

## **MODULE -1**

### **BASICS OF C PROGRAMMING**

#### **INTRODUCTION TO C LANGUAGE**

C is a procedural oriented programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system. The main features of the C language include low-level memory access, a simple set of keywords, and a clean style, these features make C language suitable for system programming's like an operating system or compiler development.

#### **Characteristics of procedure-oriented programming language:**

1. It emphasis on algorithm (doing this).
2. Large programs are divided into smaller programs known as functions.
3. Function can communicate by global variable.
4. Data move freely from one function to another function.
5. Functions change the value of data at any time from any place. (Functions transform data from one form to another.)
6. It uses top-down programming approach.

#### **Features of C Programming Language:**

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. Statically Type
5. General-Purpose Language
6. Rich set of built-in Operators
7. Libraries with rich Functions
8. Middle-Level Language
9. Portability
10. Easy to Extend

### **ALGORITHM**

- An Algorithm is the collection of thoughts processes which give the precise method of solving a problem.
- In simpler: Algorithm is defined as the step by step procedure to solve a problem.
- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

**Let's try to learn algorithm-writing by using an example.**

#### **Example**

**Problem** – Design an algorithm to add two numbers and display the result.

**Step 1** – START

**Step 2** – declare three integers **a, b & c**

**Step 3** – define values of **a & b**

**Step 4** – add values of **a & b**

**Step 5** – store output of step 4 to **c**

**Step 6** – print **c**

**Step 7** – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

**Step 1** – START ADD

**Step 2** – get values of **a & b**

**Step 3** –  $c \leftarrow a + b$

**Step 4** – display c



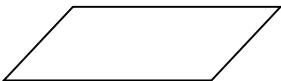
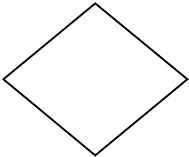
**Step 5** – STOP

### **FLOWCHART**

It is a pictorial / diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

**These are some points to keep in mind while developing a flowchart –**

- Flowchart can have only one start and one stop symbol
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Symbol	Meaning
	Used at the beginning and end of the algorithm to show start and end of the program
	Indicates processes like mathematical operations.
	Used for denoting program inputs and outputs
	Stands for decision statements in a program, where answer is usually Yes or No



Stands for looping statements in a program, where the statements are repeated

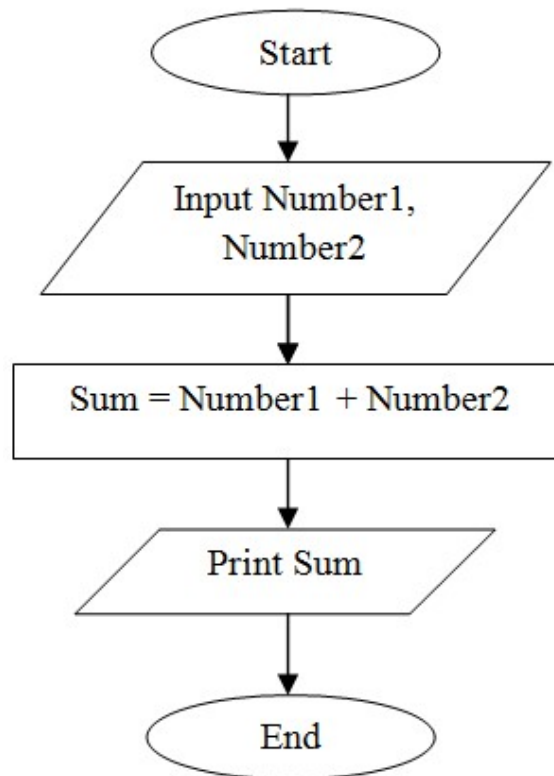


Shows relationships between different shapes



Connects two or more parts of a flowchart, which are on the same page

**Example:** Write a flowchart to perform addition of two numbers-



### **APPLICATIONS OF C**

1. Operating Systems
2. GUI
3. Embedded Systems
4. Database
5. Ease of Computation
6. Gaming
7. Development of New languages
8. Google
9. Assemblers
10. Text Editors
11. Drivers
12. Interpreters
13. Network Devices
14. Compiler Design

### **BASIC CONCEPT OF C PROGRAM ( OR ) STRUCTURE OF C PROGRAM ( OR ) FORMAT OF C PROGRAM ( OR ) GENERAL OUTLINE OF C PROGRAM**

#### **1. Comments and Programming Style**

- ✓ Comments are the short description of the problem to be solved. The comment is not strictly necessary because program without it is technically correct.
- ✓ But comments are considered the single most important part of good programming style. The possible comment can be in this form.

**/\*Program 1.C print the numbers from 4to9 and their squares\*/**

The comments begin with /\* and ends with \*/

- ✓ These symbols are known as comment delimiters because they delimit or mark the beginning and end of the comment. Between them we can put any message we want, abbreviations, misspelled words, and so on,

- ✓ Everything between the comment delimiters is ignored by the C compiler. Note that we start the comment with the name of the program prog1.C; this is the name of the file which will be saved in the memory or in the secondary memory.

### 2. The Program Header

After the comment the next line will be the program header of our C program which looks like follows.

```
/*stdio.h*/
```

- ✓ Any line in a program that starts with # is an instruction to the compiler, not an actual statement in C language. The line containing #include tells the compiler to allow our program to perform standard input output operation.
  - ✓ The # include directive tells the compiler that we will be using parts of the standard library function.
  - ✓ A function is a building block of a program typically each function performs one particular task. Information about this function is containing in header files. In our case we use the header files called <stdio.h> will be included in our program stdio is the short form for standard input output. That dot (.) h says this file is a header file. The pointed brackets < and > tells the compiler that exact location of the header file.
3. **Global declaration:** The variables that are common to all the functions are declared as global variables. This section is optional.

4. **main ():** The second line after the comment, main () is called the main program header. It tells the C compiler that we are using the main () in the program. Every program must have a main function [main ()] because the execution of program starts from main. The parenthesis symbol (and) indicates that it is a function.

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

```
main()
```

### 5. The body or action portion of the program

- ✓ In every C main program, we use a set of braces {and} containing a series of a C statement which comprise the body; this is called the action portion of the program (or) body of the function.

#### **The Structure of “C” Program**

Here is a structure of our C program

*/\*program specifies structure of C\*/*

```
#include<stdio.h>

main ( )
{
    -----
    /*action portion or body of the function*/
    -----
    -----
}
```

6. User defined functions: These are the functions defined or written by the user. This section is optional.

### **WRITING MY FIRST C PROGRAM**

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open any editor and write the following code:

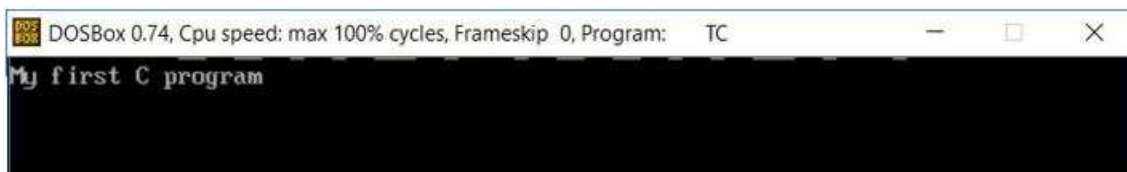
```
#include <stdio.h>
int main()
{
    printf("My first C program");
return 0;
}
```

- **#include <stdio.h>** includes the **standard input output** library functions.
- The **printf()** function is defined in **stdio.h**.
- **int main()** The **main()** function is the **entry point of every program** in c language.
- **printf()**: The **printf()** function is used to print data on the console.
- **return 0**: The **return 0** statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

### How to compile and run the C program

There are 2 ways to compile and run the c program, by menu and by shortcut.

- By menu
  - Now **click on the compile menu then compile sub menu** to compile the c program.
  - Then **click on the run menu then run sub menu** to run the c program.
- By shortcut
  - **Or, press ctrl+f9** keys compile and run the program directly.
  - You will see the following output on user screen(**alt+F5**)



### FILES CREATED IN C PROGRAM

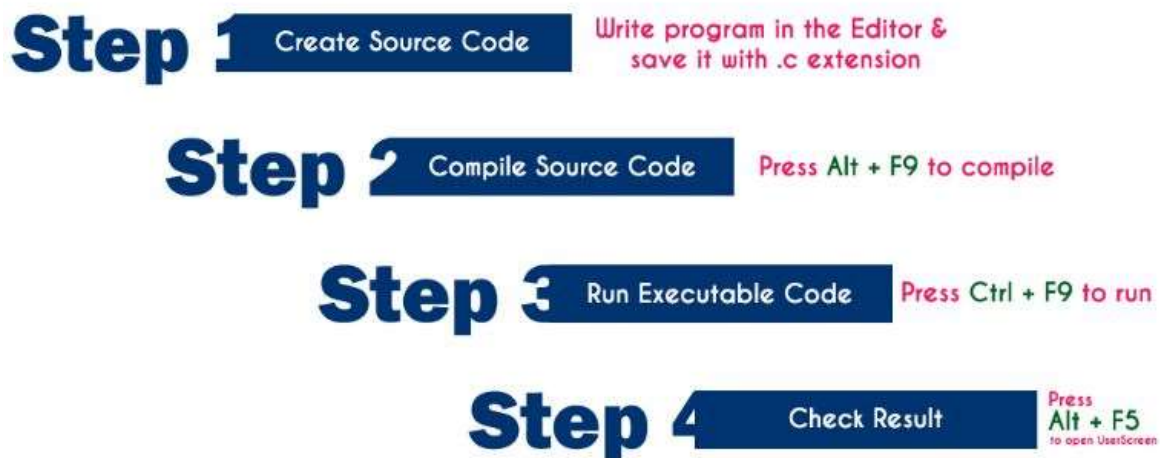
After writing C program when we compile and execute our program, there are various types of files are created.

1. **Source file(.c)**: These files contain function definitions, and the entire program logics, these files are human readable and by convention their names end with **.c**.
2. **Header file(.h)**: These files contain function prototypes and various pre-processor statements. They are used to allow source code files to access externally-defined functions and by convention their names end with **.h**.



3. **Object file(.obj):** These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves and by convention their names end with **.obj**.
4. **Binary executables file(.exe):** These files are produced as the output of a program called a “linker“. The linker links together a number of object files to produce a binary file that can be directly executed. It contains symbols that the linker can extract from the archive and insert into an executable as it is being built. And by convention their names end with **.exe** in windows.
5. **Bak file(.bak):** In programming, **".bak"** is a filename extension commonly used to signify a backup copy of a file. When a program is about to overwrite an existing file (for example, when the user saves the document they are working on), the program may first make a copy of the existing file, with **.bak** appended to the filename. This common **.bak** naming scheme makes it possible to retrieve the original contents of the file.

### STEPS USED IN COMPILING AND EXECUTING C PROGRAM



#### Step 1: Creating Source Code

Source code is a file with C programming instructions in high level language. To create source code, we use any text editor to write the program instructions. The instructions written in the

source code must follow the C programming language rules. The following steps are used to create source code file in Windows OS...

- Click on Start button
- Select Run
- Type cmd and press Enter
- Type cd c:\TC\bin in the command prompt and press Enter
- Type TC press Enter
- Click on File -> New in C Editor window
- Type the program
- Save it as FileName.c (Use shortcut key F2 to save)

### **Step 2: Compile Source Code (Alt + F9)**

Compilation is the process of converting high level language instructions into low level language instructions. We use the shortcut key Alt + F9 to compile a C program in Turbo C.

**Compilation is the process of converting high level language instructions into low level language instructions.**

Whenever we press Alt + F9, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into object code and stores it as file with .obj extension. Then the object code is given to the Linker. The Linker combines both the object code and specified header file code and generates an Executable file with .exe extension.

### **Step 3: Executing / Running Executable File (Ctrl + F9)**

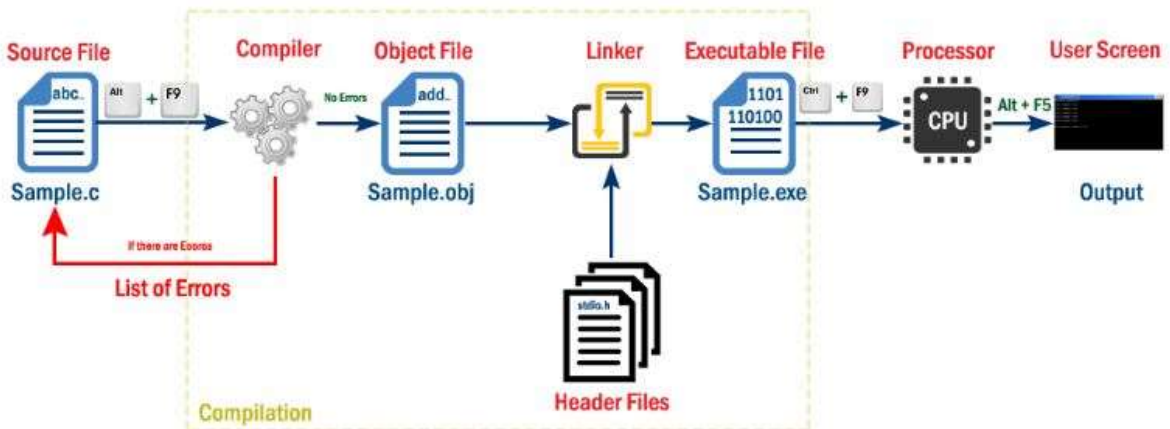
After completing compilation successfully, an executable file is created with .exe extension. The processor can understand this .exe file content so that it can perform the task specified in the source file.

We use a shortcut key Ctrl + F9 to run a C program. Whenever we press Ctrl + F9, the .exe file is submitted to the CPU. On receiving .exe file, CPU performs the task according to the instruction

written in the file. The result generated from the execution is placed in a window called User Screen.

### **Step 4: Check Result (Alt + F5)**

After running the program, the result is placed into User Screen. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key Alt + F5 to open the User Screen and check the result. When we execute a C program it undergoes with following process...



### **Important Points**

- C program file (Source file) must save with .c extension.
- Compiler converts complete program at a time from high level language to low level language.
- Input to the compiler is .c file and output from the compiler is .exe file, but it also generates .obj file in this process.
- Compiler converts the file only if there are no errors in the source code.
- CPU places the result in User Screen window.

### **Overall Process**

- Type the program in C editor and save with .c extension (Press F2 to save).
- Press Alt + F9 to compile the program.
- If there are errors, correct the errors and recompile the program.
- If there are no errors, then press Ctrl + F9 to execute / run the program.
- Press Alt + F5 to open User Screen and check the result.

### **IDENTIFIER**

An identifier is used for any variable, function, labels in your program etc.

In C language, an identifier is a combination of alphanumeric characters, i.e. first begin with a letter of the alphabet or an underline, and the remaining are letter of an alphabet, any numeric digit, or the underline.

### **Rules for naming identifiers / variable**

The rules that must be followed while naming the identifiers are as follows –

- The case of alphabetic characters is significant. For example, using "TUTORIAL" for a variable is not the same as using "tutorial" and neither of them is the same as using "TutoRial for a variable. All three refer to different variables.
- There is no rule on how long an identifier can be. We can run into problems in some compilers, if an identifier is longer than 31 characters. This is different for the different compilers.
- A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
- The first letter of an identifier should be either a letter or an underscore.
- You cannot use keywords like int, while etc. as identifiers.
- Identifiers must be unique

For example,

```
int student;  
float marks;
```

- Here, student and marks are identifiers.
- We have to remember that the identifier names must be different from keywords. We cannot use *int* as an identifier, because *int* is a keyword.
- Keywords are used along with identifiers to define them. Keywords explain the functionality of the identifiers to the compiler.

### **VARIABLE**

A **variable** in simple terms is a storage place which has some memory allocated to it. Basically, a variable used to store some form of data. Different types of variables require different amounts of memory, and have some specific set of operations which can be applied on them.

#### **Variable Declaration:**

A typical variable declaration is of the form:

***type variable\_name;***

or for multiple variables:

***type variable1\_name, variable2\_name, variable3\_name;***

example:

***int a;***

***or***

***int a,b,c;***

### **Types of Variables in C**

#### **1. Local Variable**

A variable that is declared and used inside the function or block is called local variable. It's scope is limited to function or block. It cannot be used outside the block. Local variables need to be initialized before use.

#### **2. Global Variable**

A variable that is declared outside the function or block is called a global variable.

It is declared at the starting of program. It is available to all the functions.

### **BASIC INPUT AND OUTPUT(formatted) STATEMENTS IN C**

C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output.

***scanf() :- standard input function***

The scanf() method, in C, reads the value from the console as per the type specified.

**Syntax:**



### How to take input and output of basic types in C?

The basic type in C includes types like int, float, char, etc. In order to input or output the specific type, the X in the above syntax is changed with the specific format specifier of that type.

The Syntax for input and output for these are:

- **Integer:**

**Input:** `scanf("%d", &intVariable);`

**Output:** `printf("%d", intVariable);`

- **Float:**

**Input:** `scanf("%f", &floatVariable);`

**Output:** `printf("%f", floatVariable);`

- **Character:**

**Input:** `scanf("%c", &charVariable);`

**Output:** `printf("%c", charVariable);`

**// C program to show input and output functions**

```
#include <stdio.h>
intmain()
{
    // Declare the variables
    intnum;
    charch;
    floatf;

    // Input the integer
    printf("Enter the integer: ");
    scanf("%d", &num);

    // Output the integer
    printf("\nEntered integer is: %d", num);

    // Input the float
    printf("\n\nEnter the float: ");
```

```
scanf("%f", &f);

// Output the float
printf("\nEntered float is: %f", f);
// Input the Character
printf("\n\nEnter the Character: ");
scanf("%c", &ch);
// Output the Character
printf("\nEntered integer is: %c", ch);
return 0;
}
```

### Output:

```
Enter the integer: 10
Entered integer is: 10
Enter the float: 2.5
Entered float is: 2.500000
Enter the Character: A
Entered Character is: A
```

The advanced type in C includes type like ***String***. In order to input or output the string type, the **X** in the above syntax is changed with the **%s** format specifier.

The Syntax for input and output for String is:

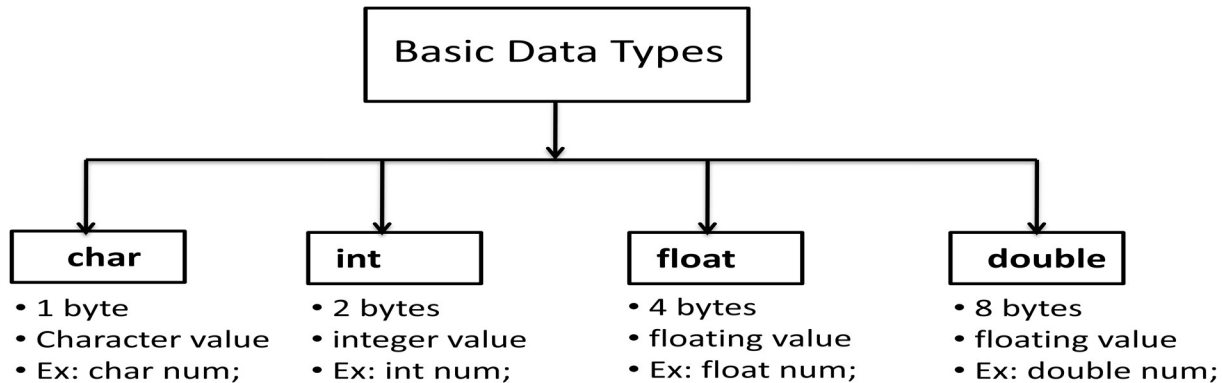
**Input:** `scanf("%s", stringVariable);`

**Output:** `printf("%s", stringVariable);`



### DATA TYPES IN C

Data type defines the type of data to be stored in the memory location; there are pre-defined data types in C which has fixed size memory location to store the data.



- 1. Character type:** It allows to store 1 byte character type data  
1 byte =  $2^8$  bits = 0 to 255 values can be stored  
Ex: A  $\rightarrow$  65  $\rightarrow$  01000001  
The ASCII value of A is 65 and it is stored in binary form ie 01000001
  - a) unsigned characters support values from 0 to 255
  - b) signed characters' support values from -128 to 127 (in a 2's complement)
- 2. Integer type:** it is used to whole number or integer numbers; it supports 4 different types of data. The size of the integer is machine dependent.
  - a) short int 

→ unsigned	→ 2 bytes ( 0 to 65535 )
→ signed	→ 2 bytes ( -32768 to 32765 )
  - b) int 

→ unsigned	→ 2 bytes ( 0 to 65535 )
→ signed	→ 2 bytes ( -32768 to 32765 )
  - c) long int 

→ unsigned	→ 4 bytes
→ signed	→ 4 bytes
- 3. float type:** It is single precision floating point number and it is used to store fractional values. The size of float is 4 bytes.
- 4. double type:** It is double precision floating point number and it is used to store fractional values. The size of double is 8 bytes.

### CONSTANTS

Constants are identifiers whose value does not change during the execution of the program. There are different types of constants such as:

1. integer constant
2. floating constant
3. character constant
4. constant expression
5. string constant
6. enumeration constant

1. **integer constant**: It consists of sequence of digits. There are different types of integer constant such as:

Name	Data type	Example
1. integer constant	int	1234
2. long integer constant	long int( l or L)	123456789L
3. unsigned integer	unsigned int (u or U)	1234
4. unsigned long integer	Unsigned long int(ul or UL)	123456789UL

Integer constant can be written in 3 different number systems

- a) **Decimal integer constant (base 10)**: digits from 0 to 9 must be used to represent decimal integer value

**ex: 1234 or 12 or 123456 or 1225679**

- b) **Octal integer constant (base 8)**: digits from 0 to 7 must be used to represent octal integer value but the first digit must be 0

**ex: 01 or 07 or 067 or 056**

- c) **Hexa decimal integer value (base 16)**: Any digits from 0 to 9 and A to F can be used to represent hexa decimal value but it should start with 0X or 0x

**Example:** 31 is decimal value, that can be written into octal value by dividing the number by 8 and it becomes 037 and the same thing can be converted into hexa decimal and it becomes 0x1F

**ie**  $31 \rightarrow 037 \rightarrow 0x1F$

octal or hexadecimal constant can be long int or unsigned long int.

2. **floating point constant:** It is single precision floating point number which contains a decimal point(123.45) or exponential(1e-2) ie ( $3 \times 10^5 \rightarrow 3e+5$ )

There are different types of floating constant such as:

Name	Data type	Example
1. floating point constant	float( f or F)	3.142f
2. floating point constant	double	3.142
3. long double	Long double(l or L)	3.142134567L

3. **Character constant:** A character enclosed in a single quote is called character constant

**For ex:** 'A', 'a', '0'

The value of character constant is the numeric value in ASCII format

Constant	ASCII value
'A'	65
'a'	97
'0'	48
' '	32

**Escape sequence:** Two characters can be enclosed in a single quotes or double quotes but it is considered as single character only

For ex:

Character name	Meaning
<code>\a</code>	Alert (bell) character
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\\</code>	Backslash
<code>\“</code>	Double quote
<code>\‘</code>	Single quote
<code>\?</code>	Question mark
<code>\000</code>	Octal number
<code>\xhh</code>	Hexa decimal number
<code>\0</code>	Null character

4. **Constant expression:** Any C valid expression can be defined as a constant called as defined constant.

**Example:** `# define pi 3.142`

The value for pi is defined as 3.142 so wherever I want to use the value, I can use the name pi to represent the value 3.142 in the program.

5. **String constant or string literal:** It is a sequence of zero or more characters that are enclosed in a pair of double quotes and each and every string constant should end with an NULL character.

**Example:1)** " "(empty string constant)

2) "welcome"(array of characters is also called as string constant)

w e l c o m e \0

NULL character

Here the compiler adds a *null character* at the end of the string in the memory

**Strlen():** In C we are using strlen function to find the length of the string

**Ex:** `char USN = "3BR14IS010";` the length of the string USN ie `strlen(USN)` is 10 characters ie 9 +1 null character.

6. **Enumeration constant:** It is a list of constant integer values which assigns values to name. use the keyword *enum* to represent enumeration constant

**Ex 1:** `enum Boolean = {No, Yes};` here it assigns integer value 0 to first name, and 1 to next name and so on.

**Ex 2:** `enum months = {Jan=1, Feb, Mar, Apr, . . . Dec};`

Here it assigns 2 to Feb & 3 to Mar and so on.

**Ex 3:** `enum days = {Monday=1, Tuesday, Wednesday. . . . Saturday};`

Here it assigns 2 to Tuesday & 3 to Wednesday and so on.

## **'C' KEYWORDS**

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

`int money;`

Here, *int* is a keyword that indicates *money* is a variable of type *int* (integer).

As C is a *case sensitive language*, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

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

## OPERATORS

Operators instruct the compiler to perform operation on some operand.

Operation instructions are specified by operators, in which operands can be variables, expressions or constants.

- ✓ Some operators operate on a single operand and they are called *Unary operators*.
- ✓ Most operators operate on two operands is called *Binary operator*.
- ✓ Operators operate on two operands is called *Ternary operator*.

### Types of operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators

4. Assignment Operators
5. Bitwise Operators
6. Increment and Decrement Operators
7. Conditional (ternary) operators

### 1. Arithmetic Operators

- The binary arithmetic operators are `+`, `-`, `*`, `/`, and the modulus operator `%`. Integer division truncates any fractional part.
- The expression `x%y`  
Produces the remainder when `x` is divided by `y`, and thus is zero when `y` divides `x` exactly the `%` operator cannot be applied to a `float` or `double`.
- The binary `+` and `-` operators have the same precedence, which is lower than the precedence of `*`, `/` and `%`, which is in turn lower than unary `+` and `-`. Arithmetic operators associate left to right.

Operator	Meaning	Example
<code>+</code>	Binary plus	<code>c = a + b</code>
<code>-</code>	Binary minus	<code>c = a - b</code>
<code>*</code>	Multiplication	<code>c = a * b</code>
<code>/</code>	Division	<code>c = a / b</code>
<code>%</code>	Modulus	<code>c = a % b</code>

### 2. Relational Operators

- These operators are used to make comparison between two expressions. All these operators are binary and require two operands. These comparisons can be done with the help of relational operators. Each one of these operators compares its left hand side operand with its right hand side operand.

Operator	Meaning	Example
<	Less than	$a < b$
>	Greater than	$a > b$
<=	Less than or equals to	$a < = b$
>=	greater than or equals to	$a > = b$
==	equals to	$a = = b$
!=	Not equals to	$a != b$

### 3. Logical Operator

These operators are used to combine one or more conditions

Operator	Meaning	Example
&&	Logical AND	$(A \&\& B)$ is false
	Logical OR	$(A    B)$ is true
!	Logical NOT	$!(A \&\& B)$ is true

The first two operator && and || are binary, whereas exclamation (!) is an unary operator.

**Ex: Logical AND**

**$a > b \&\& x = = 10$**

The expression on the left is  $a > b$  and that on the right is  $x = 10$ . The whole expression evaluates to true only if both expressions are true

**Ex: Logical OR**

**$a < b || a < n$**

the expression is true if one of them is true or both of them are true.



### 4. AssignmentOperator ( = )

- The equals ( = ) sign is used for assigning a value to a variable

**Syntax:** *VariableName* = *expression*;

The left hand side has to be a variable and the right hand side has to be a valid expression( often called value )

**Ex:** `a=32`/\* constant value 20 is assigned to variable a\*/

`b = z+10*a`;/\* is an expression\*

`c=sqrt(2)`;/\* it is a square root built in function \*/

### 5. Bitwise Operator

Operator	Meaning	Example
&	Bitwise AND	a & b
	Bitwise OR	a   b
^	Bitwise Ex-OR	a ^ b
~	Bitwise compliment	~a
<<	Shift left	a << 1
>>	Right shift	a >> 1

### 6. Increment and Decrement Operators

- C offers two unusual unary operators for incrementing and decrementing variables. These are ++ and – operators and are known as increment and decrement operators respectively. These operators increase or decrease the value of a variable on which they operate by one.

**Ex:** `int a = 5`

Operator	Meaning	Example
++a	Pre-increment	++a = 6
a++	Post-increment	a++ = 5
--a	Pre-decrement	--a = 4
a--	Post-decrement	a-- = 5

- In the first case the value of a after the execution of this statement will be 11. Since b is incremented and then assigned. In the second case the value of a will be 10 since it is assigned first and then incremented.

### 7. Conditional (ternary) operator

- ✓ An alternate method to using a simple if – else construct is the conditional expression operator ?. It is called the ternary operator, which operates on three operands.

**Syntax:** *expression1 ? expression 2 : expression 3*

- ✓ Here the expression1 is evaluated first; if it is true then the value of expression2 is the result. Otherwise the expression3 is the result.

**Ex:** *( a > b ) ? a : b*

### EXPRESSIONS

- ✓ An expression is a combination of variables, constants and operators written according to the syntax of the language.
- ✓ Every expression evaluates to a value **i.e.** every expression results in some value of a valid data type, that can be assigned to a variable

**Ex:** *a+b*

*a+200+30*

*c+b\*z*

*z+200+c/3*

## PRECEDENCE & ASSOCIATIVITY TABLE

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= &= ^=  = <<= >>=	right to left
,	left to right

10 - 3 % 8 + 6 / 4

\_\_\_\_\_

10 - 3 + 6 / 4

\_\_\_\_\_

10 - 3 + 1

\_\_\_\_\_

7 + 1

\_\_\_\_\_

8

17 - 8 / 4 \* 2 + 3 - ++a

\_\_\_\_\_

17 - 8 / 4 \* 2 + 3 - 6

\_\_\_\_\_

17 - 2 \* 2 + 3 - 6

\_\_\_\_\_

17 - 4 + 3 - 6

\_\_\_\_\_

13 + 4 - 6

\_\_\_\_\_

16 - 6

\_\_\_\_\_

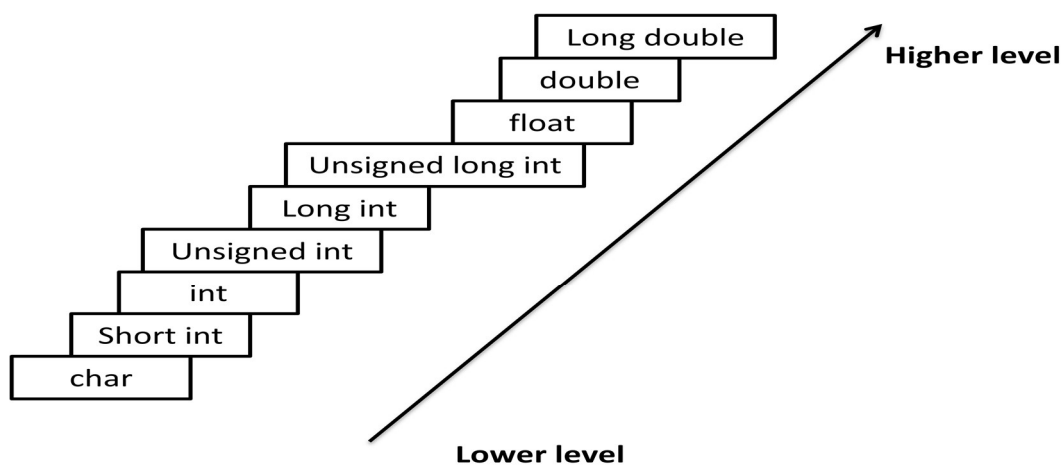
10

### TYPE CONVERSION AND TYPE CASTING

Type conversion or casting of variables refers to changing a variable of one data type into another. Type conversion is done implicitly whereas typecasting has to be done explicitly by the programmer.

### TYPE CONVERSION

Type Conversion is done when the expression has variables of different data types. To evaluate the expression, the data type is promoted from a lower to higher level. The hierarchy is given below:



**Fig: Conversion Hierarchy of Data types**

Type Conversion is automatically done when we assign an integer value to a floating point variable.

```
float x;  
int y=3;  
x = y;
```

Now,  $x=3.0$ , as the integer value is automatically converted into equivalent floating point representation. But when a float is converted into int type then there will be loss of value.

Consider the following group of statements

```
float f=3.5;  
int i;  
i = f;
```

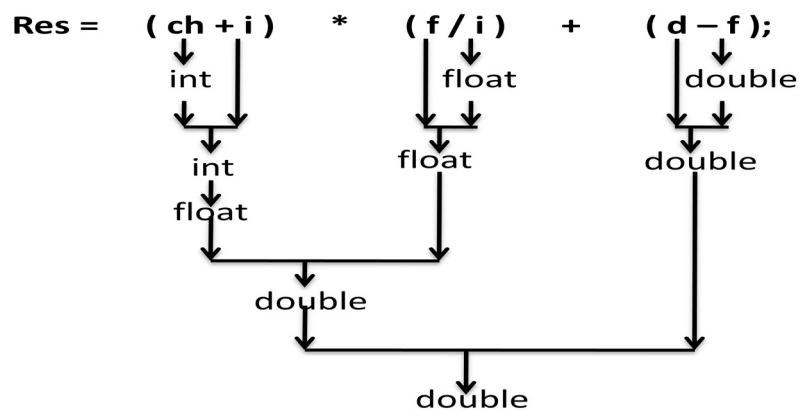
The statement `i = f` results in `f` to be demoted to type `int` the fractional part of `f` will be lost and `i` will contain 3 (not 3.5). In this case demotion takes place a higher-level data type is converted into lower type, some information is lost.

```
char ch;
```

```
int i;
```

```
float f;
```

```
double d, res;
```



**Fig: Type Conversion**

## TYPECASTING

Typecasting is also known as forced conversion. Typecasting an arithmetic expression tells the compiler to represent the value of the expression in a certain way.

It is done when the value of a higher data type has to be converted into the value of a lower data type.

Ex: if we need to explicitly typecast an integer variable into a floating-point variable, then the code to perform type casting can be given as,

```
float salary = 1000.00;  
int sal;  
sal = (int) salary;
```

when a floating point number is converted into integer type then the digits after the decimal point will be lost.

```
Ex: int a = 500, b = 70;
```

```
Float res;
```

```
res = (float) a / b;
```

**ex: program to convert a floating-point number into integer**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    float f_num;
    int i_num;
    clrscr( );
    printf(" Enter any floating point number");
    scanf ( "%f", &f_num);
    i_num = (int) f_num;
    printf(" The integer value of float is %d", i_num);
    getch( );
}
```

**Output:**

Enter any floating-point number: 23.45

The integer value of float is: 23

## DECISION CONTROL & LOOPING STATEMENTS

### Statements and Blocks

- ✓ An expression such as `x = 0` or `i++` or `printf(...)` becomes a **statement** when it is followed by a semicolon, as in

```
x = 0;
```

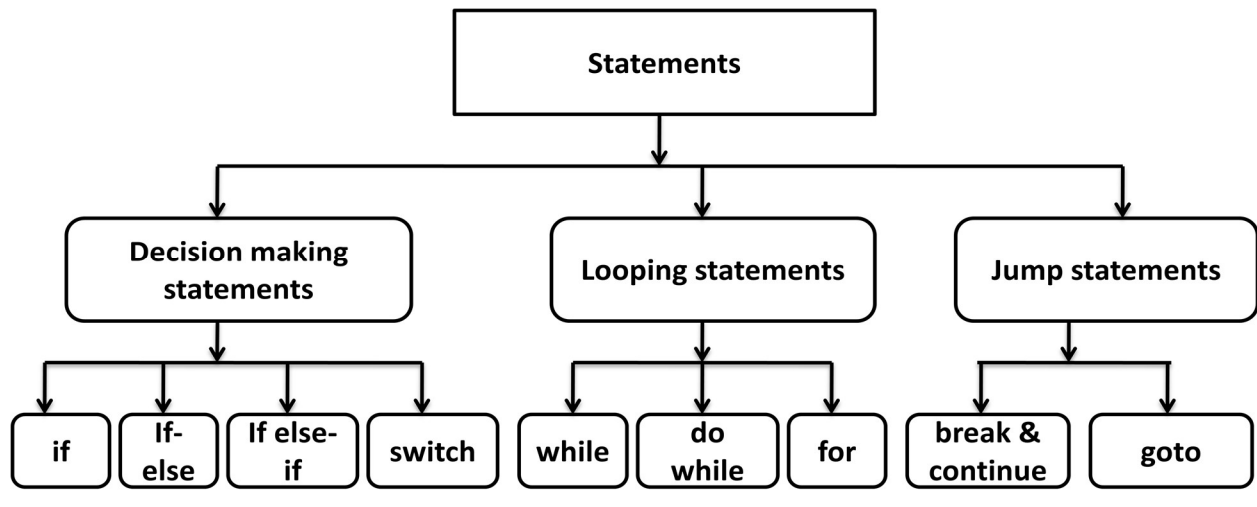
```
i++;
```

```
printf(...);
```

In C, the semicolon is a statement terminator

- ✓ **Braces** {and} are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around

multiple statements after an if, else, while, or for are another. (Variables can be declared inside any block;



### CONDITIONAL BRANCHING/ DECISION MAKING STATEMENTS

Conditional Branching Statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not.

These **decision control statements** include:

- if statement
- if - else statement
- if – else – if statement
- switch statement

#### 1. IF STATEMENT

- ✓ The if statement is the simplest form of decision control statement that is frequently used in decision making. The general form is:

```
if ( expression )
{
    Statement 1;
    .....
    Statement n;
```

}

- ✓ The if structure may include one statement or n statements enclosed within curly brackets. First the test expression is evaluated, if the test expression is true, the statement of **if** block (statements) are executed, otherwise these statements will be skipped and the execution will jump to statement x.
- ✓ The statement in an if construct is any valid C language expression that may include logical operators. Note that there is no semicolon after the test expression. This is because the condition and statement should be placed together as a single statement.

**ex: program to determine whether a person is eligible to vote**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int age;
    printf("Enter the age");
    scanf("%d ", &age);
    if ( age > = 18 )
        printf("You are eligible to vote");
    getch( );
}
```

**Output:**

```
Enter the age: 30
You are eligible to vote
```

### NESTED-IF IN C

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, C allow us to use nested if statements within if statements, i.e, we can place an if statement inside another if statement.



Syntax:

```
if (condition1)
{
    // Executes when condition1 is true

    if (condition2)
    {
        // Executes when condition2 is true
    }
}

// C program to illustrate nested-if statement
#include <stdio.h>
#include <conio.h>
voidmain()
{
    inti = 10;

    if(i == 10)
    {
        // First if statement
        if(i < 15)
            printf("i is smaller than 15\n");

        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if(i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }

    getch();
}
```

### 2. IF ELSE STATEMENT

The `if-else` statement is used to express decisions. Formally the syntax is

```
if (expression)  
    statement1  
else  
    statement2
```

- ✓ Where the `else` part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement1* is executed. If it is false (*expression* is zero) and if there is an `else` part, *statement2* is executed instead.
- ✓ The statement in an `if` construct is any valid C language expression that may include logical operators. Note that there is no semicolon after the test expression. This is because the condition and statement should be placed together as a single statement.

```
if ( a > b )  
    z = a;  
else  
    z = b;
```

**ex: program to determine the largest of two numbers**

```
#include<stdio.h>  
#include<conio.h>  
void main( )  
{  
    int age, b, large;  
    printf("Enter the value of a and b");  
    scanf("%d %d", &a, &b);  
    if ( a > b )  
        large = a;  
    else  
        large = b;  
    printf("large: %d", large);  
    getch();  
}
```

### Output:

```
Enter the value of a and b : 12    23
large: 23
```

### 3. IF – ELSE - IF STATEMENT

- ✓ In if statement, if the result is true, the statements followed by the expression is executed else if the expression is false, the statement is skipped by the compiler.
- ✓ However, if we want a separate set of statements to be executed if the expression returns a zero value?. In such cases we use an if-else statement rather than using simple if statement. The general form of a simple if-else statement is shown below:

```
if (expression)
    statement
else if(expression)
    statement
else if(expression)
    statement
else if (expression)
    statement
else
    statement
```

occurs so often that it is worth a brief separate discussion.

- ✓ This sequence of `if` statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if an *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group of them in braces. The last `else` part handles the "none of the above" or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else
    statement
```

- ✓ Can be omitted, or it may be used for error checking to catch an "impossible" condition.

**ex: program to determine the largest of three numbers**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a, b, c, large;
printf("enter the value of a ,b and c");
scanf("%d %d %d", &a, &b, &c);
    if ( ( a > b ) && ( a > c ) )
        large = a;
    else if ( ( b > a ) && ( b > c ) )
        large = b;
    printf("large: %d", large);
getch();
}
```

**Output:**

```
Enter the value of a and b : 12    23    55
large: 55
```

### **SWITCH STATEMENT**

- ✓ The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression)
{
    Case const-expr: statements; break;
    Caseconst-expr:statements; break;
    Caseconst-expr:statements; break;
    Caseconst-expr:statements; break;
    Caseconst-expr:statements; break;
    default: statements ; break;
}
```

- ✓ Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different.
- ✓ The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.
- ✓ The `break` statement causes an immediate exit from the `switch`. Because cases serve just as labels, after the code for one case is done. As a matter of good form, put a `break` after the last case (the `default` here) even though it's logically unnecessary.

**Ex: Program to demonstrate the use of switch case**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    char    ch = 'C';
    switch ( ch )
    {
        case 'O': printf("Outstanding\n");
                    break;
        case 'A': printf("Excellent\n");
                    break;
        case 'B': printf("Good\n");
                    break;
        case 'C': printf("Fair\n");
                    break;
        case 'F': printf("Fail\n");
                    break;
        default: printf("Invalid Grade\n");
                    break;
    }
```

**Output:** Fair

**The following rules should be followed before using switch:**

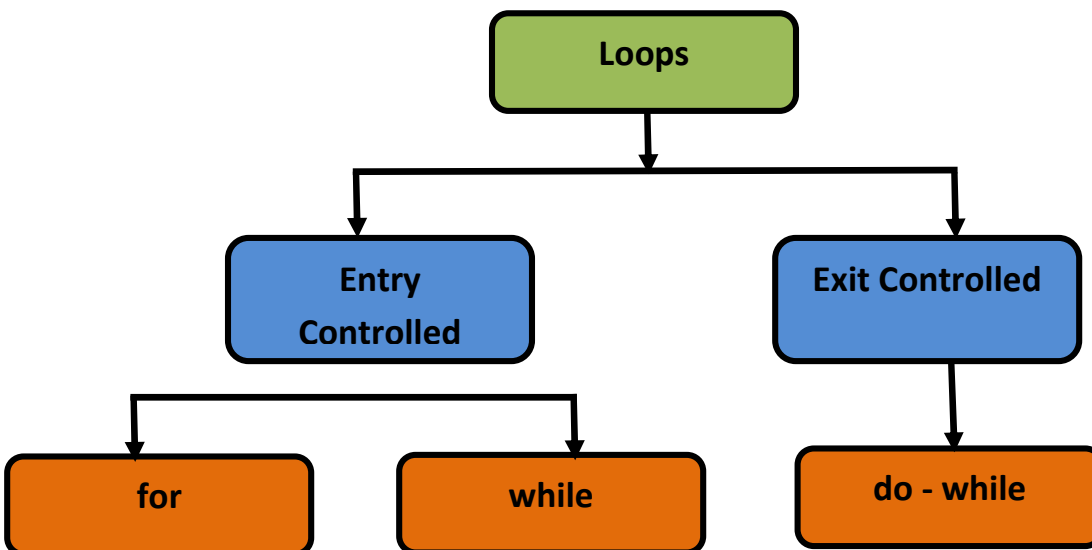
- The control expression that follows the keyboard switch must be of integer type ( ie either be an integer or any value that can be converted to an integer )
- Each case label should be followed with a constant or a constant expression.
- Every case label must evaluate to a unique constant expression value.
- Case labels must end with a colon.
- Two case labels may have the same set of actions associated with it.
- The default label is optional and is executed only when the value of the expression does not match with any labeled constant expression. It is recommended to use default case in every switch case statement.
- There can be only one default label in a switch statement.

**Advantages of using switch case statement**

- ✓ Easy to debug.
- ✓ Easy to read and understand.
- ✓ Executes faster than if – else statement.

### **LOOPS (ITERATIVE STATEMENTS)**

- Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression.



- A loop is a sequence of instructions that is repeated until a certain condition is reached.

There are mainly two types of loops:

- **Entry Controlled loops:** In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry-controlled loops.
- **Exit Controlled Loops:** In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute atleast once, irrespective of whether the test condition is true or false. **do – while loop** is exit controlled loop.

### 1. WHILE LOOP

The while loop repeats one or more statements while a particular condition is true.

```
while (expression)  
    statement
```

- The *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*.
- In the while loop, the condition is tested before any of the statements in the statement block is executed.
- If the condition is true, only then the statements will be executed **or** else the control will jump to the next statements outside the while loop.
- A while loop is also known as top checking loop. Since the control condition is placed as the first line of the code.
- If the control condition evaluates to false, then the statements enclosed in the loop are never executed.

**Ex: program to demonstrate while loop to print first 10 numbers.**

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

```
int i = 0;
while ( i <= 10 )
{
    printf(" %d", i);
    i = i + 1;
}
getch( );
}
```

**Output:** 0 1 2 3 4 5 6 7 8 9 10

## 2. DO - WHILE LOOP

- The do while loop is similar to the while loop. The only difference is that in a do while loop, the test condition is tested at the end of the loop.
- Here the test condition is tested at the end, it means that the body of the loop gets executed at least once.( even if the condition is false ).

The syntax of the do is

```
do
    statement
while (expression);
```

- The *statement* is executed, and then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates.
- Note that the test condition is enclosed in parentheses and followed by semicolon.
- Similar to the while loop the do while loop continues to execute a condition is true.
- Do - while loop is a bottom – checking loop, since the control expression is placed after the body of the loop.
- The **major advantage** of using a do – while loop is that it always executes at least once, even if the user enters invalid input, the loop will execute. One complete execution of the loop takes place before the first comparison is been done.



**Ex: program to demonstrate do - while loop to calculate the sum of first n numbers.**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int  n, i = 0, sum = 0;
    printf("Enter the value of n :");
    scanf("%d", &n);
    do
    {
        sum = sum + i;
        i = i + 1;
    } while ( i <= n );
    printf("The sum of n natural numbers is %d", sum);
}
```

**Output:** Enter the value of n: 18

The sum of n natural numbers is: 171

### **3. FOR LOOP**

- The for loop is used to execute single or group of statements a limited number of times.
- In for loop the condition is tested before the statements contained in the body are executed.
- If the condition does not hold true, then the body of the loop may never get executed.

**The for statement**

```
for (expr1; expr2; expr3)
    statement
```

is also equivalent to

```
for( initialization; condition; update )
```

```
{  
    Statements;  
}
```

- When a for loop is used, the loop variable is initialized only once.
- With every iteration of the loop, the value of the loop is updated and the condition is checked.
- If the condition is true, the statement block of the loop is executed, else the statements comparing the statement block of the for loop are skipped and the control jumps to the immediate statement following the for loop body.
- In the syntax of the for loop, initialization of the loop variable allows the programmer to give it a value.
- The condition specifies that until the condition is true the statements in the for loop will be executed.
- Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting some other value like,  $i+=2$ , where 'i' is the loop variable.
- Note that every section of the for loop is separated from the other with a semicolon.

**Ex: program to demonstrate for loop to print first n numbers**

```
#include<stdio.h>  
#include<conio.h>  
main( )  
{  
    int i, n;  
    printf("Enter the value of n");  
    scanf("%d",&n);  
    for ( i =0; i <= n; i++ )  
        printf(" %d ", i );  
    return 0;  
}
```

- ✓ In this code, **i** is the loop variable. Initially it is initialized with value zero.
- ✓ Suppose the user enters 10 as the value for **n**. then the condition is checked

- ✓ Since the condition is true as **i** is less than **n**, the statement in the for loop is executed and the value of **i** is printed.
- ✓ After every iteration the value of **i** is incremented.
- ✓ When **i = n** the control jumps to the return 0 statement.

### **NESTED LOOPS**

**Nested loop** means a loop statement inside another loop statement. That is why nested loops are also called as “**loop inside loop**”.

Syntax for Nested For loop:

```
for ( initialization; condition; increment )  
  
{  
  
    for ( initialization; condition; increment )  
  
    {  
  
        // statement of inside loop  
  
    }  
  
    // statement of outer loop  
  
}
```

Syntax for Nested While loop:

```
while(condition)  
  
{  
  
    while(condition)  
  
    {  
  
        // statement of inside loop  
  
    }  
  
    // statement of outer loop  
  
}
```

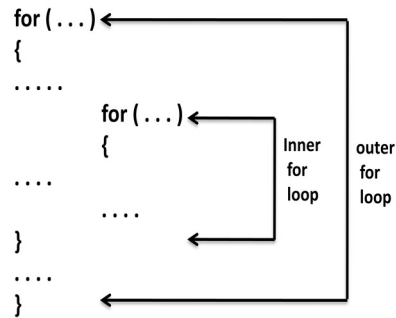
### Syntax for Nested Do-While loop:

```
do
{
    do
    {
        // statement of inside loop
    }while(condition);
    // statement of outer loop
}while(condition);
```

**Note:** *There is no rule that a loop must be nested inside its own type. In fact, there can be any type of loop nested inside any type and to any level.*

### Syntax:

```
do
{
    while(condition)
    {
        for ( initialization; condition; increment )
        {
            // statement of inside for loop
        }
        // statement of inside while loop
    }
    // statement of outer do-while loop
}while(condition);
```



\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

```
#include<stdio.h>

void main ( )

{

int i, j;

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

    {

        printf("\n");

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

            printf("*");

    }

}
```

### Example 2: C Program to print half Pyramid of \*

```
*
**
***
****
*****

#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i)
    {
        for (j = 1; j <= i; ++j)
        {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

### Example 3: C Program to print half Pyramid of numbers

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
#include <stdio.h>
int main()
{
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i)
    {
        for (j = 1; j <= i; ++j)
        {
            printf("%d ", j);
        }
        printf("\n");
    }
    return 0;
}
```


## **BREAK AND CONTINUE STATEMENTS**

### **Break statement:**

- In C break statement is used to terminate the execution of the nearest en-closing loop in which it appears.
- The break statement is widely used in switch, for loop, while loop, do - while loop.
- When the compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears.
- The syntax is:


***break;***

```
while ( . . . )
{
.....
If ( condition )
break;
.....
}
```



(Transfer control out  
of the while loop )

```
for ( . . . )
{
.....
If ( condition )
break;
.....
}
```



(Transfer control out  
of the for loop )

**Ex: program to demonstrate the use of break statement**

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

int i = 0;
while ( i <= 10 )
{
    if ( i == 5 )
        break;
    printf("%d", i );
    i = i + 1;
}
return 0;
}
```

- Note that the code is meant to print first 10 numbers using a while loop, but it actually print only numbers from 0 to 4.
- As soon as i becomes equal to 5, the break statement is executed and the control jumps to the statement following the while loop.
- Hence the break statement is used to exit a loop from any point within its body, bypassing its normal termination expression.
- When a break statement is encountered inside a loop, the loop is immediately terminated and the control is passed to the next statement following the loop.



### Continue statement:

- ✓ Similar to break statement, the continue statement can only appear in the body of the loop. When the compiler encounters a continue statement then the rest of the statement in the loop are skipped and the control is unconditionally transferred to the loop continuation portion of the nearest enclosing loop.

- ✓ The syntax is

***continue;***

- ✓ When the continue statement is encountered in the while loop and in the do-while loop, the control is transferred to the code that tests the controlling expression.
- ✓ However, if it is placed with a for loop, the continue statement causes a branch to the code that updates the loop variable.

```
while ( . . . ) ←  
{  
.....  
If ( condition )  
continue;  
.....  
}  
.....
```

**(Transfer control to the  
conditional expression of  
the while loop )**

```
for ( . . . ) ←  
{  
.....  
If ( condition )  
continue;  
.....  
}  
.....
```

**(Transfer control to the  
conditional expression of  
the for loop )**

**Ex: program to demonstrate the use of continue statement**

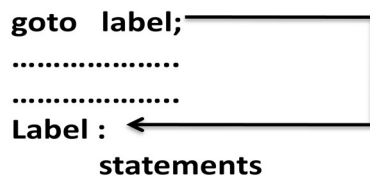
```
#include<stdio.h>  
#include<conio.h>  
int main( )  
{  
  
    int i ;  
    for ( i = 0; i <= 10; i++ )  
    {  
        if ( i == 5 )  
            continue;  
        printf("%d", i );  
    }  
}
```

```
        i = i + 1;
    }
    return 0;
}
```

- ✓ The code given here is meant to print numbers from 0 to 10
- ✓ But as soon as 'i' becomes equal to 5 the continue statement is encountered, so the rest of the statements in the for loop are skipped and the control passes to the expression that increments the value of i.
- ✓ The output of the program is:  
0 1 2 3 4 6 7 8 9 10
- ✓ Note that the no 5 is not printed as continue caused early incrementation of i and skipping of the statement that printed the value of i.

### **goto statement**

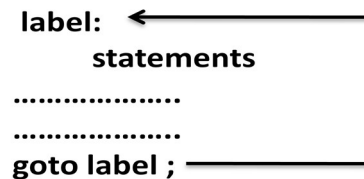
- The **goto** statement is used to transfer the control to a specified **label**; however, the label must reside in the same function and can appear only before one statement in the same function.
- The syntax of **goto** statement is shown below:



The diagram illustrates a forward jump. On the left, the code `goto label;` is shown, followed by three lines of dotted lines representing statements. An arrow points from the `label` in the `goto` statement to the `Label :` which is followed by three lines of dotted lines representing statements. The label and its statements are enclosed in a rectangular box.

```
goto label; .....
.....
Label : ← .....
        statements
```

**Fig: forward jump**



The diagram illustrates a backward jump. On the right, a rectangular box contains the code `label: ←` followed by three lines of dotted lines representing statements, and then `goto label ;`. An arrow points from the `goto label ;` statement back to the `label:` at the top of the box.

```
label: ← .....
        statements
.....
goto label ;
```

**Fig: Backward jump**

- Here **label** is an identifier that specifies the place where the branch is to be made.
- **Label** can be any valid variable name that is followed by a colon ( : )
- The **label** is placed immediately before the statement where the control has to be transferred.
- The **label** can be placed anywhere in the program either before or after the **goto** statement.

- Whenever the **goto** statement is encountered the control is immediately transferred to the statements following the label.
- Therefore, **goto** statement breaks the normal sequential execution of the program.
- If the label is placed after the **goto** statement, then it is called a **forward jump** and in case it is located before the **goto** statement, it is called **backward jump**.

**Ex: program to calculate the sum of all positive numbers**

```
#include<stdio.h>
void main( )
{
    int num, sum = 0;
read:
    printf("enter the number, enter 999 to end");
    scanf("%d",&num);
    if(num!=999)
    {
        if(num<0)
            goto read;
        sum+=num;
        goto read;
    }
    printf("Sum of the numbers entered by the user is
    %d",sum);
}
```

### ASSIGNMENT QUESTIONS

1. Explain the basic structure of C program
2. List and explain the files created while executing a C program
3. Explain the steps used in compiling & executing a C program
4. Define a variable? List out the rules for declaring a identifier
5. Explain the data types supported in C
6. List and explain different constants supported in C language
7. List and explain different types of operators in C
8. Explain type conversion and type casting
9. Explain decision making or conditional statements in C
10. Explain switch statement in C
11. Explain different types of looping statements in C
12. Explain break, continue and goto statements in C
13. Write a C program to print a given(any) pattern
14. Write a C program to check a given number is:
  - a. Palindrome
  - b. Armstrong Number
  - c. Perfect Number
  - d. Strong Number

\*\*\*\*\*End of Module-1\*\*\*\*\*