

# **PROBLEM SOLVING THROUGH PROGRAMMING (18ESCS01)**

## **UNIT-2: C PROGRAMMING BASICS**

1

# C PROGRAMMING BASICS

- Problem formulation – Problem solving
- Introduction to 'C' Programming
- Structure to 'C' Programming
- Execution of 'C' Program
- Important concepts in 'C' programming:  
Character set, Tokens, Keywords, Constants,  
Variables & Data types
- Operators & Expressions in 'C'
- Managing Input & output operations
- Control Statements: Branching & Looping  
statements
- Solving simple scientific & statistical problems.

# PROBLEM FORMULATION – PROBLEM SOLVING



# PROBLEM FORMULATION – PROBLEM SOLVING

## PROBLEM FORMULATION

- Effective problem formulation is fundamental success of all analysis in command and control assessment because the **problems are often complex and ill defined.**
- It is essential for two important aspects:
  - Solve the problem in short time.
  - Save time which helps to ensure quality.
- Note: First find out what the question is, then find out what the real question is – can form a problem easily.

# PROBLEM FORMULATION – PROBLEM SOLVING

## 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 the problem.
- Method of problem solving:
  - Recognize and understand the problems
  - Accumulate facts
  - Select appropriate theory
  - Make necessary assumptions
  - Solve the problems
  - Verify the results

# PROBLEM FORMULATION – PROBLEM SOLVING

## PROBLEM SOLVING

- The 4 steps in using a computer as a problem solving tool:

**Step 1:** Develop an algorithm / flowchart

**Step 2:** Develop a program in computer language

**Step 3:** Compile, debug and Run the program

**Step 4:** Run the program, input data, and get the results from computer.

# INTRODUCTION TO 'C' PROGRAMMING



# INTRODUCTION TO C PROGRAMMING

- 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 language.

## 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.
- But like BCPL and B turned out to be very specific, Dennis Ritchie 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.



# INTRODUCTION TO C PROGRAMMING

## FEATURES of 'C'

- C is a general purpose language .
- C is a structural Language.
- C is middle level language i.e., it supports both the low and high level language features.
- C is flexible and more powerful language with rich set of operators.
- 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 language allows reference to memory location with the help of pointers, which holds the address of the memory locations.

# STRUCTURE OF A 'C' PROGRAM



# STRUCTURE OF C PROGRAM

<b>Documentation Section</b>
<b>Pre-processor Section</b>
<b>Definition Section</b>
<b>Global Declaration Section</b>
<b>main ( )</b> <b>{</b> <b>DECLARATION PART</b> <b>EXECUTION PART</b> <b>}</b>
<b>Sub program section</b> <b>{</b> <b>Body of the sub program</b> <b>}</b>

# STRUCTURE OF C 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 are very helpful in identifying the program features and underlying logic 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) Pre-processor section:

- It is used to link system library files, for defining the macros and for defining the conditional inclusion.
- **Example: #include<stdio.h>**

## (iii) Definition section:

- The definition section defines all symbolic constants.
- **Example: #define pi 3.14**

# STRUCTURE OF C PROGRAM

## (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. I.e. in 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 '}' .

# STRUCTURE OF C PROGRAM

## 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 (;)

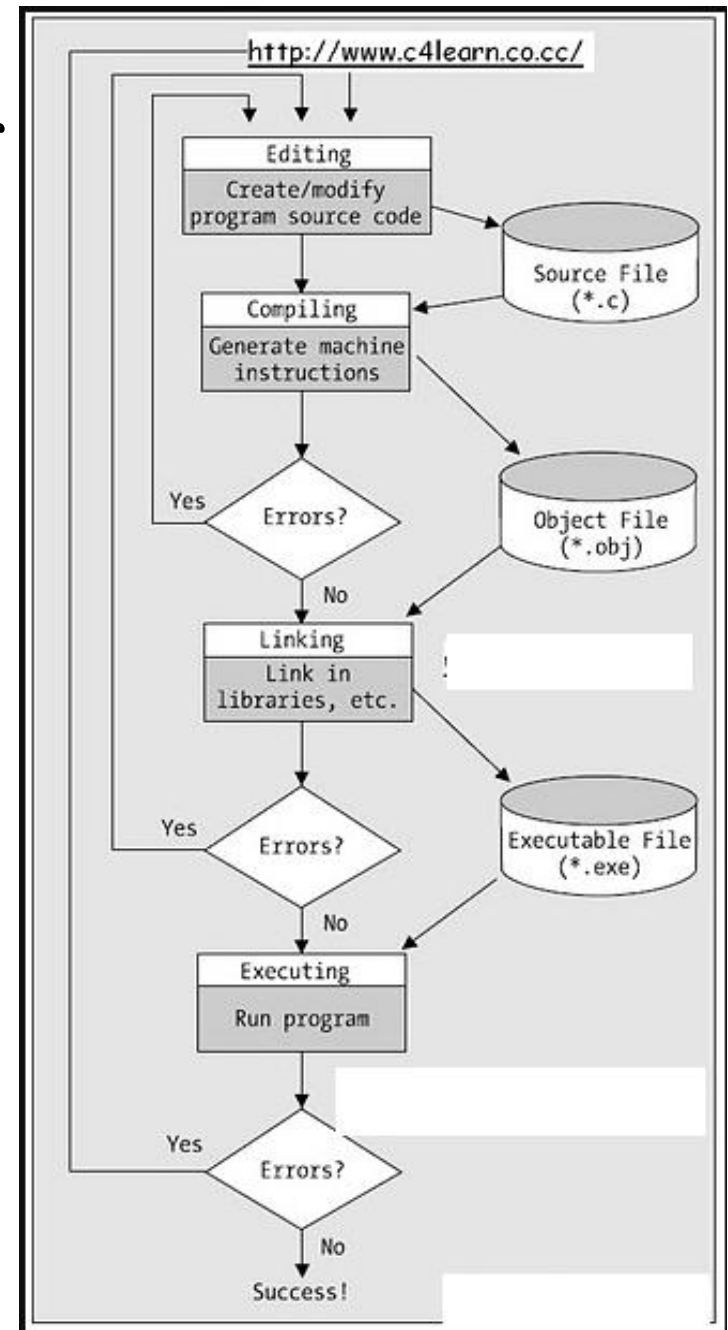
# EXECUTION OF 'C' PROGRAM



# EXECUTING C PROGRAM

- Execution is the process of running the program, to execute a 'C' program, we need to follow the steps:

- (i) Create/Edit the program
- (ii) Compiling the program
- (iii) Checking errors
- (iv) Linking Libraries
- (v) Executing the program





# EXECUTING C PROGRAM

## (i) Create the program

### ○ Steps are as follows:

- This is First Step is Creating and Editing Program.
- Firstly, we have to write a C program using an editor.
- Save the Program by using [.C] Extension.
- File Saved with [.C] extension is called “Source Program “.

## (ii) Compiling the program

- Compiling C Program: C Source code with [.C] Extension is given as input to compiler such as Turbo C and compiler converts it into Equivalent Machine Instruction.
- Compiler Checks for errors. If source code is error-free then Code is converted into Object File [.obj ].

# EXECUTING C PROGRAM

## (iii) Checking Errors

- During Compilation Compiler will check for error, if compiler finds any error then it will report it.
- User has to go back to the source program and edit the program to correct the errors.
- After editing and saving the program, it is compiled again to check for any errors.
- If program is error-free, then program is linked with appropriate libraries.

## (iv) Linking Libraries

- The program is linked with included header files.
- The program is linked with other libraries.
- The process of linking the source program with libraries and header files is achieved using a software called Linker.

# EXECUTING C PROGRAM

## (v) Executing the program

- This is the process of running and testing the program with the sample data.
- Once the program is compiled and linked, an executable “.exe” file is generated. The program is executed using this “.exe” file.
- 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.

# IMPORTANT CONCEPTS OF 'C' PROGRAMMING

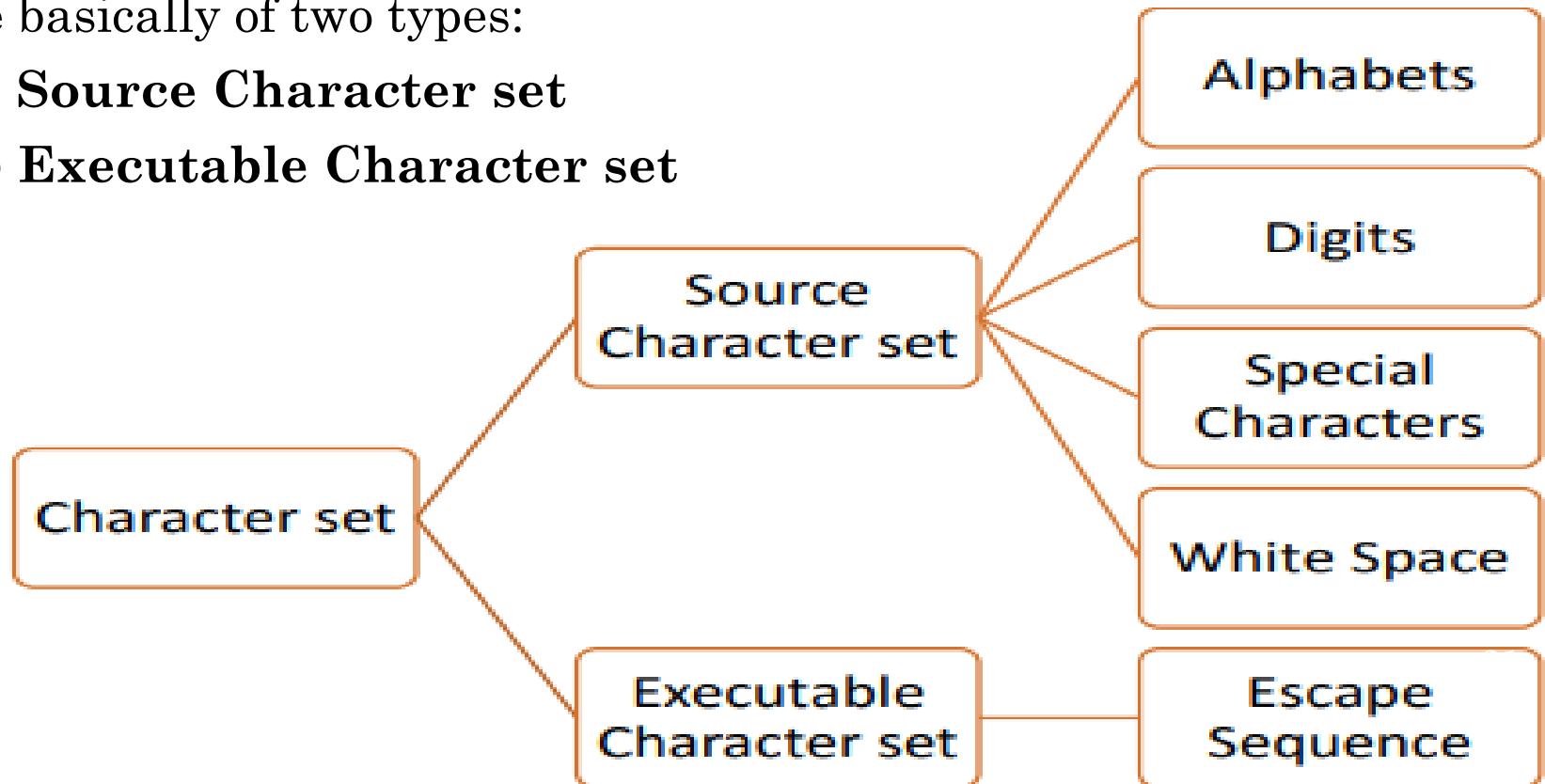


## (A) CHARACTER SET

- 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 known as its character set.
- These characters can be represented in the computers. C programs are basically of two types:

(a) **Source Character set**

(b) **Executable Character set**



# (A) CHARACTER SET

## (a) Source Character Set:

- They are used to construct the statements in the source programs.
- **Alphabets:**
  - **Uppercase Letters (26):** A - Z,
  - **Lowercase Letters (26):** a - z.
- **Digits (10):** 0 to 9
- **Special characters (30):**  
! # % ^ & \* ( ) - \_ = + ~ ' " : ; ? / | \ { } [ ] , . < > \$
- **White Spaces (5) :** \b - Blank space, \t - horizontal tab, \f - Form Feed, \v - vertical tab, \n - new line

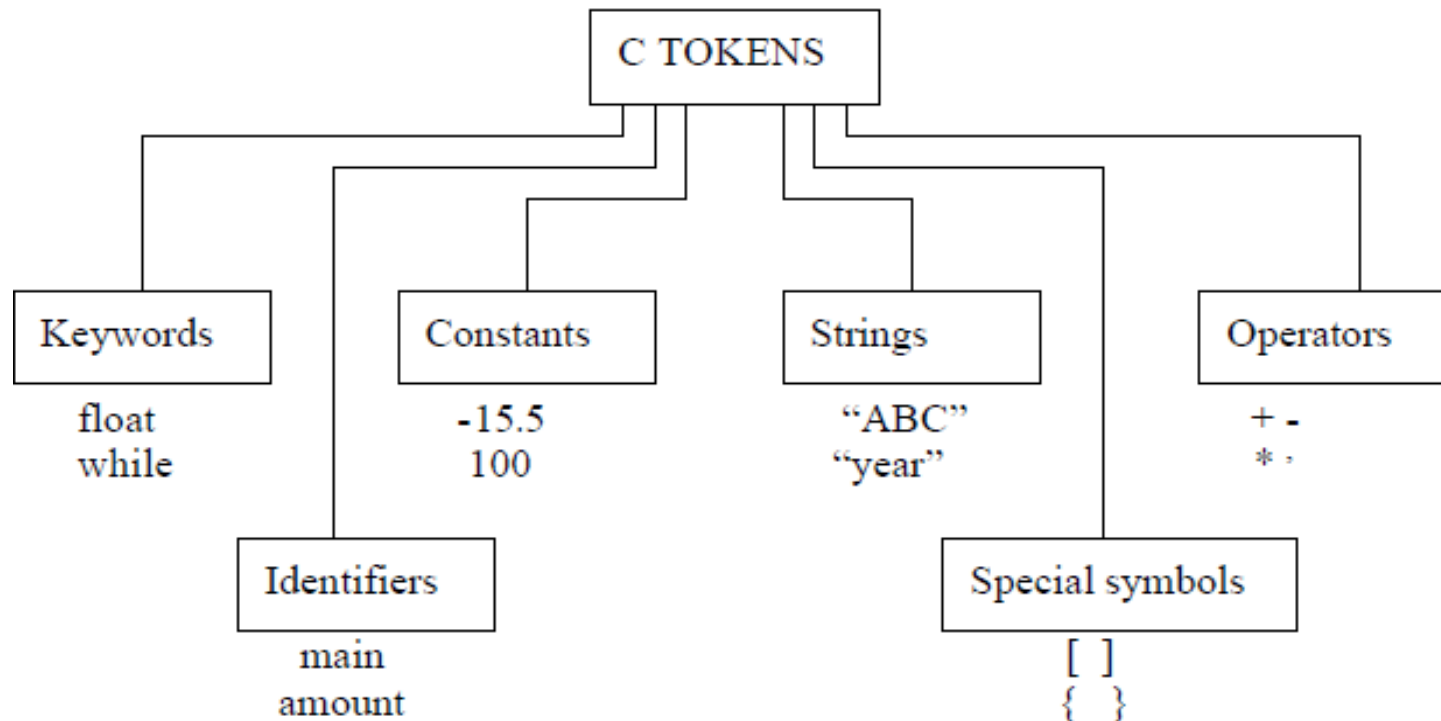
## (A) CHARACTER SET

### (b) Executable Character Set:

Character	ASCII	Escape Sequence	Result
Null	000	\0	Null
Alarm (Bell)	007	\a	Beep Sound
Back Space	008	\b	Moves Previous Position
Horizontal Tab	009	\t	Moves next horizontal tab
New line	010	\n	Moves New line
Vertical tab	011	\v	Moves next vertical tab

## (B) TOKENS

- **C tokens** are the basic building blocks in C language which are constructed together to write a C program. Each and every smallest individual units in a C program are known as C tokens.
- The tokens are usually referred as individual text and punctuation in a passage of text.
- C tokens has following types:





## (I) KEYWORDS

- 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, double, else, enum, extern, float, for, goto, if, return, register, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while**

## (II) 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, etc.**

- Example of **invalid identifiers**:

**Length of line, Year's, etc.**

### **RULES FOR WRITING IDENTIFIERS:**

- It contains letters and digits.
- ‘\_’ can also be used.
- First character must be a letter or \_
- Contain only 31 characters.
- No space and special symbols are allowed.
- It cannot be a keyword.

### (III) CONSTANTS

- The item whose values cannot be changed during the execution of program called **constants**.
- Three types of constants:

#### (a) Literal Constants:

- A literal constant is a value that you put directly in your code.

#### (b) Symbolic Constants:

- A symbolic constant is a constant that has a name.
- **Example:** #define PI 3.14

#### (c) Qualifier Constants:

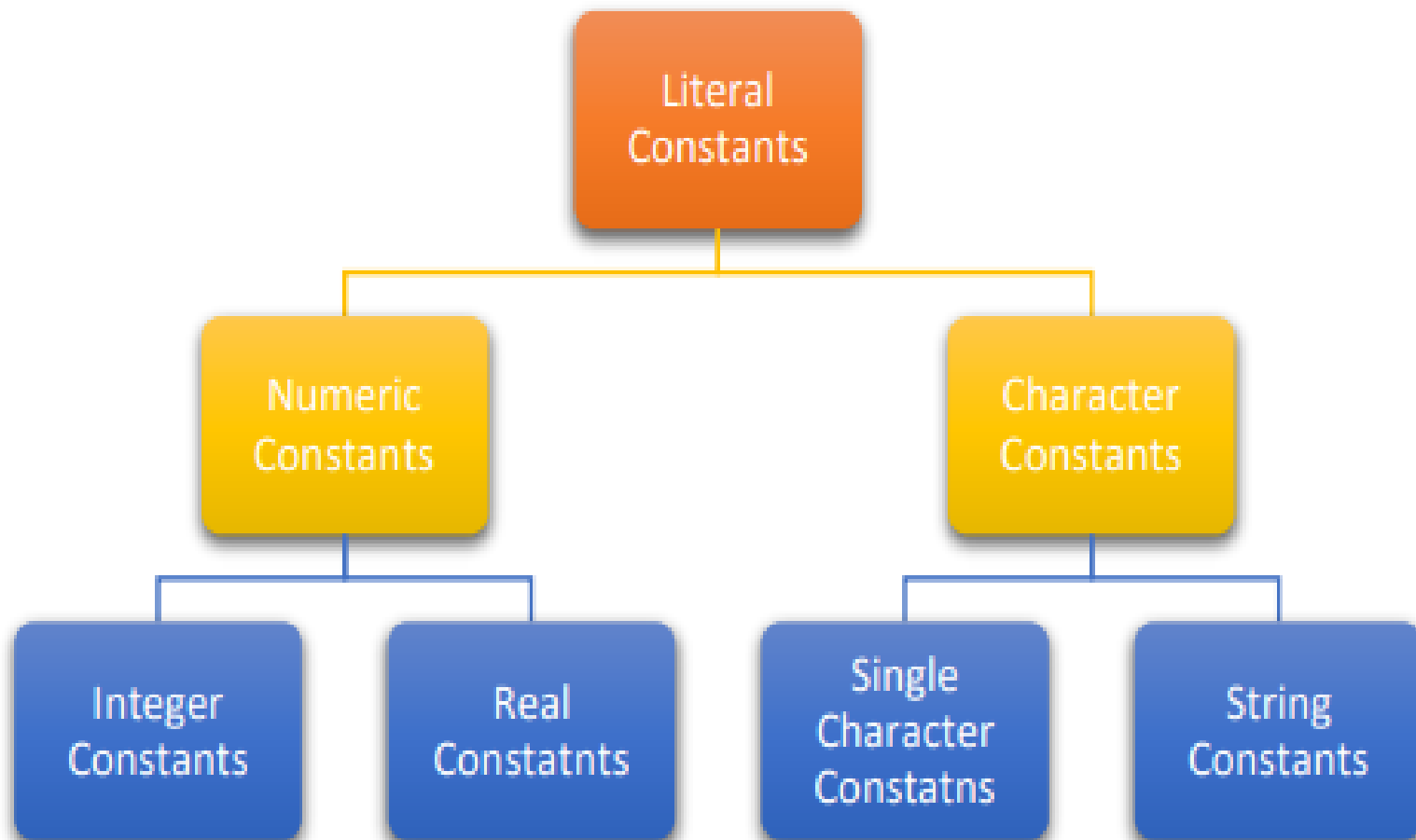
- The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed .
- **Example:** const float pi=3.14;

# CONSTANTS – LITERAL CONSTANTS

- Literal constants are classified into 2 types:

**(1) Numeric Constants**

**(2) Character Constants**



# CONSTANTS –NUMERIC CONSTANTS

## (i) 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 are:
  - 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 -32768 to 32767 .
- Examples of **valid numeric** constants: **426, +785, -100** etc.
- Examples of **Invalid numeric** constants: **2.3, 0.235, 3,500**, etc.

# CONSTANTS – NUMERIC CONSTANTS

## (ii) 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 are:
  - 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.
- Examples of **valid real** constants:  
**+325.34, 426.0, -32.76, -48.456** etc.

# CONSTANTS – CHARACTER CONSTANTS

## (i) Single Character Constants:

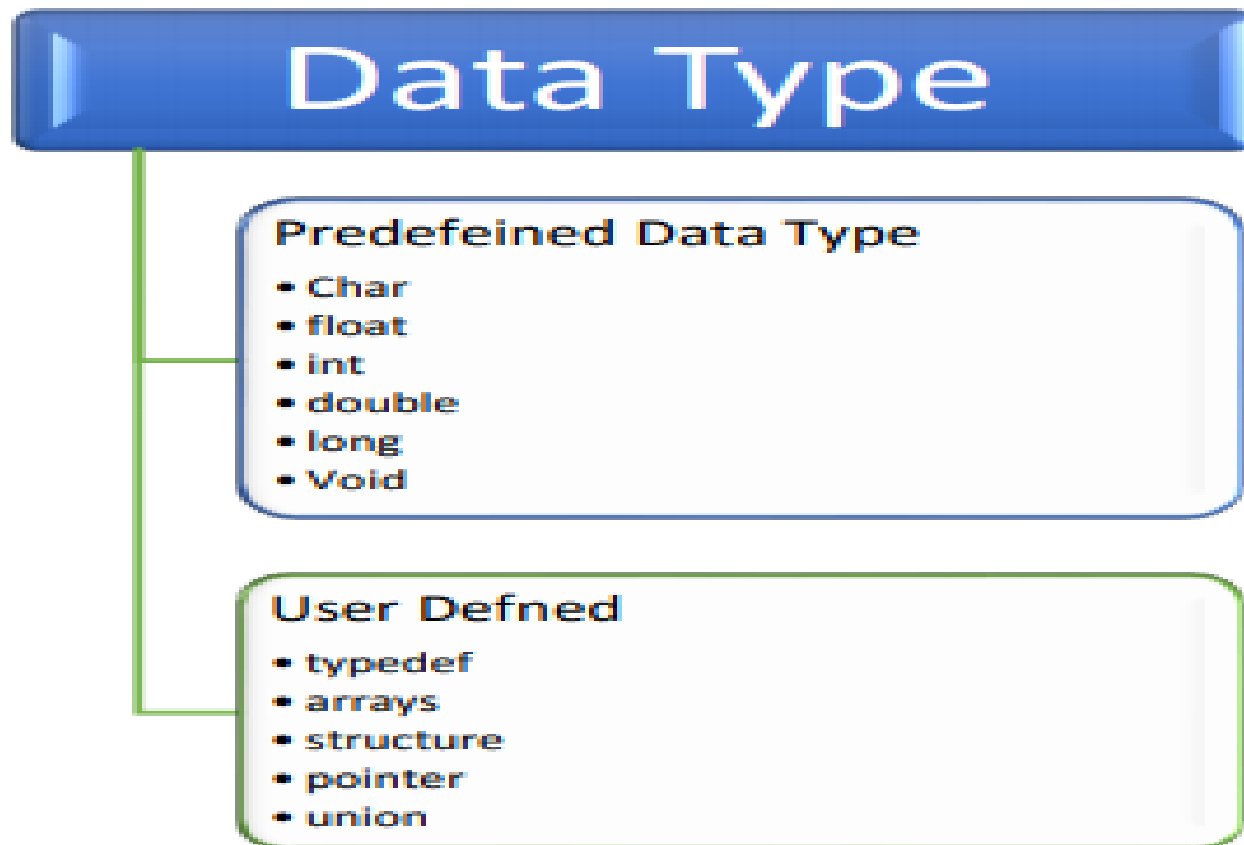
- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas (‘ ’).
- The maximum length of a character constant can be 1 character.
- **Example:** ‘A’, ‘I’, ‘5’, etc.

## (ii) String Character Constants:

- String Constants are sequence of characters within double quote marks (“ ”).
- The string may be combination of all kinds of symbols.
- **Example:** “Hello”, “India”, “444”, “a” etc.

## (IV) DATA TYPE

- 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.





## (IV) DATA TYPE

### PRIMARY DATA TYPES

#### Integral Type

##### Integer

signed type	unsigned type
int	unsigned int
short int	unsigned short int
long int	unsigned long int

##### Character

signed char
unsigned char

#### Floating Point Type

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

## DATA TYPE – INTEGER DATA TYPE

### (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;`

### (iii) Signed Integer:

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

### (iv) Unsigned Integer:

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

## DATA TYPE – CHARACTER DATA TYPE

### (i) Signed character:

- It occupies 1 byte of memory.
- Range is from -128 to 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';`

## DATA TYPE – FLOAT DATA TYPE

- It occupies 4 bytes of memory.
- Range is:  $-3.4e-38$  to  $3.4e+38$ .
- Format specifier is %f.
- **Example:** float f=3.14;

## DATA TYPE – 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

## Integer Types

Type	Size (bits)	Range
int or signed int	16	-32,768 to 32767
unsigned int	16	0 to 65535
short int	8	-128 to 127
unsigned short int	8	0 to 255
long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295

## Floating Point Types

Type	Size(bits)	Range
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

## Character Types

Type	Size (bits)	Range
char	8	-128 to 127
unsigned char	8	0 to 255

## (v) 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.
- Example: **si\_int**;

# VARIABLES

## (i) 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;**  
**float s;**  
**char m;**

# VARIABLES

## (ii) 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 = 5;**  
**int z = 15;**  
**char ch = 's';**



# VARIABLES

## (iii) CONSTANT VARIABLES:

- 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;**

# VARIABLES

## (iv) VOLATILE VARIABLES:

- 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;**

# EXPRESSION USING OPERATORS IN 'C'



# 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 OPERATORS

- (a) Arithmetic Operators
- (b) Relational Operators
- (c) Logical Operators
- (d) Assignment Operators
- (e) Increment and decrement Operators
- (f) Conditional Operators
- (g) Bitwise Operators
- (h) Special Operators

# OPERATORS AND EXPRESSIONS

## (A) ARITHMETIC OPERATORS

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

<i>Operator</i>	<i>Operation</i>	<i>Example</i>
+	Addition	$2+2=4$
-	Subtraction	$2-2=0$
*	Multiplication	$2*2=4$
/	Division	$2/2=1$
%	Modulo Division	$2\%2=0$

# OPERATORS AND EXPRESSIONS

## (A) ARITHMETIC OPERATORS - 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 : 800

Division of a, b is : 2

Modulus of a, b is : 0

# OPERATORS AND EXPRESSIONS

## (B) 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

# OPERATORS AND EXPRESSIONS

## (B) RELATIONAL OPERATORS - 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



# OPERATORS AND EXPRESSIONS

## (C) 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	!	logical NOT	!((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

# OPERATORS AND EXPRESSIONS

## (C) LOGICAL OPERATORS - 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

# OPERATORS AND EXPRESSIONS

## (D) 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

# OPERATORS AND EXPRESSIONS

## (D) ASSIGNMENT OPERATORS - EXAMPLE

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

```
int main( )
```

```
{
```

```
int Total=0, i;
```

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

```
{
```

```
    Total+= i; // This is same as Total = Total + i
```

```
}
```

```
printf("Total = %d", Total);
```

```
}
```

**Output:**

Total = 45

# OPERATORS AND EXPRESSIONS

## (E) 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	<code>++i</code>	Value of i is incremented before assigning it to variable i.
2	Post-increment	<code>i++</code>	Value of i is incremented after assigning it to variable i.
3	Pre decrement	<code>--i</code>	Value of i is decremented before assigning it to variable i.
4	Post_decrement	<code>i--</code>	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 -- ;`

# OPERATORS AND EXPRESSIONS

## (E) INCREMENT & DECREMENT OPERATORS - EXAMPLE

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

```
void main()
```

```
{
```

```
int i = 1;
```

```
while(i < 10)
```

```
{
```

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

```
i++;
```

```
}
```

```
}
```

**Output:**

1 2 3 4 5 6 7 8 9

# OPERATORS AND EXPRESSIONS

## (F) CONDITIONAL OPERATORS

- Conditional operators return one value if condition is true and returns another value if condition is false.
- This operator is also called as ternary operator.
- Syntax: **(Condition? true\_value: false\_value);**
- Example: **(A > 100 ? 0 : 1);**

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

# OPERATORS AND EXPRESSIONS

## (F) CONDITIONAL OPERATORS - 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
```



# OPERATORS AND EXPRESSIONS

## (G) BITWISE OPERATORS

- Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for  $\&$ ,  $|$ , and  $\wedge$  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

- Example:** Assume if  $A = 60$ ; and  $B = 13$ ;  
     $A = 0011\ 1100$   $B = 0000\ 1101$   
     $A \& B = 0000\ 1100$   
     $A | B = 0011\ 1101$   
     $A \wedge B = 0011\ 0001$   
     $\sim A = 1100\ 0011$

# OPERATORS AND EXPRESSIONS

## (G) BITWISE OPERATORS

- 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

# OPERATORS AND EXPRESSIONS

## (G) BITWISE OPERATORS - 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 );
```

```
}
```

# OPERATORS AND EXPRESSIONS

## (H) SPECIAL OPERATORS

- Below are some of special operators that C language.

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.

## (H) SPECIAL OPERATORS - 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 = 2

Size of variable b = 4

# OPERATORS AND EXPRESSIONS

## 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.

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

# OPERATORS AND EXPRESSIONS

## OPERATORS PRECEDENCE - EXAMPLE

- 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.

## OPERATORS PRECEDENCE - EXAMPLE

### Program

```
/*Evaluation of expressions*/  
main( )  
{  
float a, b, c, y, x, z;  
a = 9;  
b = 12;  
c = 3;  
x = a - b / 3 + c * 2 - 1;  
y = a - b / (3 + c) * (2 - 1);  
z = a - (b / (3 + c) * 2) - 1;  
printf("x = %f \n",x);  
printf("y = %f \n",y);  
printf("z = %f \n",z);  
}
```

### Output:

**x = 10.000000**

**y = 7.000000**

**z = 4.000000**

# MANAGING INPUT & OUTPUT OPERATORS





# MANAGING INPUT & OUTPUT OPERATIONS

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

## Formatted I/O Functions

### Input Functions

- scanf()
- fscanf()

### Output Functions

- printf()
- fprintf()

## Unformatted I/O Functions

### Input Functions

- getc()
- getch()
- getchar()
- getche()
- gets()

### Output Functions

- putc()
- putchar()
- putchar()
- putche()
- putchar()

# MANAGING INPUT & OUTPUT OPERATIONS

## FORMATTED INPUT FUNCTIONS

### (a) **scanf( ) function:**

- The `scanf( )` function is used to read information the standard 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 ) ;`

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

# MANAGING INPUT & OUTPUT OPERATIONS

## FORMATTED OUTPUT FUNCTIONS

### (a) printf( ) function:

- The output data or result of an operation can be displayed from the computer to a standard 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:
  - Characters that are simply printed as they are.
  - Conversion specifications that begin with a % sign.
  - Escape sequences that begin with a \ sign.

#### ○ Example:

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

#### Output:

```
Average = 346
Percentage = 69.20
```

# MANAGING INPUT & OUTPUT OPERATIONS

## UNFORMATTED INPUT 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 / Output.

### (a) 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( );**

### (b) getchar( ) function:

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

# MANAGING INPUT & OUTPUT OPERATIONS

## UNFORMATTED INPUT FUNCTIONS

### (c) getche() function:

- getche() also accepts only single character, but unlike getch(), getche() displays the entered character in the screen.
- Syntax: **variable\_name = getche();**

### Example for getch(), getche(), getchar():

```
void main()  
{ char ch ;  
  printf ( "\n Press any key to continue:" ) ;  
  getch() ; /* will not echo the character */  
  printf ( "\n Type any character" ) ;  
  ch = getche() ; /* will echo the character typed */  
  printf ( "\n Type 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

# MANAGING INPUT & OUTPUT OPERATIONS

## UNFORMATTED INPUT FUNCTIONS

### (d) 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);**

# MANAGING INPUT & OUTPUT OPERATIONS

## UNFORMATTED OUTPUT FUNCTIONS

### (a) **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);**

### (b) **putchar()** function:

- **putchar** displays one character at a time to the Monitor.
- Syntax: **putchar(variable\_name);**
- Example: **char z[20] = “welcome”;**  
**putchar(z);**

# MANAGING INPUT & OUTPUT OPERATIONS

## UNFORMATTED OUTPUT FUNCTIONS

### (c) 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 efgh
Abcd efgh
```



## Example Program

*/\*A program to read a character from keyboard and then prints it in reverse case\*/*

*/\*This program uses three new functions: **islower,toupper,and tolower.***

```
#include<stdio.h>
#include<ctype.h>
main()
{
char alphabet;
printf("Enter an alphabet");
putchar('\n');
alphabet = getchar();
if(islower(alphabet))
putchar(toupper(alphabet));
else
putchar(tolower(alphabet));
}
```

### Output:

**Enter An alphabet**

**a**

**A**

**Enter An alphabet**

**Q**

**q**

**Enter An alphabet**

**z**

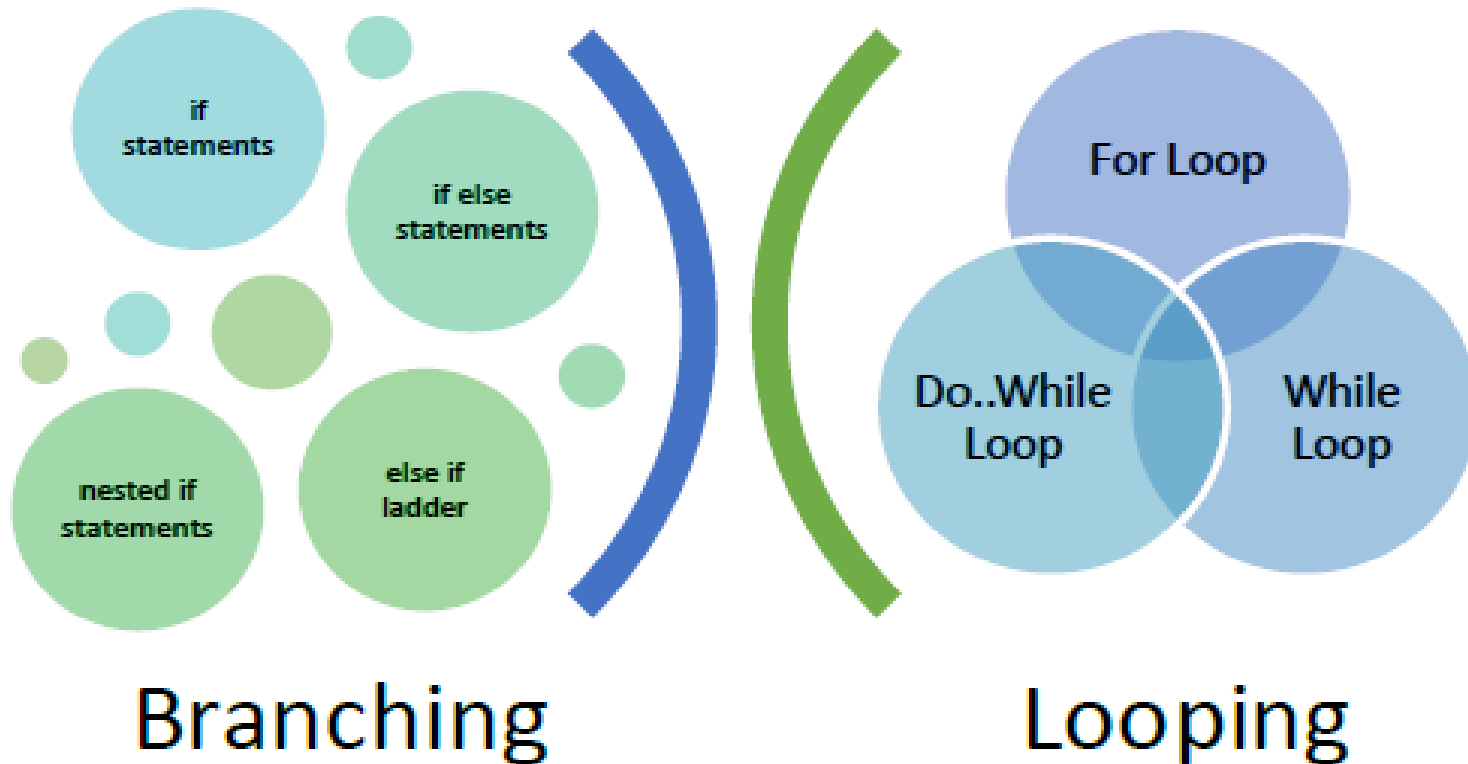
**Z**

# DECISION MAKING & BRANCHING



# 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



# CONTROL STATEMENTS

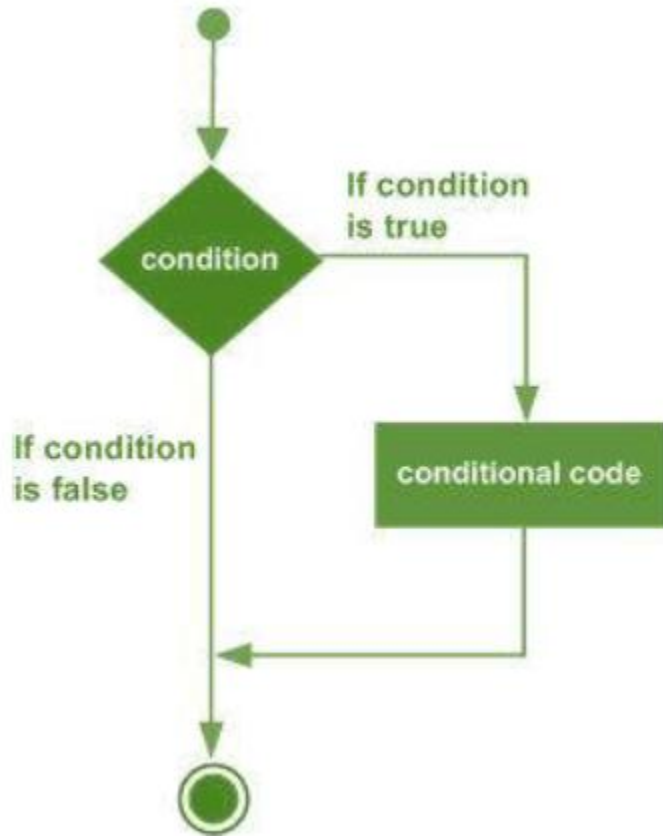
## BRANCHING STATEMENTS

- 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:
  - (a) if statements
  - (b) ifelse statements
  - (c) nested if statements
  - (d) elseif statements
  - (e) Switch statements

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – IF STATEMENT

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

Syntax:	Flow Diagram:
<pre>if(condition) {     Statements; }</pre>	 <pre>graph TD; Start(( )) --&gt; Condition{condition}; Condition -- "If condition is true" --&gt; Code[conditional code]; Condition -- "If condition is false" --&gt; End((( ))); Code --&gt; End;</pre>

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – IF STATEMENT

### 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);
}
```

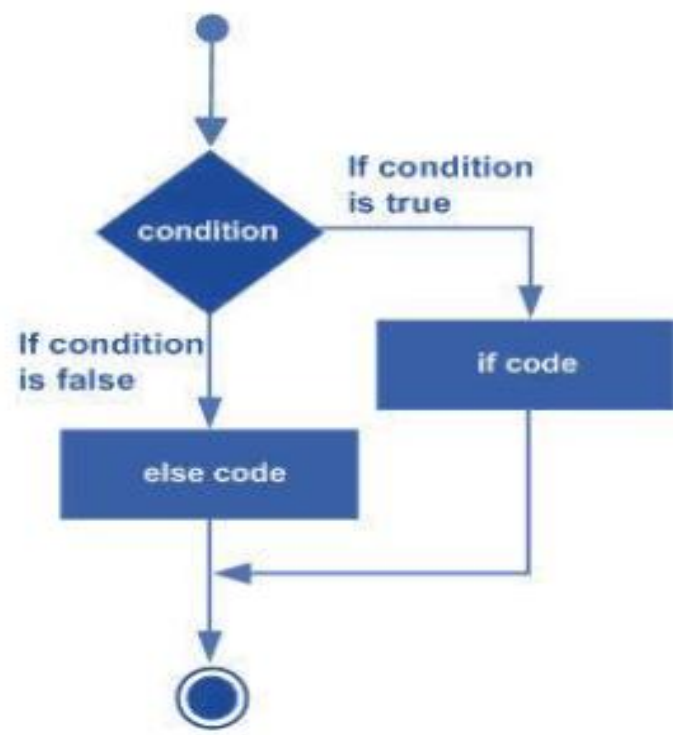
### Output:

A is less than 20  
Value of a is: 10

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – IF..ELSE STATEMENT

- 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:	Flow Diagram:
<pre>if(condition) { True Statements; } else { False statements; }</pre>	 <pre>graph TD; Start(( )) --&gt; Condition{condition}; Condition -- "If condition is true" --&gt; IfCode[if code]; Condition -- "If condition is false" --&gt; ElseCode[else code]; IfCode --&gt; Join(( )); ElseCode --&gt; Join; Join --&gt; End((( )));</pre> <p>The flow diagram illustrates the execution of an if..else statement. It begins with a start node (a solid blue circle) leading to a decision diamond labeled 'condition'. If the condition is true, the flow proceeds to a rectangular box labeled 'if code'. If the condition is false, the flow proceeds to a rectangular box labeled 'else code'. Both paths converge at a join point, indicated by a small circle, before reaching the final end node (a double blue circle).</p>

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – IF..ELSE STATEMENT

### 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



# CONTROL STATEMENTS

## BRANCHING STATEMENTS – ELSE..IF STATEMENT

- 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.

**Syntax:**

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

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – ELSE..IF STATEMENT

### 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**

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – NESTED IF STATEMENT

- 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:	Flow Diagram:
<pre>if( Condition 1) { /* Executes when the condition 1 is true */ if(Condition 2) { /* Executes when the condition 2 is true */ } } else { /* Executes</pre>	<pre>graph TD     Entry(( )) --&gt; Cond1{Condition 1}     Cond1 -- True --&gt; StmtA[Statement A]     Cond1 -- False --&gt; StmtB[statement B]     StmtA --&gt; Cond2{Condition 2}     Cond2 -- True --&gt; StmtC[Statement C]     Cond2 -- False --&gt; StmtD[Statement D]     StmtC --&gt; Next[Next statement]     StmtB --&gt; Next     StmtD --&gt; Next     Next --&gt; Exit(( ))</pre> <p>The flow diagram illustrates the execution of a nested if statement. It begins with an entry point leading to a decision diamond labeled 'Condition 1'. If 'Condition 1' is true, the flow proceeds to a rectangular box labeled 'Statement A'. If 'Condition 1' is false, the flow bypasses 'Statement A' and goes to a rectangular box labeled 'statement B'. From 'Statement A', the flow enters another decision diamond labeled 'Condition 2'. If 'Condition 2' is true, the flow goes to a rectangular box labeled 'Statement C'. If 'Condition 2' is false, the flow goes to a rectangular box labeled 'Statement D'. Both 'Statement C' and 'Statement D' lead to a final rectangular box labeled 'Next statement'. Additionally, the flow from 'statement B' also leads to the 'Next statement' box. Finally, an arrow points downwards from the 'Next statement' box, indicating the continuation of the program.</p>

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – NESTED IF STATEMENT

### 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 );
}
```

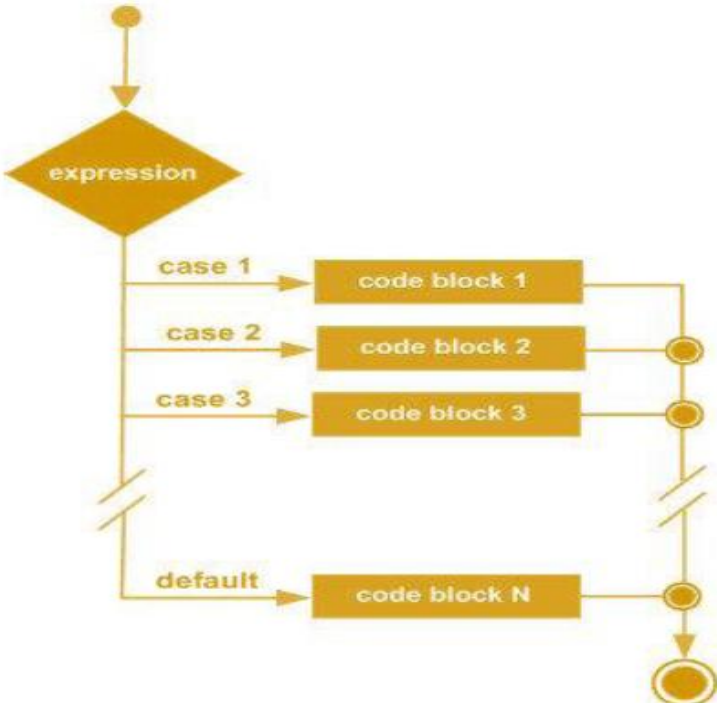
### Output:

**Value of a is 100 and b is 200**  
**Exact value of a is : 100**  
**Exact value of b is : 200**

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – SWITCH STATEMENT

- 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:	Flow Diagram:
<pre>switch (expression) {   case label1:     statements; break;   case label2:     statements; break;   default:     statements; }</pre>	 <p>The flow diagram illustrates the execution of a switch statement. It begins with a start node (a small circle) leading to a decision diamond labeled 'expression'. From the diamond, four paths emerge: 'case 1' leading to 'code block 1', 'case 2' leading to 'code block 2', 'case 3' leading to 'code block 3', and a 'default' path leading to 'code block N'. Each code block is represented by a rectangle. After each code block, there is a small circle, and a vertical line with a break symbol (two parallel diagonal lines) connects these circles. The flow ends at a final node (a larger circle) after the default path.</p>

# CONTROL STATEMENTS

## BRANCHING STATEMENTS – SWITCH STATEMENT

### Example:

```
#include <stdio.h>

void main ()
{  char grade = 'B';
  switch(grade)
  {  case 'A' : printf("Excellent!\n" );   break;
    case 'B' : printf("Good!\n" );         break;
    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:

**Good**  
**Your grade is B**

# LOOPING STATEMENTS



# CONTROL STATEMENTS

## LOOPING STATEMENTS

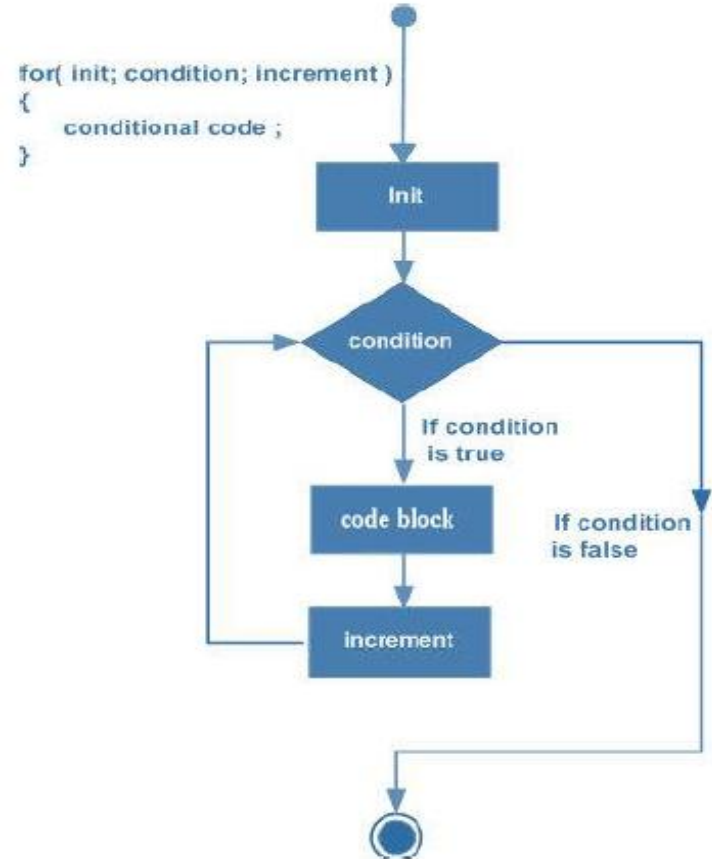
- Loop control statements in C are used to perform looping operations until the given condition is true.
- Control comes out of the loop statements once condition becomes false.
- There are 3 types:
  - (a) for loop
  - (b) while loop
  - (c) do while loop
  - (d) break
  - (e) continue
  - (f) goto



# CONTROL STATEMENTS

## LOOPING STATEMENTS – 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:	Flow Diagram:
<pre>for ( init; condition; increment ) { statement(s); }</pre>	 <pre>graph TD     Start(( )) --&gt; Init[Init]     Init --&gt; Condition{condition}     Condition -- "If condition is true" --&gt; CodeBlock[code block]     CodeBlock --&gt; Increment[Increment]     Increment --&gt; Condition     Condition -- "If condition is false" --&gt; End((( )))</pre> <p>The flow diagram illustrates the execution of a for loop. It begins with an initialization step (Init), followed by a decision diamond (condition). If the condition is true, the flow proceeds to a code block, then to an increment step, and loops back to the condition check. If the condition is false, the flow exits the loop and proceeds to the end terminal.</p>

# CONTROL STATEMENTS

## LOOPING STATEMENTS – FOR LOOP

### 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

# CONTROL STATEMENTS

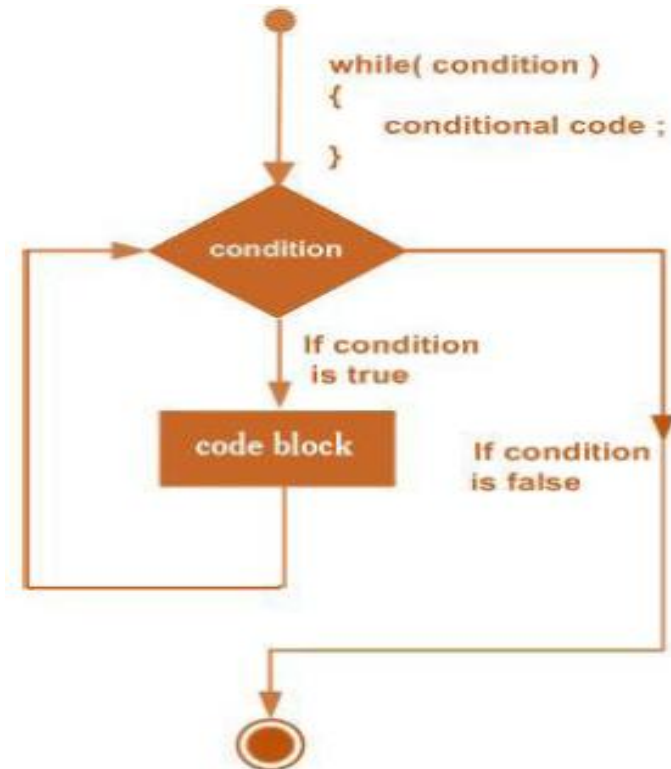
## LOOPING STATEMENTS – WHILE LOOP

- 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:



# CONTROL STATEMENTS

## LOOPING STATEMENTS – WHILE LOOP

### 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

# CONTROL STATEMENTS

## LOOPING STATEMENTS – 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:	Flow Diagram:
<pre>do { statement(s); }while( condition );</pre>	<pre>graph TD     Start(( )) --&gt; DoWhile[do { conditional code ; } while (condition)]     DoWhile --&gt; CodeBlock[code block]     CodeBlock --&gt; Condition{condition}     Condition -- "If condition is true" --&gt; CodeBlock     Condition -- "If condition is false" --&gt; End((( )))</pre>

# CONTROL STATEMENTS

## LOOPING STATEMENTS – DO...WHILE LOOP

### 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

# CONTROL STATEMENTS

## LOOPING STATEMENTS – BREAK LOOP

- 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:	Flow Diagram:
<b>break;</b>	<pre>graph TD; Entry(( )) --&gt; CC[conditional code]; CC --&gt; C{condition}; C -- "If condition is true" --&gt; CC; C -- "If condition is false" --&gt; B{break}; B --&gt; Exit(( ))</pre>

# CONTROL STATEMENTS

## LOOPING STATEMENTS – BREAK LOOP

### 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**



# CONTROL STATEMENTS

## LOOPING STATEMENTS – CONTINUE LOOP

- 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:	Flow Diagram:
<b>continue;</b>	<pre>graph TD; Start(( )) --&gt; CC[conditional code]; CC --&gt; Cond{condition}; Cond -- "If condition is true" --&gt; CC; Cond -- "If condition is false" --&gt; End((( )))</pre>

# CONTROL STATEMENTS

## LOOPING STATEMENTS – CONTINUE LOOP

### 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**

# CONTROL STATEMENTS

## LOOPING STATEMENTS – GOTO LOOP

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

Syntax:	Flow Diagram:
<pre>goto label; .... .... label: statement;</pre>	<pre>graph TD; Start(( )) --&gt; S1[statement 1]; S1 --&gt; S2[statement 2]; S2 --&gt; S3[statement 3]; S3 --&gt; D{go to label 3}; D --&gt; S1; D --&gt; End((( )));</pre>

# CONTROL STATEMENTS

## LOOPING STATEMENTS – GOTO LOOP

### 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**