is %c
- When printed using %d control string corresponding ASCII number is printed
- Example: char ch='a';

(ii) **Unsigned character:**

- It occupies 1 byte of memory
- Range is from 0 to 255
- format specifier is %c
- When printed using %d control string corresponding ASCII number is printed
- Example: unsigned char ch='a';

Float Data type:

- It occupies 4 bytes of memory
- Range -3.4e-38 to 3.4e+38

- Format specifier is %f
- Example: float f=3.14;

Double data type:

- It occupies 8 bytes of memory
- Range 1.7e-308 to 1.7e+308
- Format specifier is %lf
- Example: double d=7.86;
- Also long double range is 3.4e-4932 to 3.4e+4932(4 bytes of memory)
- Example: long double k=9.6;

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

VARIABLES

- Variable names are names given to locations in memory.
- These locations can contain integer, real or character constants.
- The value of the variable can be changed during runtime.

Rules for constructing variable name:

- The first character in the variable name must be an alphabet or underscore.
- No commas or blanks are allowed within a variable name.
- No special symbol other than an underscore (_) can be used in a variable name.
- A variable name is any combination of 1 to 31 alphabets, digits or underscores

Ex.: si_int, m_hra, pop_e_89

DECLARING VARIABLES

- The declaration of variables should be done in the declaration part of the program
- The variable must be declared before they are used in the program
- Declaration provide two things
  - Compiler obtain variable name
  - Compile allocate memory for variable according to the data type

Syntax:

```
Data_Type variable name;
```

Example:

```
Int  age;
Char
m; Float
s;
```

Initializing variables:

- Variables declared can be assigned or initialized using an assignment operator =
- The declaration and initialization can also be done in the same line

Syntax:

```
Variable_name = constant;
Or
Data_type variable_name = constant;
```

Example:

```
Y=2;
Int x=15;
Char
ch='c';
```

Dynamic initialization:

- The initialization of variable at run time us called dynamic initialization
- The C initialization can be done at any place in the program

Example:

Swapping of two number

```
#include<stdio.h>
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter two numbers");
    scanf("%d%d",&a,&b);
    c=a;
    a=b;
    b=c;
    printf("After swapping value a=%d, b=%d",a,b);
}
```

Constant variable:

- The constant variables are used to remain unchanged value during the execution of the program.
- It can be done only by declaring the variable as a constant

Syntax:

```
const data_type variable_name=value;
```

Example:

const int m=10;

Volatile variable:

- The volatile variables are those variables that are changed at any time by other external program
- Keyword : volatile

Syntax:

```
volatile  data_type variable_name
```

Example:

volatile int d;

OPERATORS AND EXPRESSIONS

- An **Operator** is a symbol that specifies an operation to be performed on the operands
- The data items that operators acts upon are called **Operands**

- An **Operation** indicates an operation to be performed on data that may yield a new value
- An operator can operate on integer, character and floating point numbers

Types of operator:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment and decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

Arithmetic Operators:

These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus

Operator	*Operation*	*Example*
+	*Addition*	*2+2=4*
-	*Subtraction*	*2-2=0*
*	*Multiplication*	*2*2=4*
/	*Division*	*2/2=1*
%	*Modulo Division*	*2%2=0*

Example:

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

```
void main()
{
        int a=40,b=20, add,sub,mul,div,mod;
        add = a+b;
        sub = a-b;
        mul = a*b;
        div = a/b;
        mod = a%b;
        printf("Addition of a, b is : %d\n", add);
        printf("Subtraction of a, b is : %d\n", sub);
        printf("Multiplication of a, b is : %d\n",
        mul);printf("Division of a, b is : %d\n", div);
        printf("Modulus of a, b is : %d\n", mod);
}
```

Output:

```
Addition of a, b is : 60
Subtraction of a, b is : 20
Multiplication of a, b is :
800Division of a, b is : 2
Modulus of a, b is : 0
```

Relational operators:

- These operators are used to compare the value of two variables.
- These operator provide the relationship between two expressions
- If the relation is true it returns a value 1, else it returns a value 0

S.no	Operators	Example	Description
1	>	x > y	x is greater than y
2	<	x < y	x is less than y
3	>=	x >= y	x is greater than or equal to y
4	<=	x <= y	x is less than or equal to y
5	==	x == y	x is equal to y
6	!=	x != y	x is not equal to y

Example :

```
#include <stdio.h>
void main()
{
        int m=40,n=20;
        if (m == n)
        {
                printf("m and n are equal");
```

```
                }
                else
                {
                        printf("m and n are not equal");
                }
        }
```

Output: m and n are not equal

Logical Operators:

- These operators are used to perform logical operations on the given expressions.
- The result may be either 1 or 0.
- There are 3 logical operators in C language. They are, logical AND (&&), logical OR(||) and logical NOT (!).

S.no	Operators	Name	Example	Description
1	&&	logical AND	(x>5)&&(y<5)	It returns true when both conditions are true
2	||	logical OR	(x>=10)||(y>=10)	It returns true when at-least one of the condition is true
3	!	logica lNOT	!((x>5)&&(y<5))	It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

Example:

```
#include
<stdio.h>void
main()
{
        int a, b, c;
        printf("Enter a,b,c: ");
        scanf("%d %d %d", &a, &b,
        &c);if (a > b && a > c)
       {
            printf("a is Greater than b and c");
        }
         else if (b > a && b > c)
        {
            printf("b is Greater than a and c");
         }
        else if (c > a && c > b)
```

```
        {
            printf("c is Greater than a and b");
        }
        else
        {
            printf("all are equal or any two values are equal");
        }
}
```

Output:

Enter a,b,c : 3 5 8
C is greater than a and b

Assignment operators:

- It is used to assign the result of an expression to a variable
- The equal (=) sign is used as an assignment operator

Operators		Example	Explanation
Simple assignment operator	=	sum = 10	10 is assigned to variable sum
Compound assignment operators / Shorthand Assignment operators	+=	sum += 10	This is same as sum = sum + 10
	-=	sum -= 10	This is same as sum = sum – 10
	*=	sum *= 10	This is same as sum = sum * 10
	/+	sum /= 10	This is same as sum = sum / 10
	%=	sum %= 10	This is same as sum = sum % 10
	&=	sum&=10	This is same as sum = sum & 10
	^=	sum ^= 10	This is same as sum = sum ^ 10

Example:

```
# include <stdio.h>
int main()
{
int Total=0,i;
for(i=0;i<10;i+
+)
{
Total+=i; // This is same as Total = Toatal+i
}
printf("Total = %d", Total);
}
```

Output:

Total = 45

Bitwise Operators:

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q (and)	p | q (or)	p ^ q (xor)
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as

follows:A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C language are listed in the following table.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100

|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101

^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Example:

```
#include <stdio.h>
void main()
{
        int a = 60;     /* 60 = 0011 1100 */
        int b = 13;     /* 13 = 0000 1101 */
        int c = 0;

        c = a & b;      /* 12 = 0000 1100 */
        printf("Line 1 - Value of c is %d\n", c
        );

        c = a | b;      /* 61 = 0011 1101 */
        printf("Line 2 - Value of c is %d\n", c
        );

        c = a ^ b;      /* 49 = 0011 0001 */
        printf("Line 3 - Value of c is %d\n", c
        );

        c = ~a;         /*-61 = 1100 0011 */
        printf("Line 4 - Value of c is %d\n", c
        );

        c = a << 2;     /* 240 = 1111 0000 */
        printf("Line 5 - Value of c is %d\n", c );

        c = a >> 2;     /* 15 = 0000 1111 */
        printf("Line 6 - Value of c is %d\n", c );
}
```

Conditional or ternary operators:

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

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

- In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

Example:

```
#include <stdio.h>
void main()
{
        int x=1, y ;
        y = ( x ==1 ? 2 : 0 ) ;
        printf("x value is %d\n", x);
        printf("y value is %d", y);
}
```

Output:

```
x value is 1
y value is 2
```

Increment & Decrement Operators:

- Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

S.no	Operator type	Operator	Description
1	Pre increment	++i	Value of i is incremented before assigning it to variable i.
2	Post-increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre decrement	– –i	Value of i is decremented before assigning it to variable i.
4	Post_decrement	i– –	Value of i is decremented after assigning it to variable i.

Syntax:

Increment operator: ++var_name; (or) var_name++;

Decrement operator: – -var_name; (or) var_name – -;

Example:

Increment operator : ++ i ; i ++ ;

Decrement operator : - – i ; i – - ;

```
nclude <stdio.h>
id main()
```



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

Output: 1 2 3 4 5 6 7 8 9

Special Operators:

Below are some of special operators that C language offers.

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

Example:

```
#include <stdio.h>
void main()
{
  int a = 4;
  float b=6.7;
  printf("Size of variable a = %d\n", sizeof(a)
  );printf("Size of variable b = %f\n", sizeof(b)
  );
}
```

Output:
Size of variable a =
2Size of variable b =
4

Operators Precedence:

- Operator precedence determines the grouping of terms in an expression.
- This affects how an expression is evaluated.
- Certain operators have higher precedence than others;
- For example, the * operator has higher precedence than the + operator.

Example:

x = 7 + 3 * 2;

- Here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

- Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

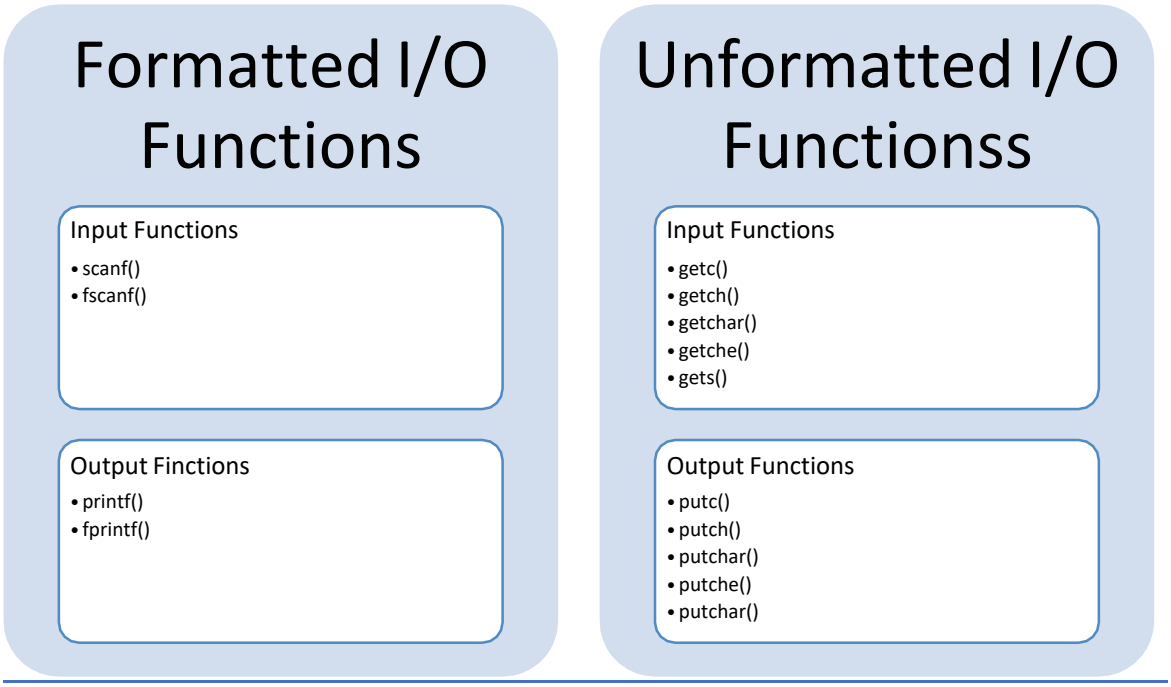
- Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

MANAGING INPUT AND OUTPUT OPERATIONS

- Reading data from input device, processing it, and displaying the result on the screen are the three task of any program
- We have two methods for providing data to the program
    1. Assigning the data to the variable in a program
    2. By using the I/O functions

Two types of Input / Output functions:



Formatted I/O Functions

The printf() function:

- The output data or result of an operation can be displayed from the computer to a standarad output device ie., Monitor
- The function is used to output any combination of data

Syntax:

```
printf ( "format string", list of variables ) ;
```

The format string can contain:

a) Characters that are simply printed as they are.
b) Conversion specifications that begin with a % sign
c) Escape sequences that begin with a \ sign

Example:

```
main( )
{
        int avg = 346 ;
        float per = 69.2 ;
        printf ( "Average = %d\nPercentage = %f", avg, per ) ;
}
```

Output:

```
Average = 346
Percentage = 69.200000
```

Format Specifications:

Data Type		Format Specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	Float	%f
	Double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

Optional Specifiers:

Specifier	Description
dd	Digits specifying field width
.	Decimal point separating field width from precision (precision stands for the number of places after the decimal point)
.dd	Digits specifying precision
-	Minus sign for left justifying the output in the specified field width

Example:

```
main( )
{
        int weight = 63 ;
        printf ( "\nweight is %d kg", weight ) ;
        printf ( "\nweight is %2d kg", weight ) ;
        printf ( "\nweight is %4d kg", weight ) ;
        printf ( "\nweight is %6d kg", weight ) ;
        printf ( "\nweight is %-6d kg", weight ) ;
}
```

Output:

```
Columns    01234567890123456789012345678901234567890
           weight is 63 kg
           weight is 63 kg
           weight is 63  kg
           weight is 63 kg
           weight is 63 kg
```

```
main( )
{
        printf ( "\n%10.1f %10.1f %10.1f", 5.0, 13.5, 133.9 ) ;
        printf ( "\n%10.1f %10.1f %10.1f", 305.0, 1200.9, 3005.3
}       );
```

Output:

```
012345678901234567890123456789012
         5.0        13.5       133.9
       305.0      1200.9      3005.3
```

Escape Sequences :

\a – Beep	\b – Back space	\t – Horizantal tab
\\ - Back slash	\n – next line	\v –vertical tab
\f – form feed	\r – carriage return	\' – Single Quote
\"- Double quote		

The Scanf() Function:

- The scanf() function is usd to read information the standarad input device
  - It is used for runtime assignment of variables
  - This function is used to enter any combination of input

Syntax:

```
scanf ( "format string", list of addresses of variables ) ;
```

Example:

```
scanf ( "%d %f %c", &c, &a, &ch ) ;
```

- & denotes the address of the variable. The values received from keyboard must be dropped into variables corresponding to these addresses

The sprintf() function:

- This function writes the output to an array of characters

Example:

```
int i = 10 ;
 char ch = 'A'
 ;
 float a = 3.14 ;sprintf ( str, "%d %c %f", i, ch, a ) ;
```

UNFORMATTED I/O FUNCTIONS

- These statements are used to I/O a single / group of characters from the I/O Device
- Here the user can't specify the type of data that is going to be Input / Ouptput

UNFORMATTED OUTPUT FUNCIONS:

1) The getch () function:

- getch() accepts only single character from keyboard.
- The character entered through getch() is not displayed in the screen (monitor).

Syntax:

```
variable_name = getch();
```

2) The getche() function:

- getche() also accepts only single character, but unlike getch(), getche() displays the entered character in the screen.

Syntax:

```
variable_name = getche();
```

3) The getchar() function:

- getchar() accepts one character type data from the keyboard.
- It requires ***Enter key*** to be typed following the character that you typed

Example:

```
void main( )
{
    char ch ;
    printf ( "\nPress any key to continue" ) ;
    getch( ) ; /* will not echo the character */
    printf ( "\nType any character" ) ;
    ch = getche( ) ; /* will echo the character typed */
    printf ( "\nType any character" ) ;
    getchar( ) ; /* will echo character, must be followed by enter key */
}
```

Output:

```
Press any key to continue
```

Type any character B

Type any character W

W

4) **The gets() function:**
   - It accepts any line of string including spaces from the standard Input device (keyboard).
   - It stops reading character from keyboard only when the enter key is pressed.

Syntax:

```
gets(variable_name);
```

Example:

```
char ch[20];
gets(ch);
```

Unformatted Output Functions:

1) **The putch() Function:**
   - putch displays any alphanumeric characters to the standard output device.
   - It displays only one character at a time.

Syntax :

```
putch(variable_name);
```

Example:

```
char
z[20]=”welcome”;
putch(z);
```

2) **The putchar() function:**
   - putchar displays one character at a time to the Monitor.

Syntax:

```
putchar(variable_name);
```

Example:

```
char
z[20]=”welcome”;
putchar(z);
```

3) The puts() Function

- puts displays a single / paragraph of text to the standard output device.

Syntax:

```
Puts(variable_name);
```

Example:

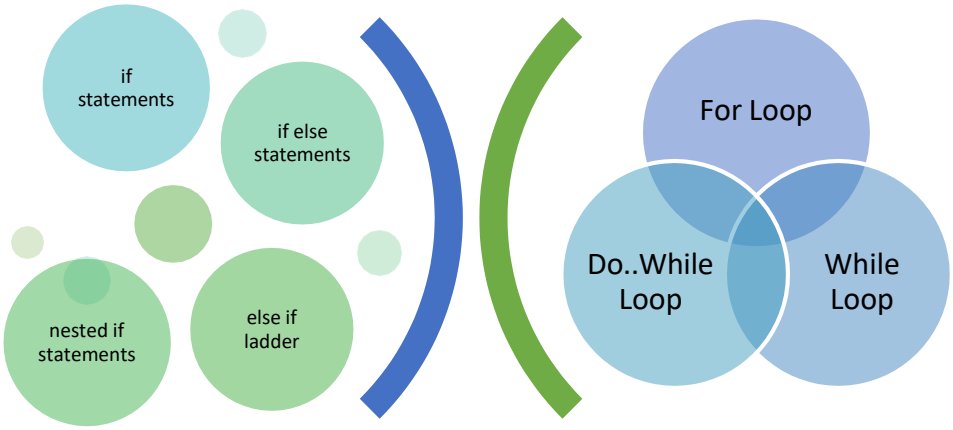
```
#include<stdio.h>
#include<conio.h
>void main()
{
        char a[20];
        gets(a);
        puts(a);
        getch();
}
```

Output:

```
   Abcd
efghAbcd
   efgh
```

CONTROL STATEMENTS

- Control Statement are program statements that are cause a jump of control from one part of program to another part of program
- These statements are classified into two types:
    - Branching Statements
    - Looping Statements



Branching Looping

Branching Statements:

- In decision making statements, group of statements are executed when condition is true. If condition is false, then else part statements are executed.
- There are 3 types of decision making control statements in C language. They are,
    - if statements

- o if else statements
- o nested if statements

- o else if statements
- o Switch statements

If Statement:

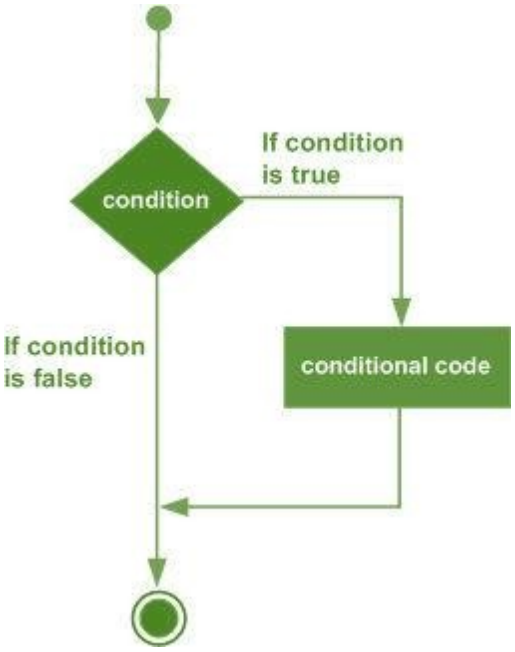
- An if statement consists of a boolean expression followed by one or more statements.

Syntax:

```
if(condition)
{
        Statements;
}
```

- If the condition is true, then the block of code inside the if statement will be executed.
- If the condition is false, then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
   int a = 10;
  if( a < 20 )
  {
      printf("a is less than 20\n" );
  }
  printf("value of a is : %d\n", a);

}
```

Output:

```
a is less than 20;
value of a is : 10
```

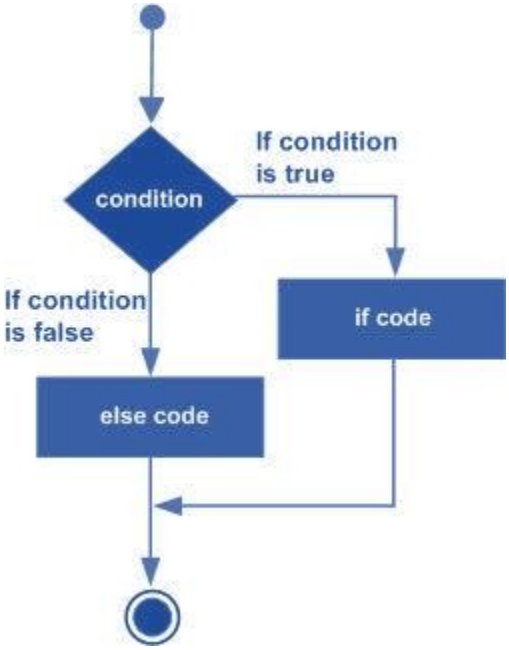
if..else statements:

- In if else control statement, group of statements are executed when condition is true.
- If condition is false, then else part statements are executed.

Syntax:

```
if(condition)
{
  True Statements
}
else
{
        False statements
}
```

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
        int a = 100;
        if( a < 20 )
         {
            printf("a is less than 20\n" );
         }
         else
         {
```

```
            printf("a is not less than 20\n" );
        }
        printf("value of a is : %d\n", a);
}
```

Output:

```
a is not less than 20;

value of a is : 100
```

else if ladder / else if statements:

- An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

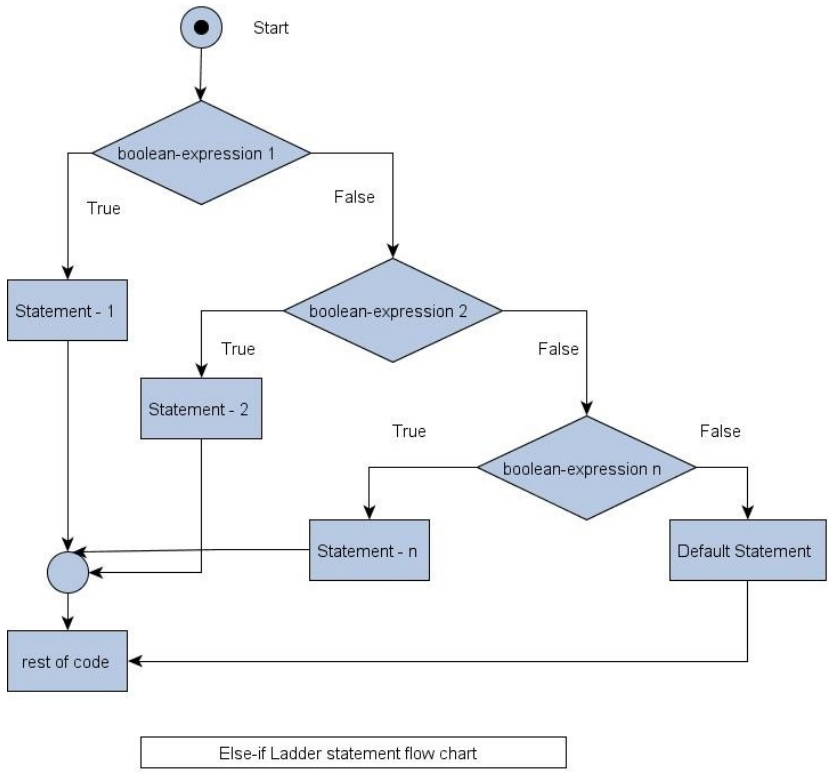
When using if , else if , else statements there are few points to keep in mind:

- o An if can have zero or one else's and it must come after any else if's.
- o An if can have zero to many else if's and they must come before the else.
- o Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

```
if(Condition1)
{
        Statements 1;
}
else if(Condition 2)
{
        Statements 2;
}
else if(Condition 3)
{
        Statements 4;
}
else
{
        Else statements;
}
```

Flow Diagram:



Else-if Ladder statement flow chart

Example:

```
#include
<stdio.h>void
main ()
{
        int a = 100;
         if( a == 10 )
        {
           printf("Value of a is 10\n" );
        }
        else if( a == 20 )
        {
            printf("Value of a is 20\n" );
        }
        else if( a == 30 )
        {
            printf("Value of a is 30\n" );
        }
        else
        {
            printf("None of the values is matching\n" );
        }
        printf("Exact value of a is: %d\n", a );

}
```

Output:

```
None of the values is matching
Exact value of a is: 100
```

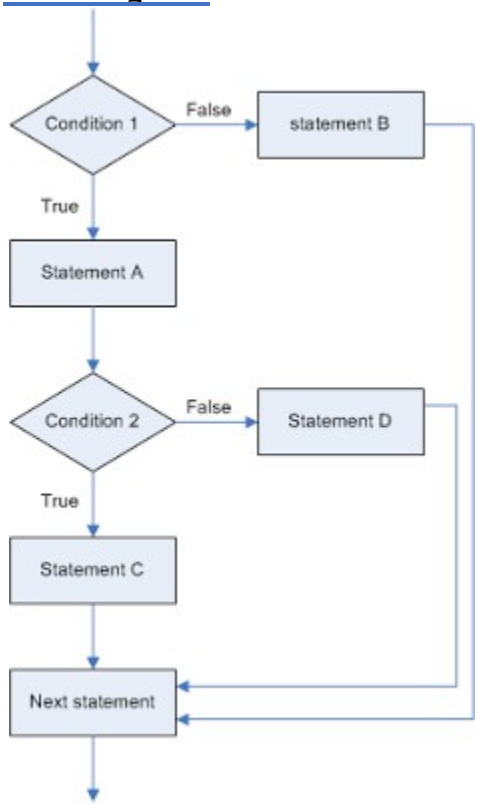
Nested if Statements:

- An if .. else statement is placed inside another if..else statements, this is known as nested if..else statements
- These are used when series of decisions are involved

Syntax:

```
if( Condition 1)
{
  /* Executes when the condition 1 is true */
  if(Condition 2)
  {
    /* Executes when the condition 2 is true */
  }
}
else
{
/* Executes when the condition  1 is fale */
}
```

Flow Diagram:



Example:

#include
<stdio.h>void

main ()

```
{
        int a = 100;
        int b = 200;
        if( a == 100 )
         {
                if( b == 200 )
                {
                  printf("Value of a is 100 and b is 200\n" );
                }
         }
         printf("Exact value of a is : %d\n", a );
         printf("Exact value of b is : %d\n", b );
}
```

Output :

```
Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200
```

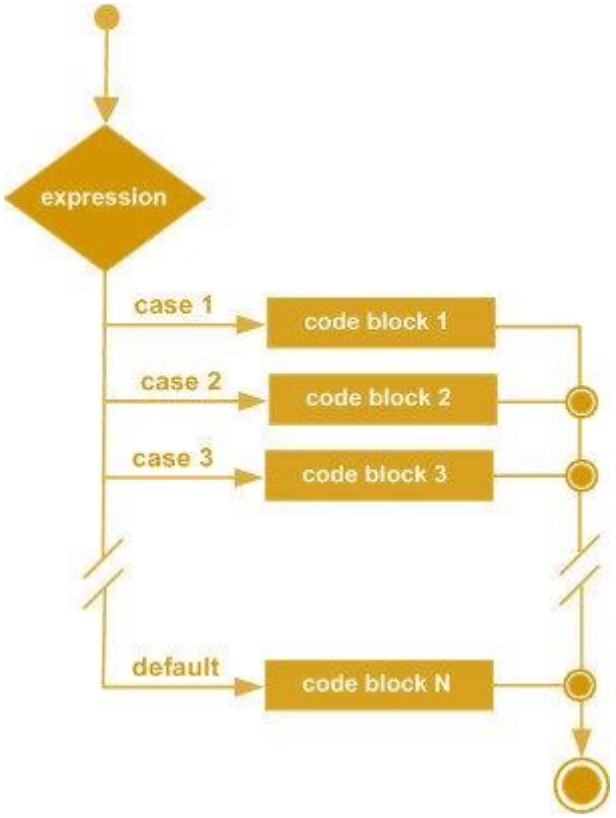
Switch Statements:

- Switch statements is a multiway branching statements based on the value of an expression
- The control is transferred to one of the many possible points
- If no match is there, then the default block is executed
- Every case statements should be terminated with a break statements

Syntax:

```
switch (expression)
{
case label1:
        statements;
        break;
    case label2:
        statements;
        break;
    default:
        statements;
    }
```

Flow Diagram:



Example:

```
#include <stdio.h>
void main ()
{
        char grade = 'B';
        switch(grade)
         {
                case 'A'
                :       printf("Excellent!\n" );
                        break;

                case 'B' :
                case 'C' : printf("Well done\n" );
                        break;

                case 'D' : printf("You passed\n" );
                        break;

                case 'F' : printf("Better try again\n" );
                        break;

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

        }
        printf("Your grade is  %c\n", grade );
```

}

Output:

Well done
Your grade is
B

LOOPING STATEMENTS

- Loop control statements in C are used to perform looping operations until the given condition is true. Control comes out of the loop statements once condition becomes false.
- There are 3 types
  - for loop
  - while loop
  - do while loop

for loop:

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

Syntax:

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

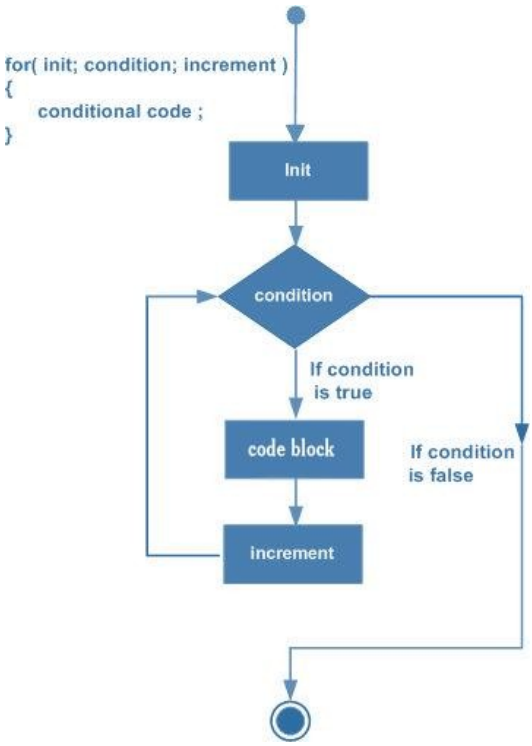
1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
  int a;
              for( a = 10; a < 20; a = a + 1 )
             {
               printf("%d", a);
             }
 }
```

Output:

10 11 12 13 14 15 16 17 18 19

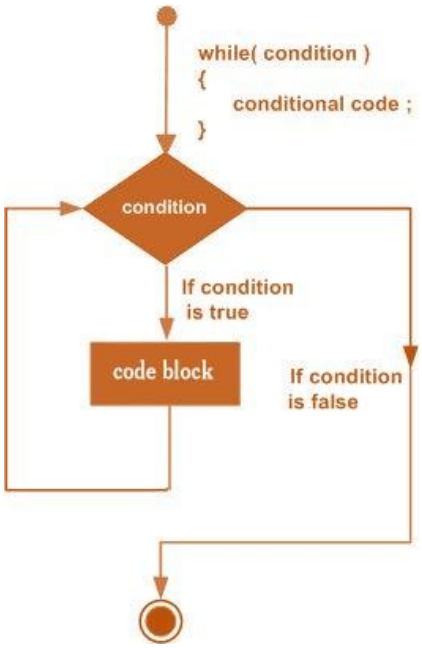
While Loop:

- A while loop statement repeatedly executes a target statement as long as a given condition is true.
- Here the condition is checked first, so it is also called as entry control statements

Syntax:

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

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
   int a = 10;
            while( a < 20 )
            {
                  printf(" %d",
                  a);a++;
            }
}
```

Output:

10 11 12 13 14 15 16 17 18 19

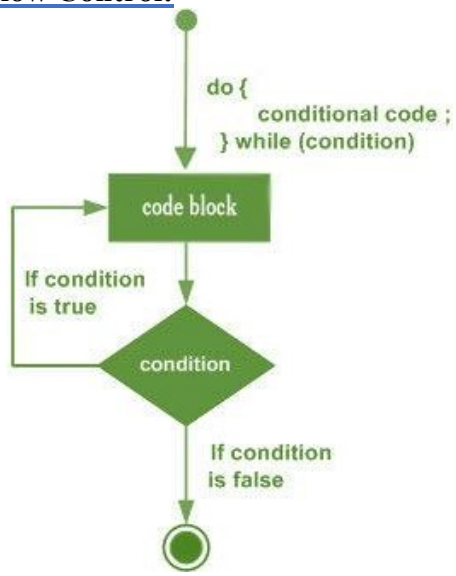
Do…while Loop:

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

Syntax:

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

Flow Control:



Example:

```
#include
<stdio.h>void
main ()
{
   int a = 10;
          do
          {
             printf("%d",
             a);a = a + 1;
          }while( a < 20 );
}
```

Output
:

10 11 12 13 14 15 16 17 18 19

Break Statement:

The **break** statement has the following two usages:

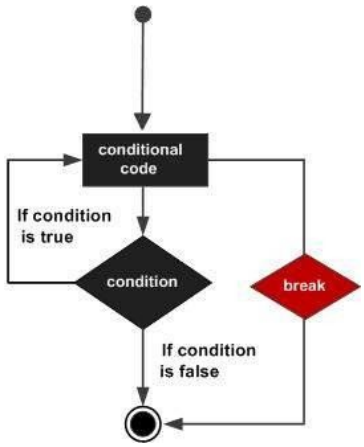
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

- It can be used to terminate a case in the **switch** statement

Syntax:

```
break;
```

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
   int a = 10;
        while( a < 20 )
          {
            printf("%d",
            a);a++;
                    if( a > 15)
                    {
                       break;
                    }
        }
}
```

Output:

10 11 12 13 14 15

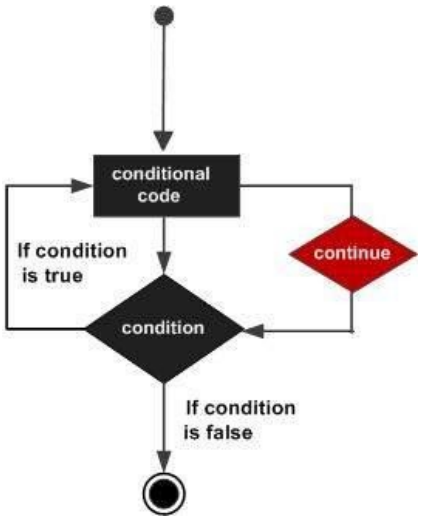
Continue Statement:

- The continue statement works somewhat like the break statement.
- Instead of forcing termination of loop, however, continue forces the next iteration of the loop to take place, skipping any code in between.

Syntax:

```
continue;
```

Flow Diagram:



Example:

```
#include
<stdio.h>void
main ()
{
   int a = 10;
        while( a < 20 )
          {
            printf("%d",
            a);a++;
                    if( a > 15)
                    {
                       continue;
                    }
        }
}
```

Output:

10 11 12 13 14 16 17 18 19

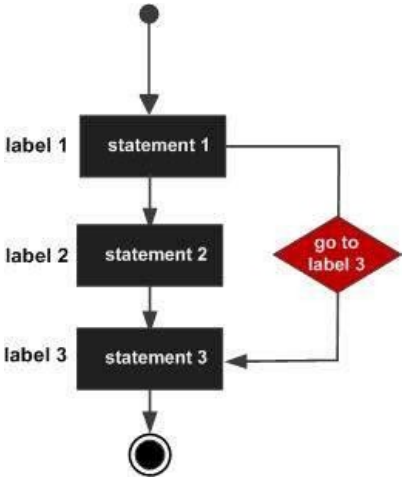
Goto Statement:

A goto statement in C programming language provides an unconditional jump fromthe goto to a labeled statement in the same function.

Syntax:

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

Flow Diagram:



Example:

```
#include
<stdio.h>void
main()
{
        int a = 10;
      LOOP: do
      {
            if( a == 15)
            {
                  a = a + 1;
                        goto LOOP;
            }
            printf(" %d",
            a);a++;
        }while( a < 20 );
}
```

Output:

10 11 12 13 14 16 17 18 19