EST 102 CPC

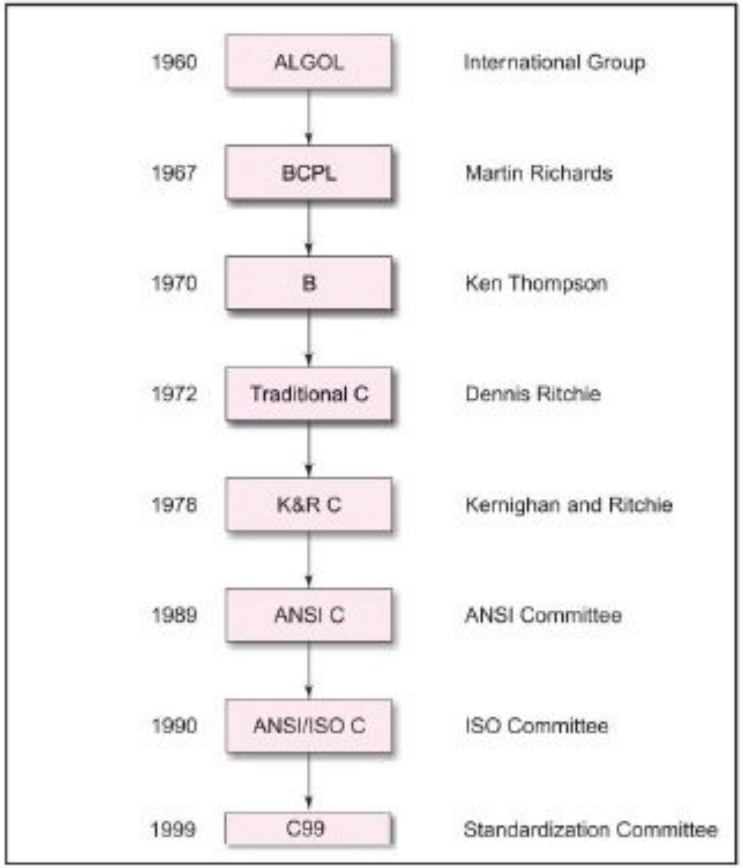
MODULE-2

Introduction to C programming

- C is a structured, high-level, machine independent programming language.
- It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.
- The root of all modern languages is **ALGOL**.
- BCPL (Basic Combined Programming Language) was developed for writing system software.
- Using many features of **BCPL,** a new language was created which was named B.
- C evolved from ALGOL, BCPL and **B** by **Dennis Ritchie** at Bell Laboratories in 1972.
- C uses many concepts from these languages and added the concept of data types and other powerful features.

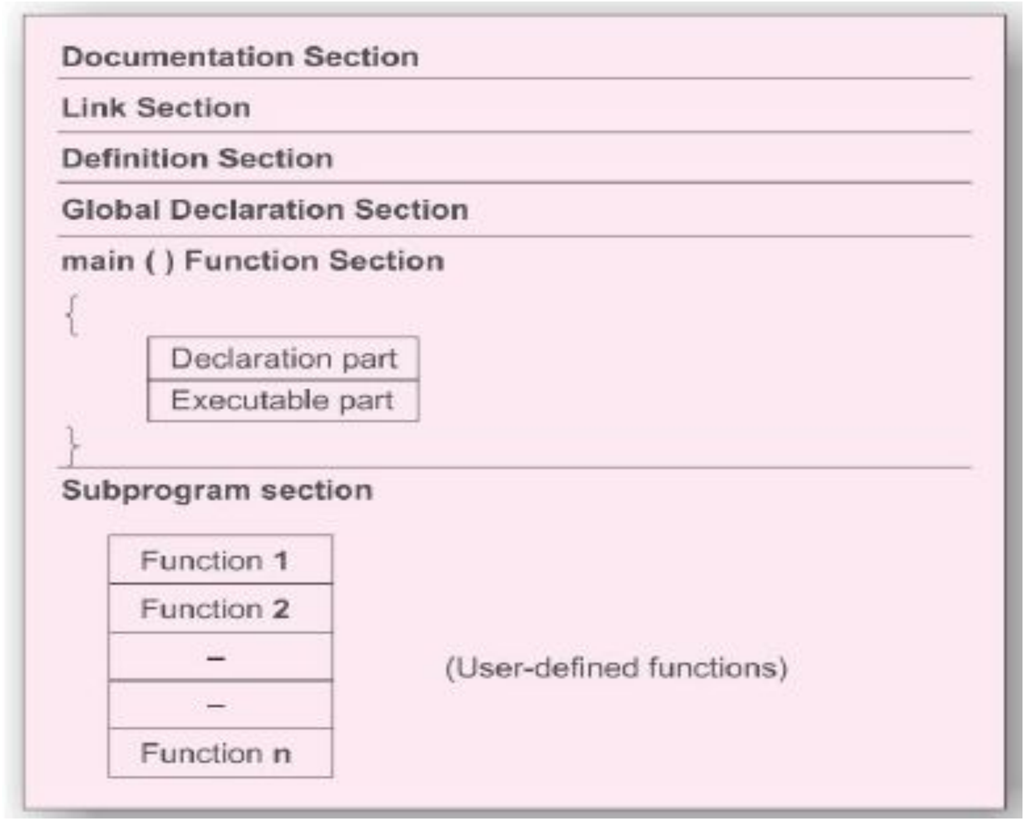
- C language became more popular with the publication of the book ‘The C Programming Language’ by Brian Kerningham and Dennis Ritchie.
- The book gained popularity and the language came to be known as “**K&R C**”.
- To ensure that the C language standard, ANSI approved a version of C which is called **ANSI C**.
- C language was again improved, enhanced and became an **ANSI/ISO** approved language.
- All popular computer languages are dynamic in nature.
- They continue to improve their power and scope by new features.
- The standardization committee of C added few features of C++ , Java to C to enhance usefulness of the language which is referred to as **C99**.

History of C Language:



Basic Structure of C programs:

- A C-program may contain one or more sections as shown below;



- The **documentation section** consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
- The **link section** provides instruction to the compiler to link functions from the system library.
- The **definition section** defines all symbolic constants.
- There are some variables that are used in more than one function which are called global variables and are declared in the global declaration section, that is outside of all the functions.
- Every C program must have one **main() function** section.
  - This section contains two parts;
    - **Declaration part**: declares all variables used in executable part.
    - **Executable part:** There is at least one statement in the executable part.
  - These 2 parts appear between the opening and closing braces.
  - The program execution begins at the opening brace and ends at the closing brace.
  - The closing brace of main() section is the logical end of the program.
  - All statements in the declaration and executable parts end with a semicolon(;).

- The **subprogram section** contains all the user-defined functions that are called in the main function.
  - These are placed immediately after the main function.
  - They can appear in any order.
- A function is a subroutine that may include one or more statements designed to perform a specific task.
- To write a C program, we create functions and then put them together.

Sample C-program to print a message:

```
#include<stdio.h>                          //pre-processor directive
main()                                     //main function declaration
{

  printf(“Hello World”);                   //to output the string on display


}
```

The #include directive

- **#include** is a **preprocessor directive.**
  - C-programs are divided into modules or functions.
  - Some functions are written by users and some others stored in C library.
  - **Library functions** are grouped category-wise and are stored in different files known as **header files.**
  - If we want to access functions stored in library, it is necessary to tell the compiler about the files to be accessed.
  - This is achieved by using the preprocessor directive #include as follows:

    #include<filename>

  - Filename is the name of the library file that contain the required function definition.
  - Preprocessor directives are placed at the beginning of a program.

- Following are some of the commonly used header files;

<stdio.h>	Standard I/O library functions
<math.h>	Mathematical functions
<string.h>	String manipulation functions
<time.h>	Time manipulation functions
<ctype.h>	Character testing & conversion functions
<stdlib.h>	Utility functions

- The pre-processor directive tells the compiler to include the mentioned header file in the program.
- stdio.h is a header file that contains the definitions of common input output functions such as scanf() and printf() etc. It activates keyboard and monitor.

The main() function

- main() is a part of every C-program.
- C permits different forms of main statement.
    - **main()**
    - **int main()**
    - **void main()**
    - main(void)
    - void main(void)
    - int main(void)
- The empty pair of parentheses indicates that the function has no arguments.
- This may be explicitly indicated by using the keyword void inside the parentheses.
- We may also specify the keyword int or void before the word main.
- The keyword **void** means that the **function does not return any valu**e to the operating system and **int** means that the **function returns an integer value** to the operating system.
- When **int** is specified , the last statement in the program must be **“return 0”**

Comments:

- **Single line comment** is represented using **//**
- The **lines beginning with /* and ending with */** are known as **comment lines**.
- Comment lines enhances the readability.
- Comment lines are **not executable** statements and therefore anything between /* and */ is ignored by the compiler.
- Comments cannot be nested in C.

Predefined functions:

- Functions that has already been written, compiled and linked together with our program at the time of linking.
- Eg: **printf** and **scanf**
- **Printf-**causes everything between the starting and ending quotation marks to be printed out.
- Consider the statement below:

  printf(“Hello world!!”);

  It produces the following output;

  Hello world!!

- The information contained between the parentheses is called the argument of the function.

Note:

- Every statement in C should end with a semicolon (;) mark.
- A newline character **(\n)** instructs the computer to go to the next(new) line.
- Horizontal tab character **(\t)** instructs the computer to leave one tab horizontal space.
- C language make distinction between uppercase and lowercase letters.
    - printf and PRINTF are not the same.
    - In C, everything is written in lowercase letters.
    - However uppercase letters are used for symbolic names representing constants.
    - Upper case letters are also used in output strings.

CHARACTER SET:

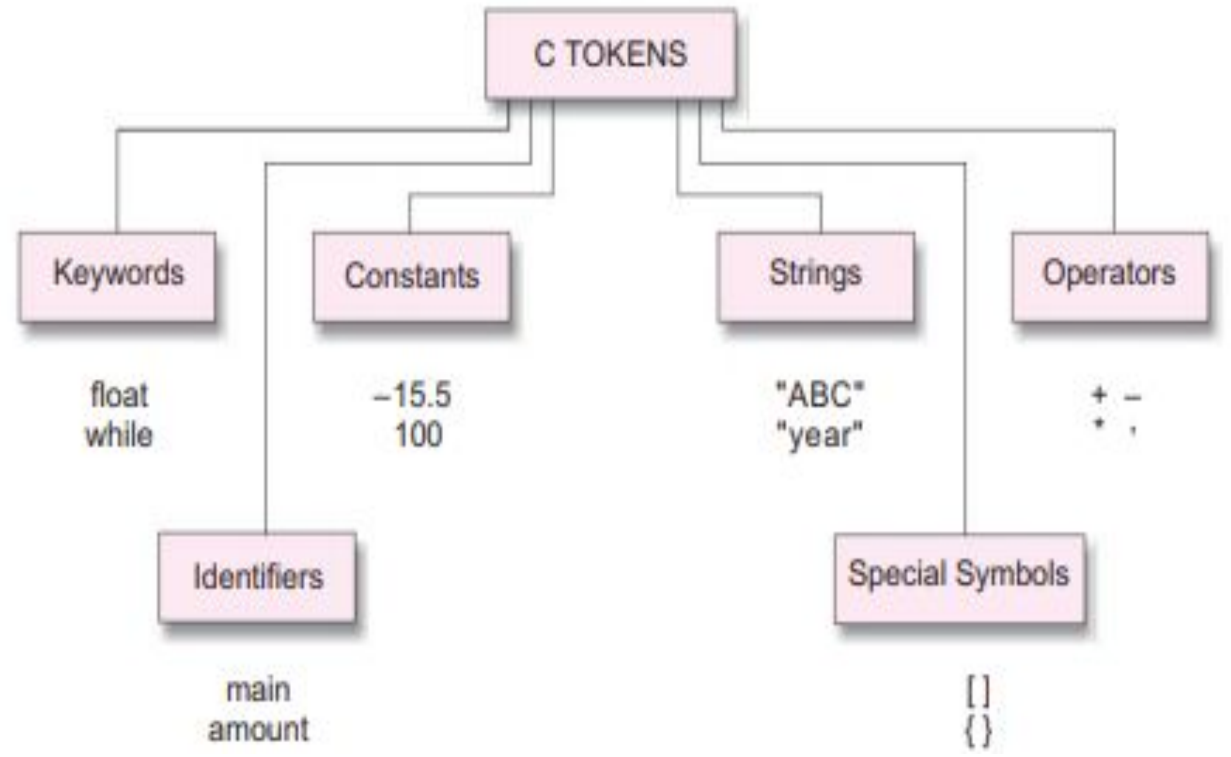
- The characters that can be used to form words, numbers, expressions depend upon the computer on which the program is run.
- The characters in C are grouped into the following categories:
  - Letters
  - Digits
  - Special characters
  - White spaces
- The compiler ignores white spaces unless they are a part of a string constant, but are prohibited between the characters of keyword and identifiers.

Letters		*Digits*
Uppercase A.....Z		All decimal digits 0 9
Lowercase a.....z		
	Special Characters	
, comma		& ampersand
. period		^ caret
; semicolon		* asterisk
: colon		– minus sign
? question mark		+ plus sign
' apostrophe		< opening angle bracket
" quotation mark		(or less than sign)
! exclamation mark		> closing angle bracket
| vertical bar		(or greater than sign)
/ slash		(left parenthesis
\ backslash		) right parenthesis
~ tilde		[left bracket
_ under score		] right bracket
$ dollar sign		{ left brace
% percent sign		} right brace
		# number sign
	White Spaces	
	Blank space	
	Horizontal tab	
	Carriage return	
	New line	
	Form feed	

C TOKENS:

- Tokens are **basic building blocks** or smallest individual units of C language.
- They are used together to write a C program.
- C has 6 types of tokens:
  - Identifiers
  - Keywords
  - Constants
  - Strings
  - Special symbols
  - Operators

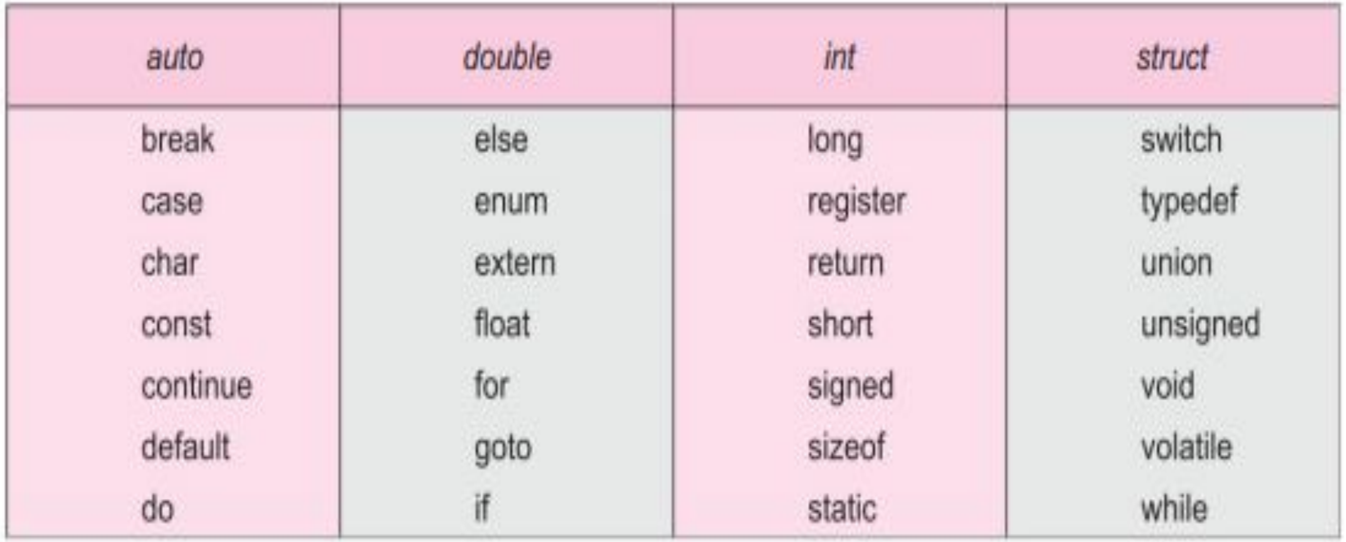
C tokens and examples:



Keywords:

- Keywords serve as basic building blocks for program statements.
- All keywords must be written in lowercase.
- Keywords will be having some specific usage associated with them.
- They have some standard, predefined meanings in C.
- These can be used only for their intended purpose.
- They can’t be used for some other purpose.
    - For eg: “default” is a keyword in C.
    - So, in a C program, we can’t use *default* as an identifier or a variable.
- The standard ANSI C supports only 32 keywords.
- Depending on the compiler used, there can be additional keywords.

The 32 keywords in ANSI C:



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

Identifiers:

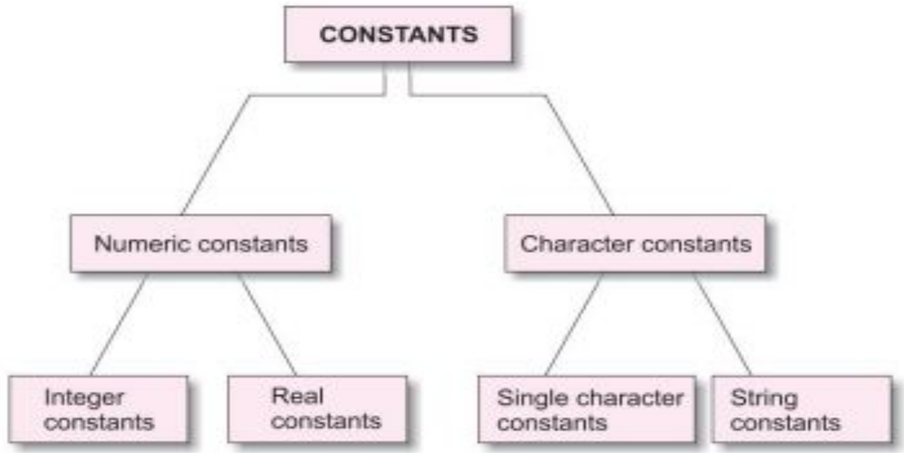
- Identifiers are the names given to various program elements like variables, constants, functions, arrays etc.,
- These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character.
- Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used.
- The underscore character is also permitted in identifiers.
    - It is usually used as a link between two words in long identifiers.
- Generally, it is recommended that the identifier should represent the purpose for which it is used.

Rules for Identifiers:

1. First character must be an alphabet(or underscore).
2. Must consist of only letters, digits or underscore.
3. Cannot use a keyword as an identifier.
4. Must not contain white space.
5. Only first 31 characters are significant.(Most of the C compilers allow a maximum of 31 characters in an identifier.)

Constants:

- Constants in C refer to fixed values that do not change during execution of a program.
- C supports several types of constants as illustrated below;



Numeric constants:

- Integer constants and real constants(floating point constants) are collectively called numeric constants,

Integer constants:

- Integer constant refers to a sequence of digits.
- There are 3 types of integers: decimal integers, octal integer and hexadecimal integer.
- Decimal integers have set of digits , 0 to 9, preceded by an optional + or - sign.
    - Valid examples of decimal integer constants are

      426   +859   0   85963   +78

- Commas, embedded spaces, non-digit characters are not permitted between digits. Eg: 15 720   20,000 $23 are illegal numbers..

Octal Integer constants:

- Consist of any combination of digits from 0 to 7, with a leading 0.
    - Eg: 037   0   0435   0521

Hexadecimal Integer constants:

- A sequence of digits preceded by 0x or 0X.
- They may include alphabets A through F or a through f.
- The letters A to F and a to f represents numbers 10 through 15.
- Eg: 0X2   0x9F   0x
- We rarely use octal and hexadecimal numbers in programming.

Real constants:

- Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, prices, temperatures, prices and so.
- These quantities are represented by **numbers containing fractional parts.**Such numbers are called real (or **floating point**) constants.
- Eg: 17.548 0.0083 435.26 -0.26 +2.3
- These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part.
- It is possible to omit digits before the decimal point.
  - ie, .95 +.788 -.5 are all valid.
- A real number may also be expressed in exponential notation.

Character constants:

Single character constants:

- A single character constant contains a single character enclosed within a pair of single quote marks.
- Example of character constants are:

  ‘5’ ‘X’ ‘;’ ‘ ‘

- The character constant ‘5’ is not the same as the number 5.
- Character constants have integer values known as ASCII values.
- For eg; printf(“%d”, ‘a’); would print the number 97, the ASCII value of the letter a.
- The statement printf(“%c”, ‘97’); would output the letter ‘a’.
- Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants.

String Constants:

- A string constant is a **sequence of characters enclosed in double quotes.**
- The characters may be letters, numbers, special characters and blank space.
- Eg: “Hello!”   “1996”   “5+6”   “X”

VARIABLES:

- A variable is a data name that is **used to store a data value.**
- A variable is an **identifier that is used to represent a single data item.**
- A variable has to be declared properly before it is used in a statement within a program.
- The general form of variable declaration is;

data-type variable-list;

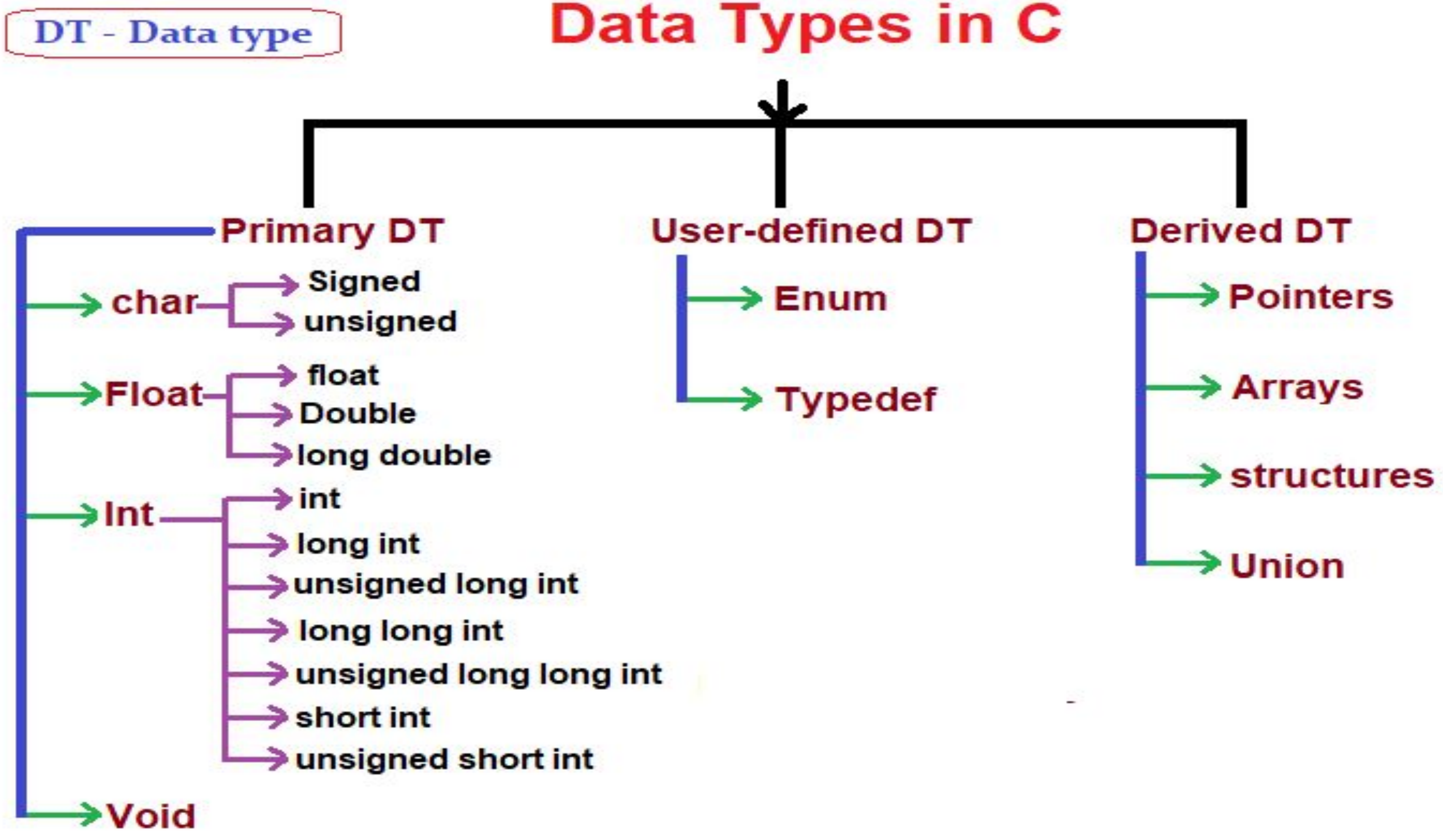
- It make take different values at different times during the execution of the program.
- A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.
- Some examples of such names: average, height, total etc.,

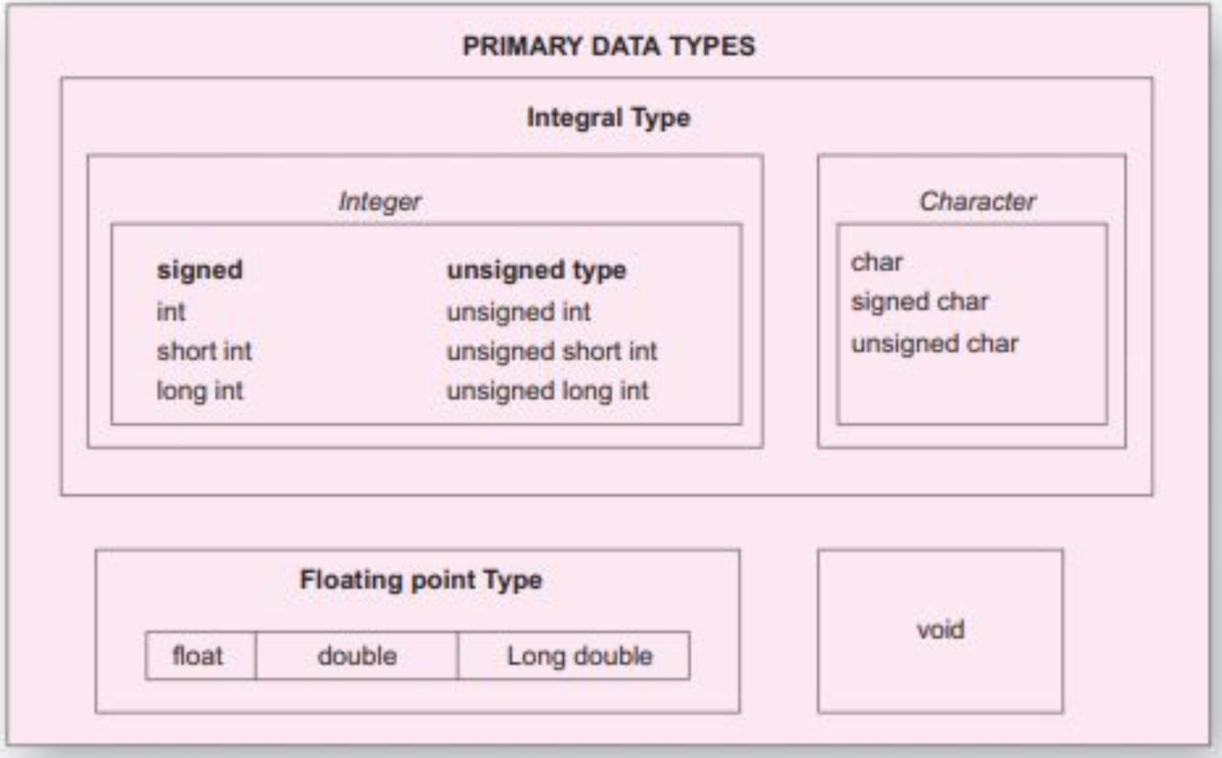
- Variable names may consist of letters, digits, and the underscore(_) character subject to the following conditions:
  - They must begin with a letter or underscore as the first character.
  - ANSI standard recognizes a length of 31 characters. However, length should not be normally more than 8 characters, since only the first 8 characters are treated as significant by many compilers.
  - Uppercase and lowercase are significant.
    - ie,Total is not same as total and TOTAL
  - It should not be a keyword.
  - Whitespace is not allowed.

DATA TYPES:

- C language is rich in its data types.
- From machine to machine, storage representations and machine instructions do vary.
- The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.
- ANSI C supports 3 classes of data types:
  - Primary /Primitive/ Fundamental data types
  - Derived data types
  - User-defined data types

Data Types in C




PRIMARY DATA TYPES
Integral Type
Integer
signed
int
short int
long int
unsigned type
unsigned int
unsigned short int
unsigned long int
Character
char
signed char
unsigned char
Floating point Type
float
double
Long double
void

Primary Data types:

- All C compilers support 5 fundamental data types;
    - Integer(**int**)
    - Character(**char**)
    - Floating point(**float**)
    - Double-precision floating point(**double**)
    - **void**

Void types:

- The void type has **no values.**
- Usually used to specify the type of functions.
- If a function is specified as type void, it do not return any value to the calling function.

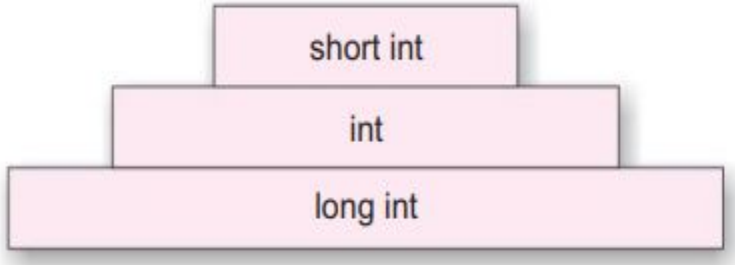
Character types:

- A **single character** can be defined as a character(char) type data.
- Characters are usually stored in **8 bits**(1 byte) of internal storage.
- The qualifier **signed** or **unsigned** may be explicitly applied to char.
- Unsigned chars have values between 0 and 255.
- Signed chars have values from -128 to 127.

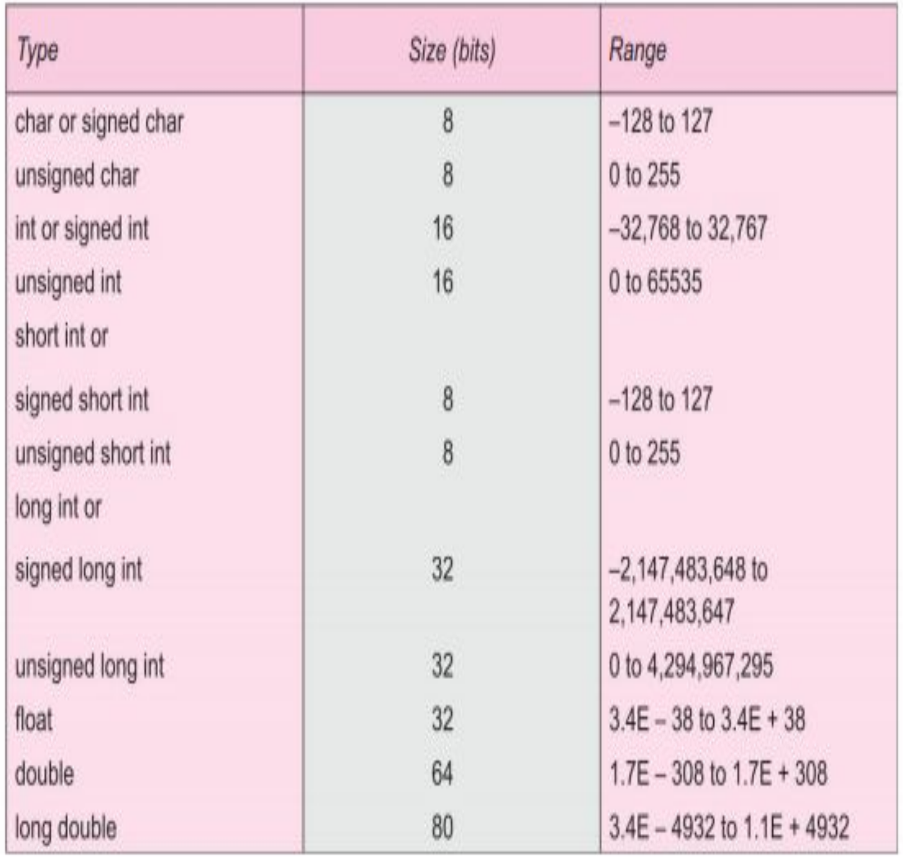
Integer types:

- Used to **hold integer quantities** that **do not contain a decimal point.**
- Generally, integers occupy one word of storage.
- The word size of machines do vary. So, the size of an integer that can be stored depends on the computer.
- C has the following classes of integer storage;
  - int
  - short int
  - long int
  - unsigned int
  - unsigned short int
  - unsigned long int

$$\left.\begin{array}{l}\text{int}\\ \text{short int}\end{array}\right\}\ \text{Signed}$$



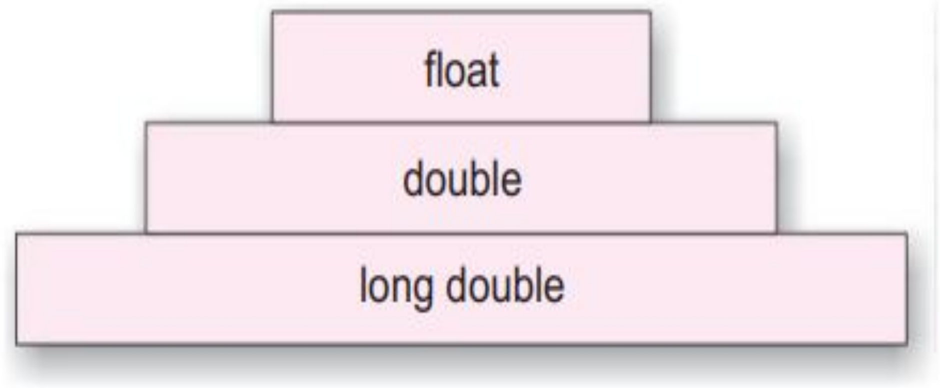
- short int represents fairly small integer values and requires half the amount of storage as a regular int number uses.
- unsigned integers are always positive and all their bits are used for magnitude.



Type	*Size (bits)*	*Range*
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
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

Floating point types:

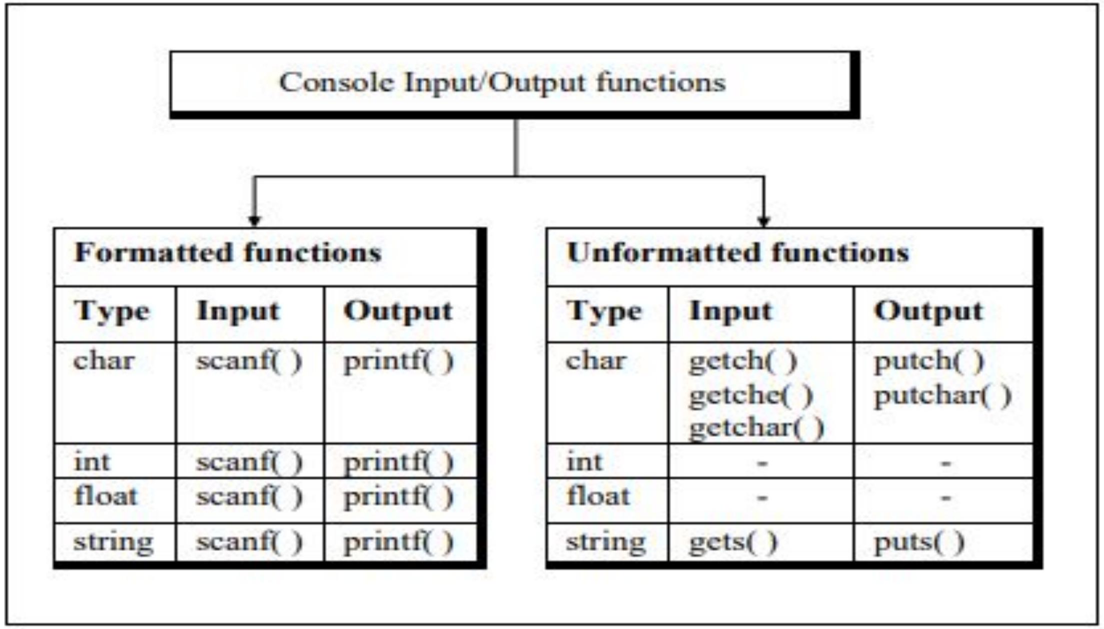
- Floating point (real) numbers are **stored in 32 bits**, with **6 digits of precision.**
- They are defined by the keyword **float.**
- When the accuracy provided by a float number is not sufficient, the type **double** can be used.
- The **double data type** uses **64 bits** giving a precision of **14 digits.**
- The data type double represents the same data type that float represents, but with a greater precision.
- To extend precision further, we may use **long double** which uses **80 bits.**


float
double
long double

Console IO operations:

- These operations allow us to receive input from the input devices like keyboard and provide output to the output devices like the Visual Display Unit.
- The console comprises of the keyboard and the screen.
- IO operations can be classified into
  - Formatted IO functions
  - Unformatted IO functions
- The basic difference between them is that the **formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements.**

- Eg: If values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions.



Console Input/Output functions

Formatted functions

Type	**Input**	**Output**
char	scanf( )	printf( )
int	scanf( )	printf( )
float	scanf( )	printf( )
string	scanf( )	printf( )

Unformatted functions

Type	**Input**	**Output**

char	getch( )
	getche( )
	getchar( )
		putch( )
		putchar( )

int	-	-
float	-	-
string	gets( )	puts( )

- The functions **printf( ), and scanf( )** fall under the category of **formatted console I/O functions**.
- These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form.

scanf:

- scanf( ) allows us to enter data from keyboard that will be formatted in a certain way.
- The general form of scanf( ) statement is as follows:
    - **scanf ("format string", list of addresses of variables)** ;
- For example: scanf ("%d %f %c", &c, &a, &ch) ;

- Note that we are sending addresses of variables (addresses are obtained by using ‘&’ the ‘address of’ operator) to scanf( ) function.
- This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses.
- The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s).
- Do not include these escape sequences in the format string.

printf:

- The general form of printf is as follows:

  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

Format Specifiers:

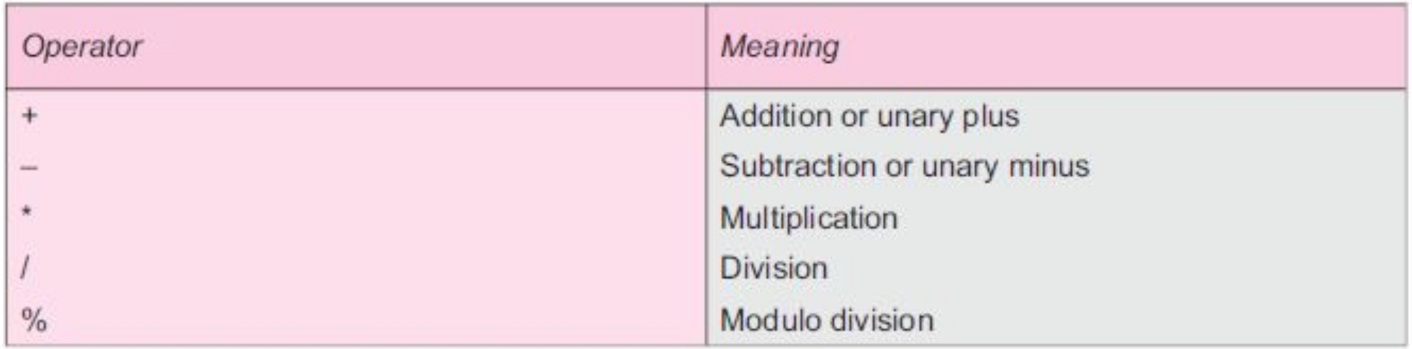
- The format specifiers are used in C for input and output purposes.
- Using this concept, the compiler can understand that what type of data is in a variable during taking input using the scanf() function and printing using printf() function.
- %d - Signed integer
- %f - Float value
- %c - Character
- %s - String

Operators & Expressions:

- C supports a rich set of built-in operators.
- An operator is a symbol that **tells the computer to perform certain mathematical and logical manipulations.**
- Operators are used in programs to **manipulate data and variables**.
- They usually form a part of mathematical or logical expressions.
- **C operators** can be classified into following categories:
    - **Arithmetic operators.**
    - **Relational operators.**
    - **Logical operators.**
    - **Assignment operators**
    - **Increment & Decrement operators**
    - **Conditional operators**
    - **Bitwise operators**
    - **Special operators**

Arithmetic Operators:

- C provides all arithmetic operators.
- Integer division truncates any fractional part.
- The modulo division operation produces the remainder of an integer division.
- The modulo division operator cannot be used on floating point data.
-



Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

- Examples of use of arithmetic operators are:
    - a-b
    - a+b
    - a*b
    - a/b
    - a%b
    - -a%b
- Here, a and b are variables and are known as **operands**.
- 3 categories of arithmetic operations:
    - **Integer Arithmetic.**
    - **Real Arithmetic**
    - **Mixed-mode arithmetic.**

<u>Integer Arithmetic:</u>

- When both **operands in an arithmetic expression are integers**, that expression is called integer expression and the operation is called integer arithmetic.
- Integer arithmetic always yields an integer value.

- Eg: If a and b are integers and a=14 and b=4, then;
  - a-b=10
  - a+b=14
  - a*b=56
  - a/b=3 (decimal part truncated)
  - a%b=2 (remainder of division)

Real Arithmetic:

- An arithmetic operation involving **only real operands**.
- A real operand may assume values either in decimal or exponential notation.
- Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.
- The operator % can’t be used with real operands.

Mixed-mode arithmetic:

- When **one of the operands is real and the other is integer**, the expression is called a mixed-mode arithmetic.
- If either operand is of the real type, then only the real operation is performed and the result is always a real number.
- Thus, 15/10.0 = 1.5

whereas

15/10 = 1

<u>Relational Operators:</u>

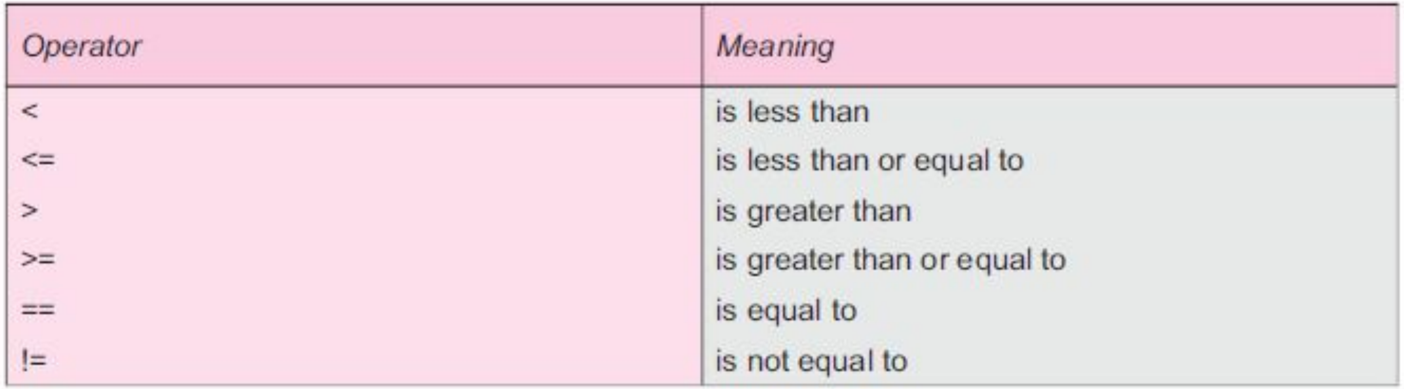
- We may come across situations where we have to **compare two quantities** and depending on their relation, we may have to take a decision.
- These comparisons can be done using relational operators.
- Eg: Comparing age of two persons, compare price of two items etc.,
- Expression containing a relational operator is called a **relational expression.**
- The **value of relational expression is either one or zero.**
- If the specified relation is true, the value is is 1.
- If the specified expression is false, the value is 0.
- Eg:

$10 < 20$ is true

And

$20 < 10$ is false

- C supports 6 relational operators:



Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

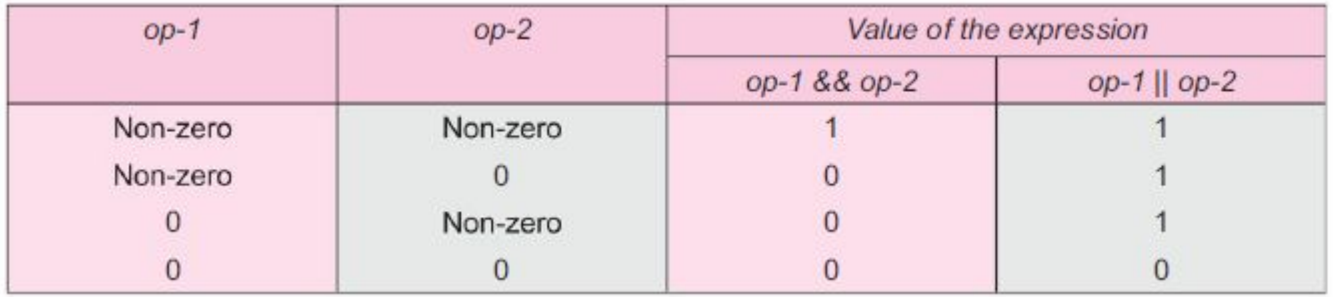
- A simple relational expression contains only one relational operator and takes the following form;
  ae-1 relational operator ae-2
  where ***ae-1*** and ***ae-2*** are **arithmetic expressions** which may be **simple constants, variables or combinations of them**.

- Examples of some relational expressions:
  - 4.5 <= 10 TRUE
  - 4.5 < -10 FALSE
  - 10 < 7+5 TRUE
  - -35 >= 0 FALSE
  - a+b = c+d TRUE only if sum of values of a and b is equal to sum of values of c and d.
- When arithmetic expressions are used in either side of relational operators, arithmetic expression will be evaluated first and then results are compared.
- **Arithmetic operators have precedence over relational operators.**
- **Among the relational operators, each one is a complement of another operator.**
  - >   is complement of   <=
  - <   is complement of   >=
  - ==   is complement of   !=

Logical Operators:

- C has the following three logical operators;
  - **&&** meaning logical AND
  - **||** meaning logical OR
  - **!** meaning logical NOT
- When we want to test more than one conditions to make a decision, we can use the logical operators && and ||.
- Eg: a>b && x==10
- Above logical expression includes a relational expression (a>b) and another relational expression (x==10) connected through a logical operator (&&).
- The expression above is true only if a>b is true and x==10 is true.
- An expression of this kind, which combines two or more relational expressions is termed as **logical expression** or a compound relational expression.

- A logical expression **yields a value one or zero** according to the following truth table;



op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

- Some examples for the usage of logical expressions are;
  - Eg1: if(age>30 && salary<10000)
  - Eg2: if(number<0 || number>100)

Assignment Operators:

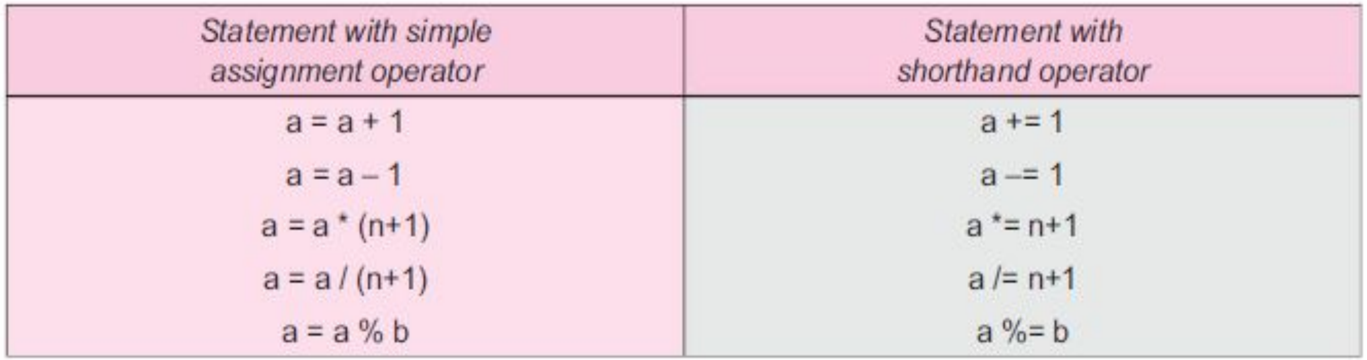
- Used **to assign the result of an expression to a variable.**
- ‘=’ is the usual assignment operator used.
- In addition, C has a set of ‘shorthand’ assignment operators of the form;

v op=exp;

where v is a variable, exp is an expression and op is a C binary arithmetic operator.

- The operator **op=** is known as the shorthand arithmetic operator.
- The statement **v op=exp;** is equivalent to **v = v op (exp);** with v evaluated only once.

- Shorthand assignment operators:



Statement with simple assignment operator	*Statement with shorthand operator*
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

- Advantages of using shorthand assignment operators;
  - What appears on the left hand side need not be repeated and therefore it becomes easier to write.
  - The statement is more concise and easier to read.
  - The statement is more efficient.

- Eg: Consider the statement below;

```
x += y+1;
```

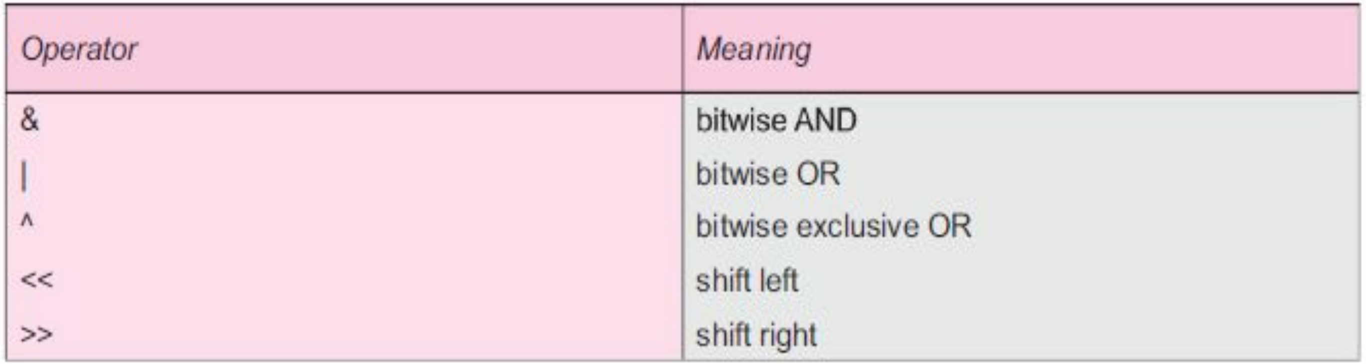
- This is same as x = x+ (y+1);
- The shorthand operator += means;
    - ‘add y+1’ to ‘x’

      or

    - ‘increment x by y+1’

Bitwise Operator:

- Used for **manipulation of data at bit level.**
- Used for testing bits, or shifting them right or left.
- Bitwise operators may not be applied to float or double.
- Following table lists the bitwise operators and their meanings;



Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

Conditional Operator:

- A ternary operator pair “?:” is available in C to construct conditional expressions of the form

  exp1 ? exp2 : exp3

  where *exp1*,*exp2*,*exp3* are expressions.

- The operator **? :** works as follows:
  - *exp1* is evaluated first.
  - If it is nonzero(true), then the expression *exp2* is evaluated and becomes the value of the expression.
  - If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression.

- **Eg**: Consider the following statements;

```
a = 10;

b = 15;

x = (a>b) ? a : b;
```

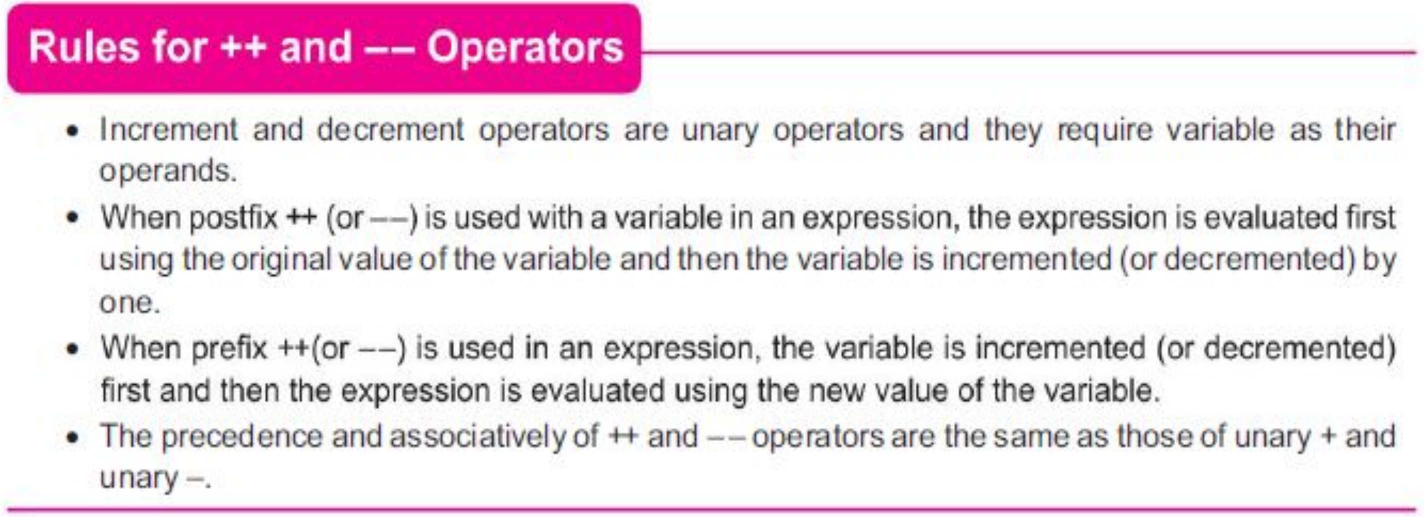
  - In this example, x will be assigned the value of b.
  - This can be achieved by using the if-else statement as follows;

```
if(a>b)

    x=a;

else

    x=b;
```

Increment & Decrement Operators:

- C allows two useful operators which are rarely found in other languages. They are;
    - Increment operator **(++)**
    - Decrement operator **(--)**
- The operator ++ adds 1 to the operand while -- subtracts 1.
- Both are unary operators and takes the following form;
    - ++m; or   m++;
    - --m;   or m--;
- ++m; is equivalent to m = m+1;
- --m; is equivalent to m= m-1;
- Both m++ and ++m mean the same thing. But they behave differently when they are used in expressions on the RHS of an assignment statement.

- Eg1: Consider m=5; and y=++m;
    - Here, the value of m and y would be 6.
- Eg2: Consider m=5; and y=m++;
    - Here, the value of y would be 5 and m would be 6.
- ie, A **prefix operator** first adds 1 to the operand and then the result is assigned to the variable on left whereas a **postfix operator** first assigns the value to the variable on the left and then increments the operand.



Rules for ++ and –– Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or ––) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++(or ––) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and –– operators are the same as those of unary + and unary –.

Special Operator:

- C supports some special operators such as;
  - comma operator
  - sizeof operator
  - pointer operators (* and &)
  - member selection operators (. and ->)

The comma operator:

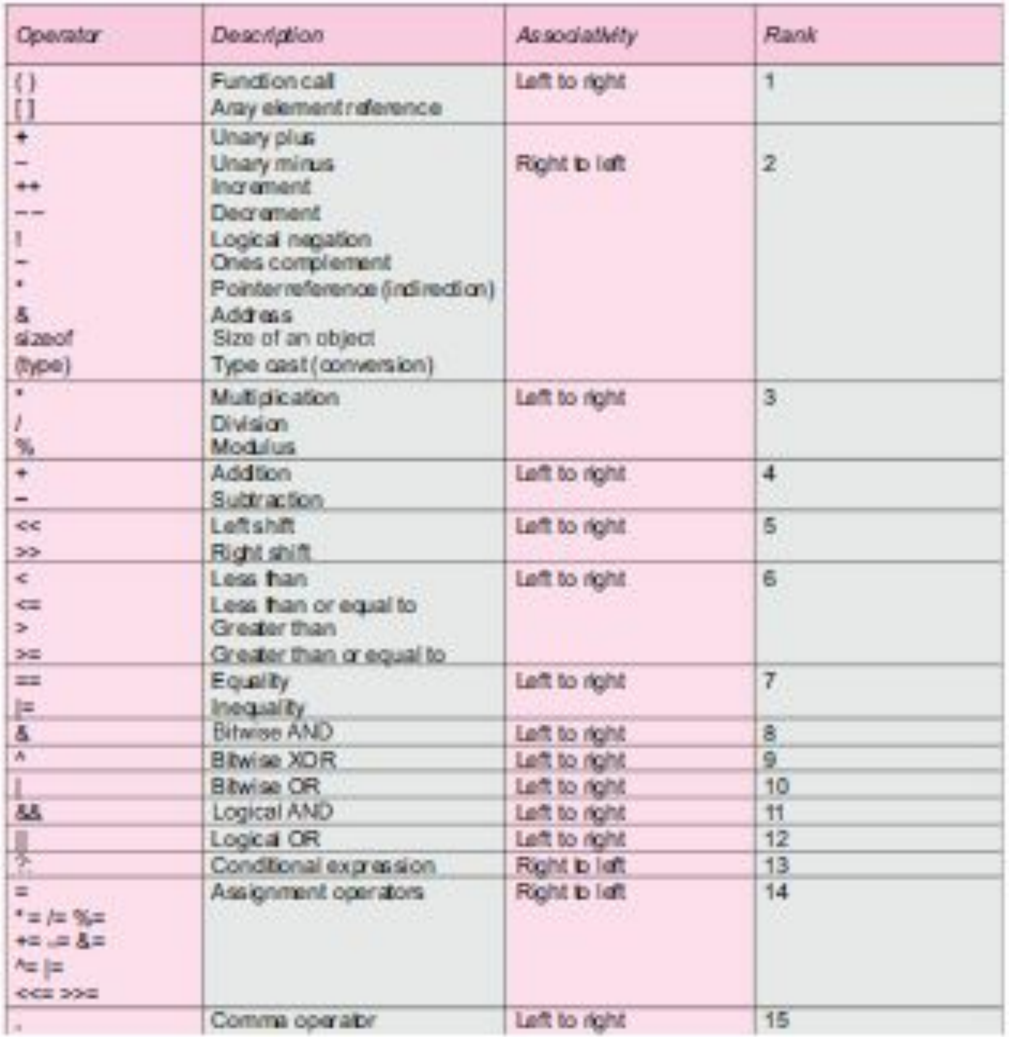
- Used to link the related expressions together.
- A comma linked list of expressions are evaluated left to right and the value of rightmost expression is the value of combined expression.
- Eg: Consider the expression, value = (x = 10, y = 5, x+y);
- The value 10 is assigned to **x**, then value 5 is assigned to **y** and finally assigns 15 to **value**.
- Parentheses is compulsory here since the comma operator has lowest precedence.

The sizeof operator:

- When the sizeof operator is used with an operand, it **returns the number of bytes the operands occupy.**
- The operand may be a variable, a constant or a data type qualifier.
- Eg:
    - m = sizeof(sum);
    - n = sizeof(longint);
- Usually sizeof operator is used to find the length of arrays and structures when their sizes are not known to the programmer.
- It is also used to allocate memory space dynamically to variables during program execution.

Operator precedence:

- Each operator in C has precedence associated with it.
- This precedence is used to determine how an expression involving more than one operator is evaluated.
- There are distinct levels of precedence and an operator may be belong to one of these levels.
- The operators at higher level of precedence are evaluated first.
- The operators at same level of precedence are evaluated from left to right or right to left depending on the level.
- This is known as the associativity property of an operator.



Operator	Description	Associativity	Rank
( ) [ ]	Function call Array element reference	Left to right	1
+ – ++ – – ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ –	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== !=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= *= /= %= += –= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

CONTROL FLOW STATEMENTS:

- C program is a set of statements which are normally executed sequentially in the order in which they appear.
- But, in practice, we may come across various situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met.
- This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.
- Such statements are called **decision making statements**.
- Since **these statements ‘control’ the flow of execution** , they are also known as **control statements**.

- C involves the following **control flow statements**:(or decision making statements:
    - **if statement**
    - **switch statement**

  Conditional statements

    - **while statement**
    - **for statement**
    - **do-while statement**

  Loop control statements

    - **break statement**
    - **continue statement**
    - **goto statement**

  Jump statements

if statement:

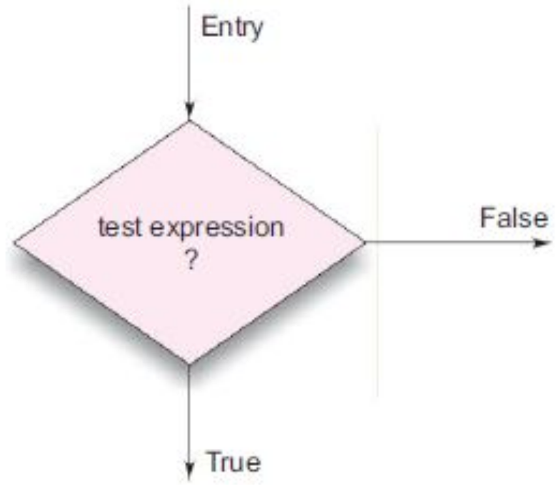
- In a program, at many times, a set of statements has to be executed in one situation and an entirely different set of statements to be executed in another situation.
- In such cases, if-statement is a very powerful statement.
- It is a **two-way decision statement.**
- It is **used in conjunction with an expression.**
- It takes the following form:

if (test expression)

- Here, the computer will evaluate the expression first and then depending on whether the value of the expression is true or false, it transfers the control to a particular statement.
- Eg:

```
if (room is dark)

    put on lights
```

- This point of program has two paths to follow, one for the true condition and other for the false condition as below;



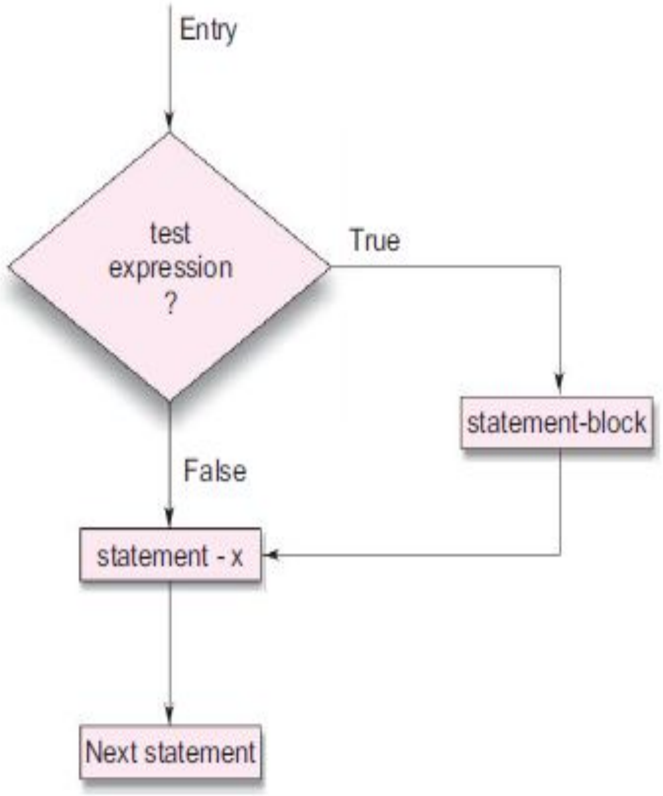
- Depending on the complexity of conditions to be tested, if statement can take any of the following forms:
    - **Simple if statement.**
    - **if.....else statement.**
    - **Nested if....else statement.**
    - **else if ladder**

Simple if statement:

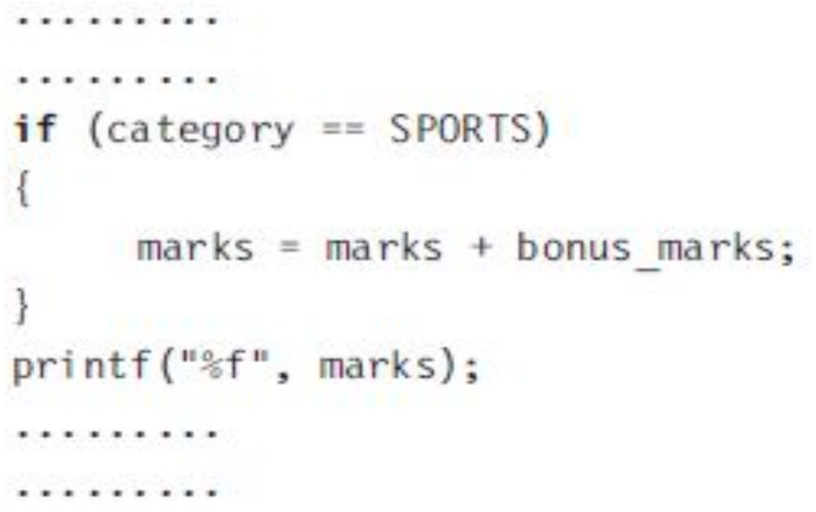
- The general form of a simple if statement is;

```
if (test expression)
  {
     statement-block;
  }
     statement-x;
```

- If the test expression is true, statement-block is executed; otherwise statement block will be skipped and the execution will jump to the statement-x.
- When the condition is true, both the statement-block and statement-x will be executed.



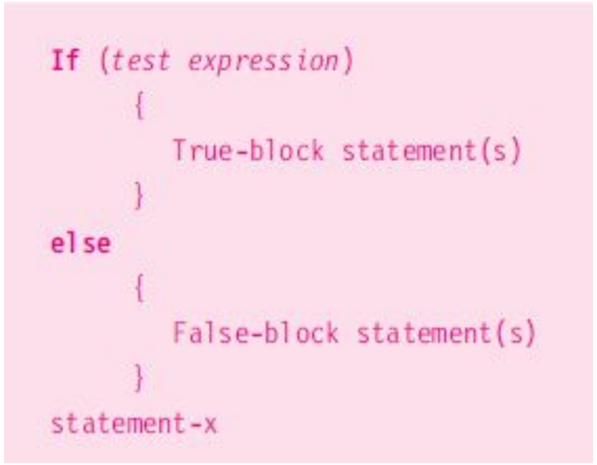
- Eg: Consider the following program segment;
  - The program tests the type of category of the student.
  - If the student belongs to the SPORTS category, then additional bonus marks are added to marks before they are printed.
  - For others bonus marks are not added.



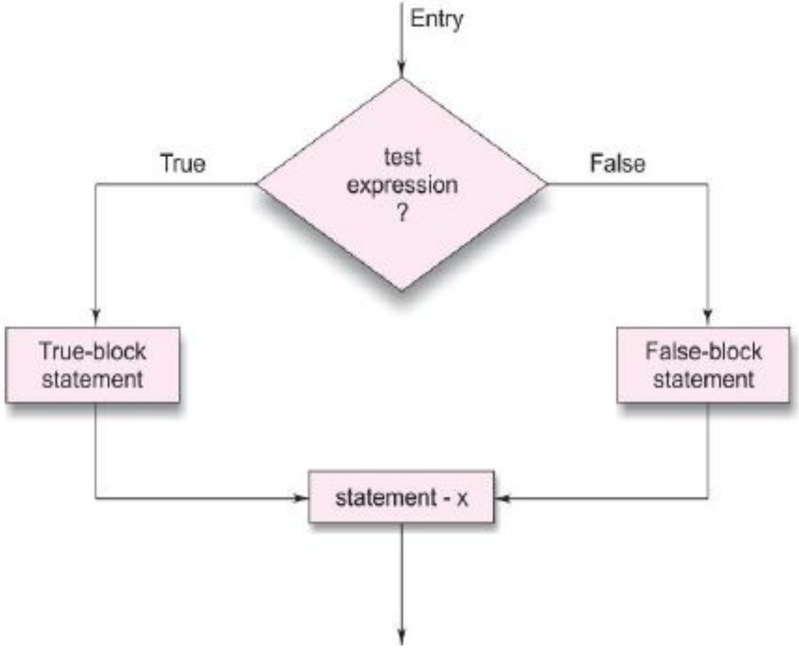
```
..........
..........
if (category == SPORTS)
{
      marks = marks + bonus_marks;
}
printf("%f", marks);
..........
..........
```

The if....else statement:

- The general form of if....else statement is;

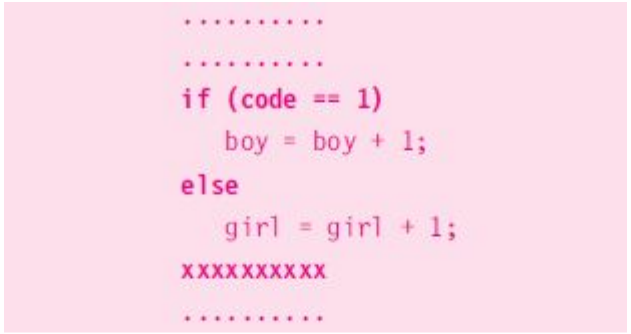


```
If (test expression)
      {
          True-block statement(s)
      }
else
      {
          False-block statement(s)
      }
statement-x
```



- If the test expression is true, then the true block statement(s) , immediately following the if statements are executed.
- Otherwise, the false block statement(s) are executed.
- In either case, either true-block or false-block will be executed, not both.
- In both the cases, the control is transferred subsequently to the statement-x.

- <u>Eg:</u> Consider the following program segment;

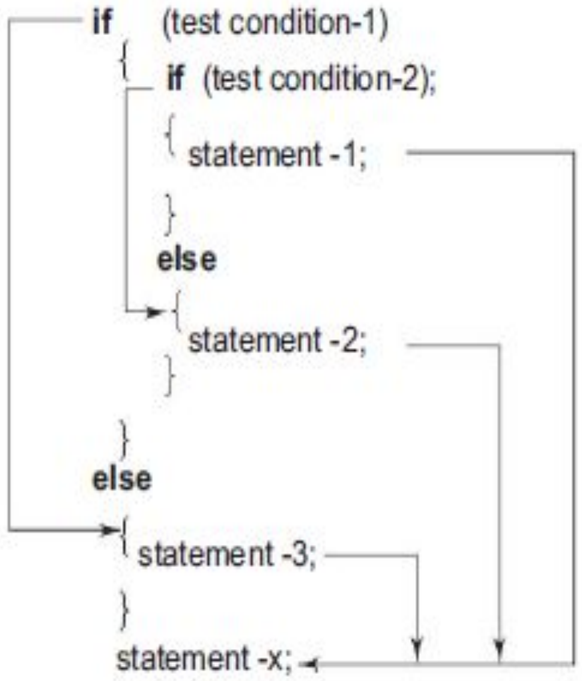


```
..........
..........
if (code == 1)
   boy = boy + 1;
else
   girl = girl + 1;
xxxxxxxxxx
..........
```

  - Here, if the code is equal to 1, statement boy = boy + 1; is executed; and the control is transferred to the statement xxxxxxxxx; , after skipping the else part.
  - If the code is not equal to 1, the statement girl = girl + 1; is executed before the control reaches the statement xxxxxxxxx;

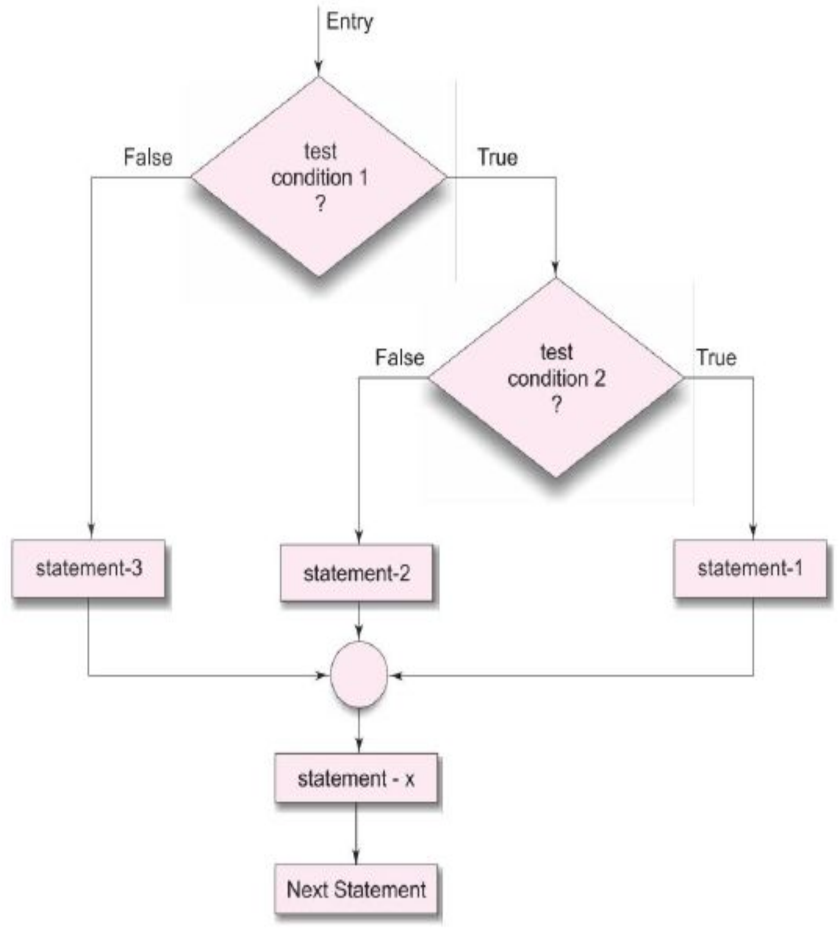
Nested if....else statement:

- When a series of decisions are involved, we may have to use more than if...else statement in nested form as shown below:



```
if    (test condition-1)
  {
  if  (test condition-2);
    {
     statement -1;
    }
   else
    {
     statement -2;
    }
  }
else
  {
   statement -3;
  }
 statement -x;
```

- If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test.
- If the condition-2 is true, the statement-1 will be executed; otherwise statement-2 will be evaluated and then the control is transferred to statement-x.

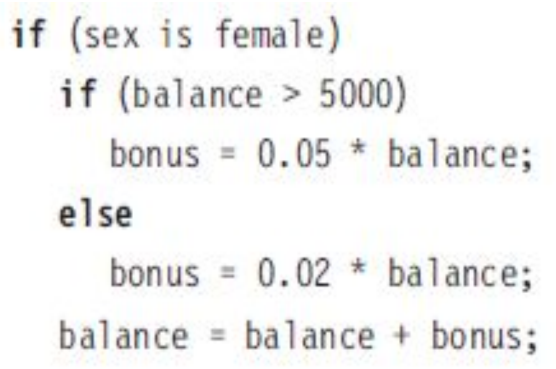


- Figure shows the flowchart of nested if...else statements.
- Eg: Consider the following code segment;

```
.........
   if (sex is female)
{
   if (balance > 5000)
      bonus = 0.05 * balance;
   else
      bonus = 0.02 * balance;
}
else
{
      bonus = 0.02 * balance;
}
balance = balance + bonus;
.........
.........
```

Dangling Else Problem:

- One of the classic problems encountered while using nested if....else statements.
- Occurs when a **matching else is not available for an if**.
- In such cases, always match an else to the most recent unmatched if in the current block.
- ie, ***“else is always paired with the most recent unpaired if”***.
- Eg:

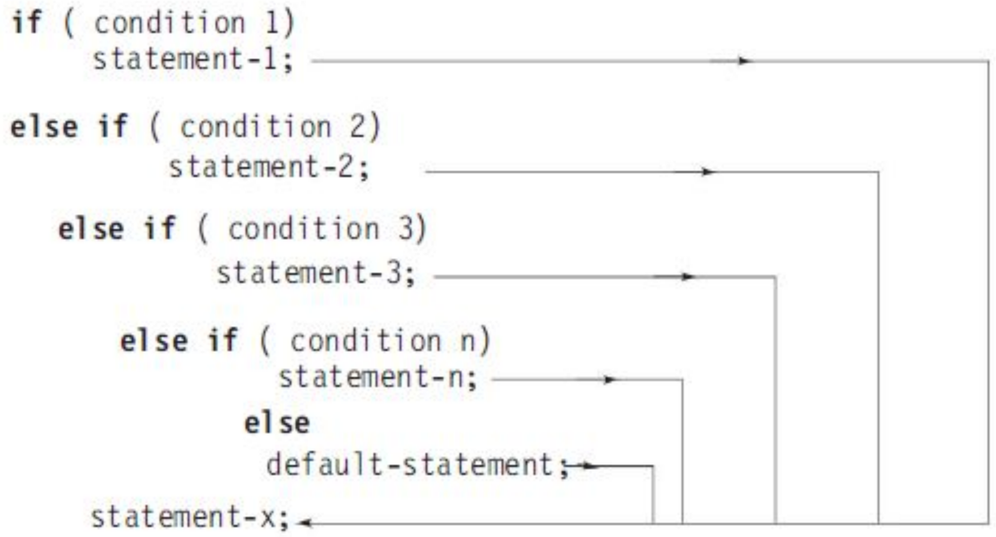


```
if (sex is female)
   if (balance > 5000)
      bonus = 0.05 * balance;
   else
      bonus = 0.02 * balance;
   balance = balance + bonus;
```

- Here, there is an ambiguity as to over which **if** the **else** belongs to.
- The **else** will be associated to the inner **if** and there is no **else** option for the outer if.
- In this example, the computer will execute the statement balance = balance + bonus; without calculating the bonus of the male account holders.

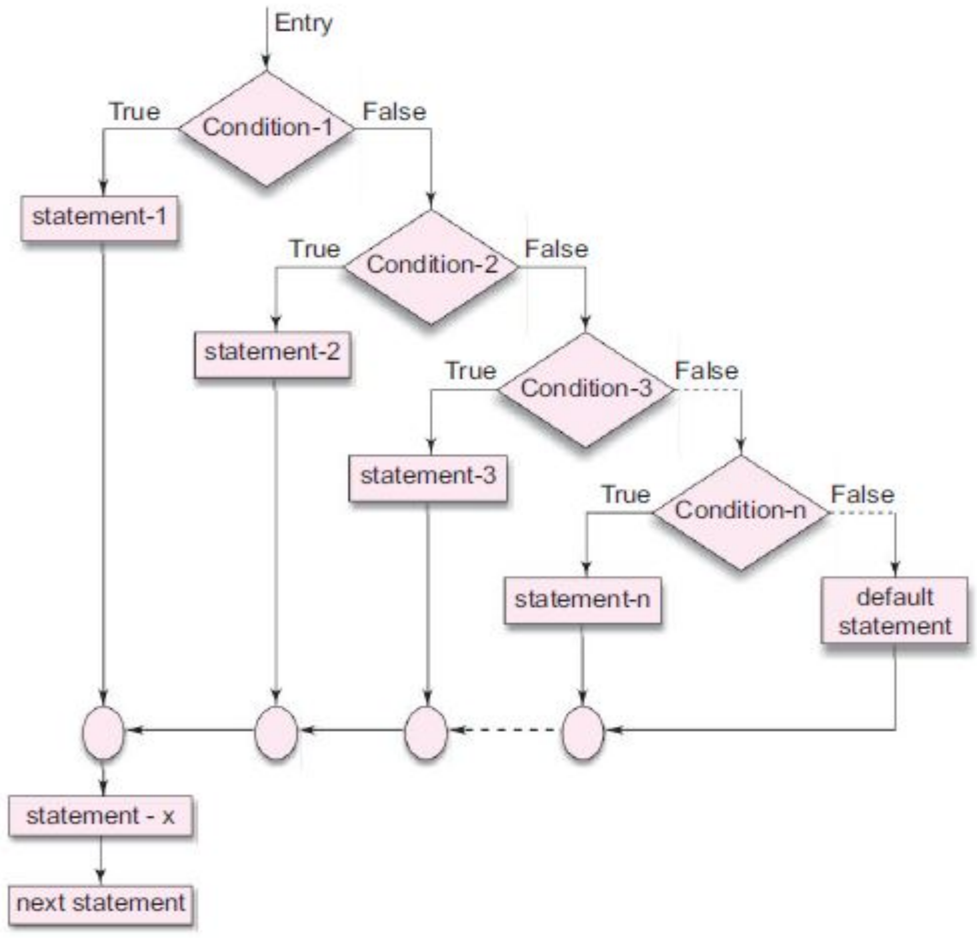
The else if ladder:

- When multipath decisions are involved, **ifs** can be put together.
- A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**.
- The general form is as follows;



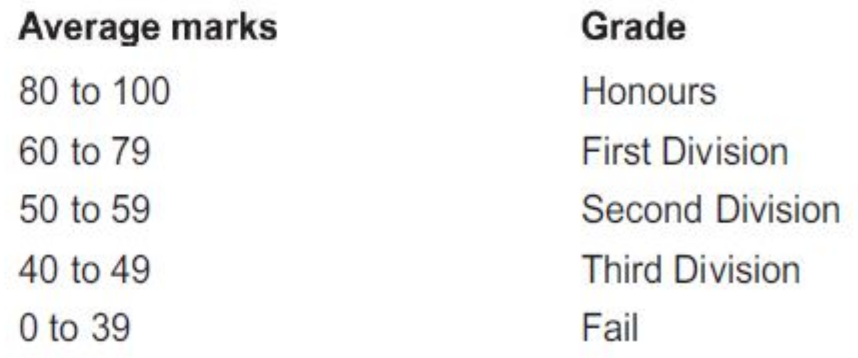
```
if ( condition 1)
     statement-1;

else if ( condition 2)
          statement-2;

    else if ( condition 3)
              statement-3;

         else if ( condition n)
                   statement-n;
                   else
                     default-statement;
     statement-x;
```

- This construct is known as the **else if** ladder.
- The conditions are evaluated from the top(of the ladder) downwards.
- As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x.
- When all the n conditions become false, the final **else** containing the default statement will be executed.



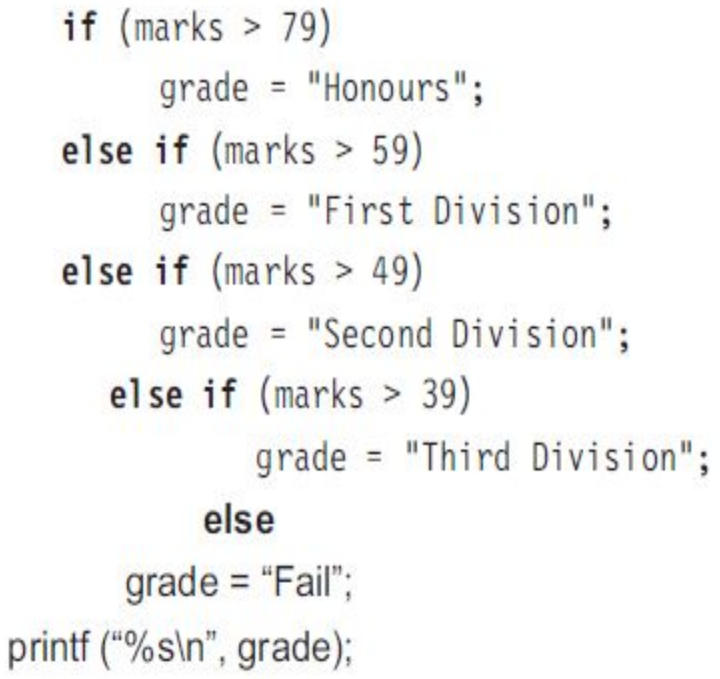
- The figure shows the logic of execution of **else if** ladder statements.

- Eg: Consider the example of grading the students in an academic institution.
- The grading is done as per the following rules;
-



Average marks	Grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

- This can be done using else if ladder as shown here:

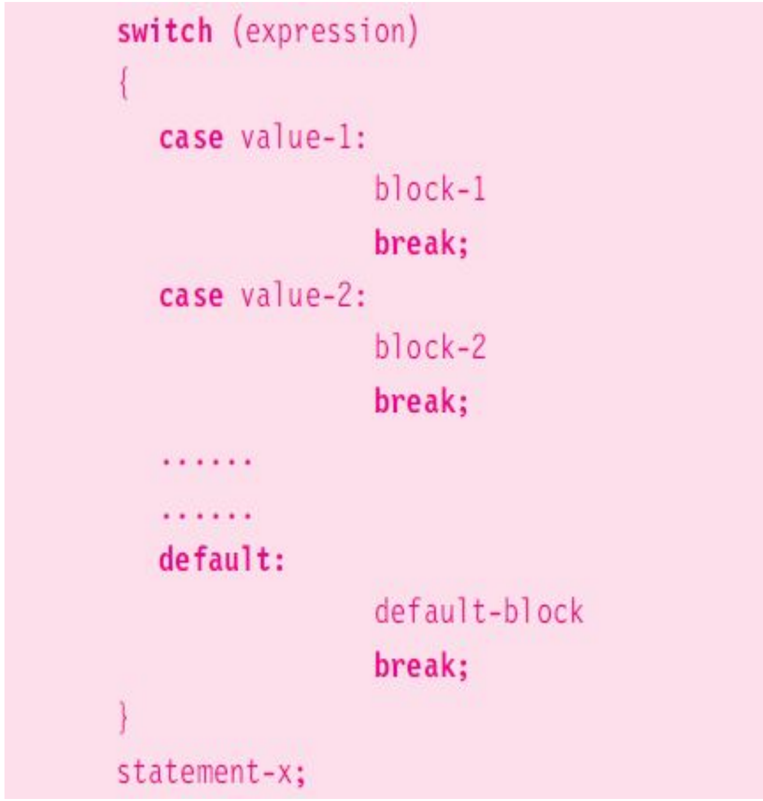


```
if (marks > 79)
    grade = "Honours";
else if (marks > 59)
    grade = "First Division";
else if (marks > 49)
    grade = "Second Division";
  else if (marks > 39)
        grade = "Third Division";
      else
    grade = "Fail";
printf ("%s\n", grade);
```

The switch statement:

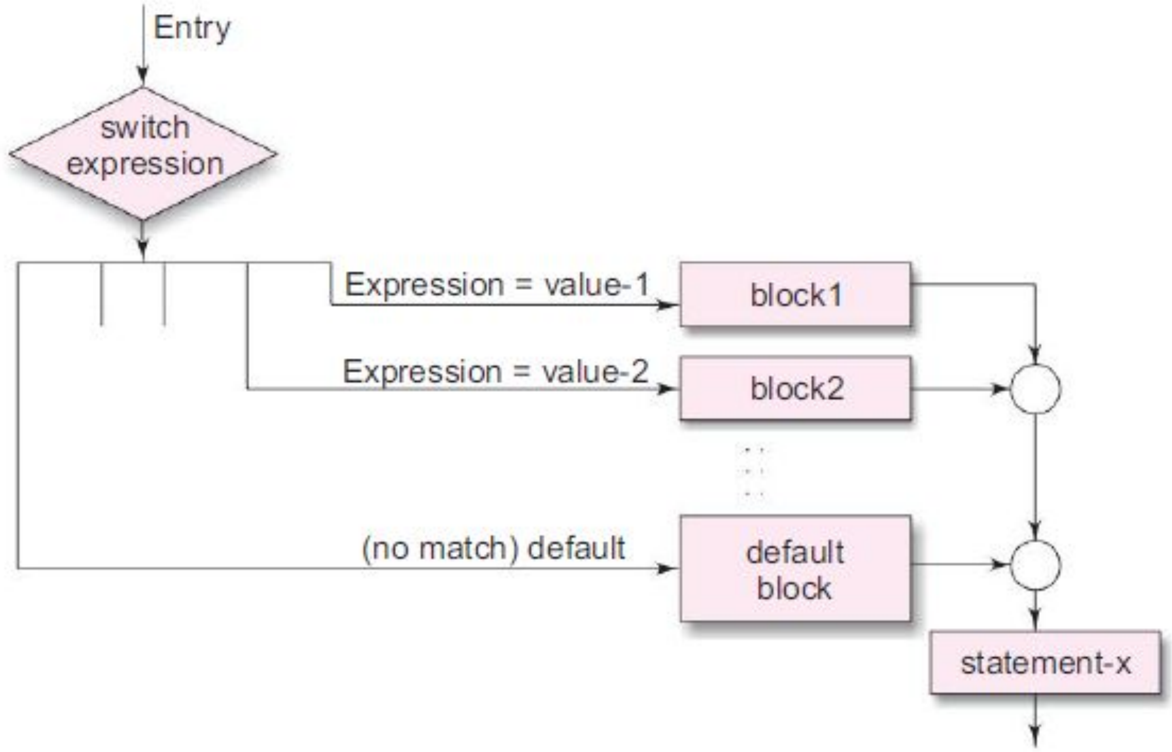
- When one of the many alternatives is to be selected, we can use an if statement to control the selection.
- But, when the number of alternatives increases, the complexity of such a program dramatically increases.
- ie, program becomes difficult to read and follow and it may even confuse the designer of the program.
- For tackling this, C has an in-built **multiway decision statement** known as **switch**.
- **The switch statement tests the value of a given variable(or expression) against a list of *case* values and when a match is found, a block of statements associated with that case is executed.**

- The general form of a switch statement is as shown:
- The ***expression*** is an integer expression or characters.
- Value-1, value-2,... are constants or constant expressions (evaluable to an integral constant) and are known as ***case labels***.
- Each of these values should be unique within a switch statement.
- Block-1,block-2,... are statement lists and may contain zero or more statements.
- Case labels end with a colon(:).

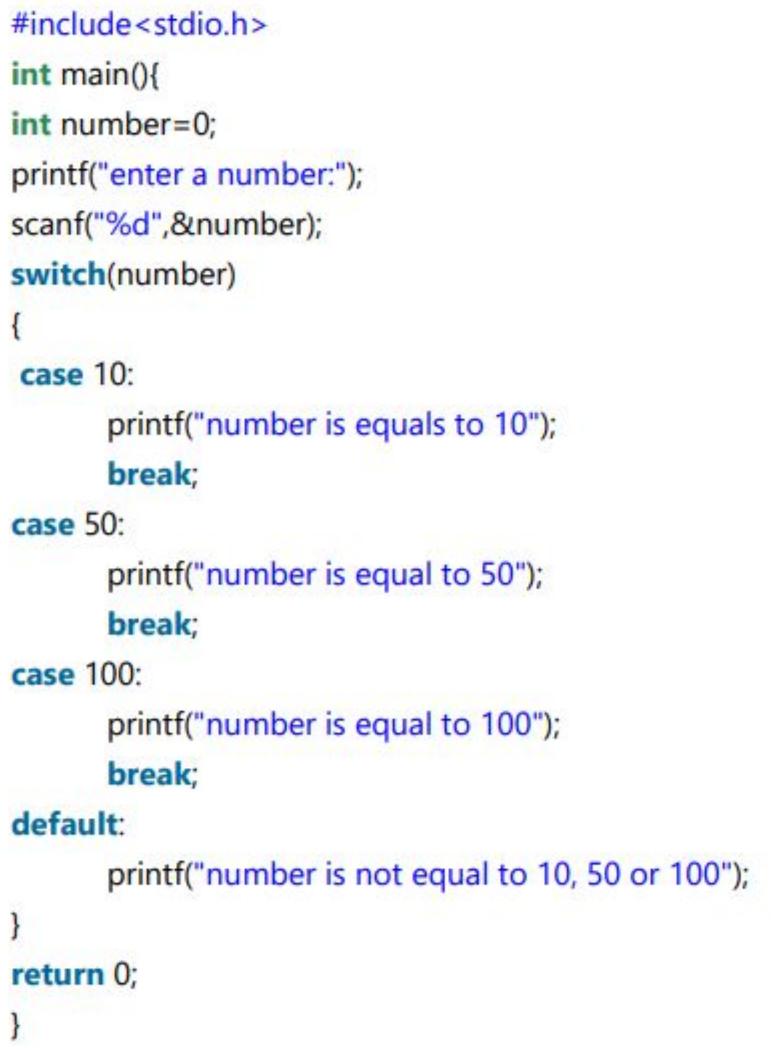


```
switch (expression)
{
   case value-1:
                    block-1
                    break;
   case value-2:
                    block-2
                    break;
   ......
   ......
   default:
                    default-block
                    break;
}
statement-x;
```

- Selection process of a switch statement:



- When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2, ...
- If a case is found whose value matches with the value of the expression,then the block of statements that follows the case are executed.
- The ***break statement*** at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the ***statement-x*** following the switch.
- The ***default*** is an optional case.
    - When present, it will be executed if the value of the expression does not match with any of the case values.
    - If not present, no action takes place if all matches fail and the control goes to the statement-x.

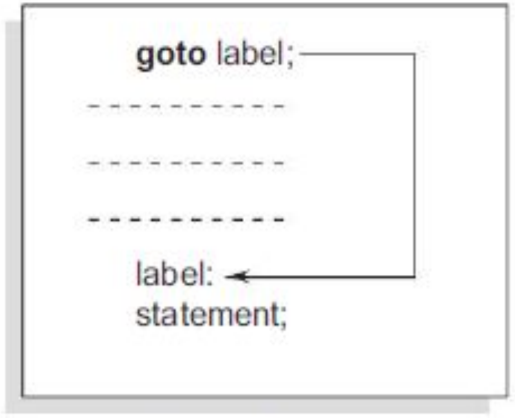


```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number)
{
 case 10:
          printf("number is equals to 10");
          break;
case 50:
          printf("number is equal to 50");
          break;
case 100:
          printf("number is equal to 100");
          break;
default:
          printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

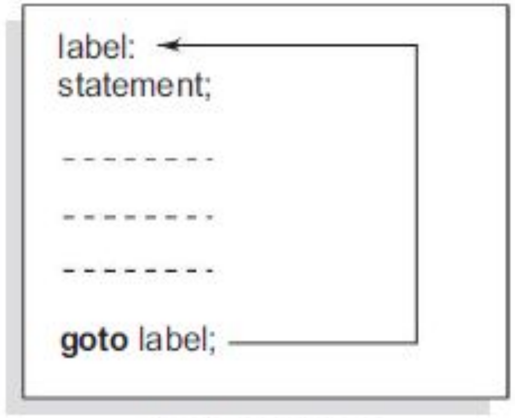
- Eg: Showing a simple example of C language switch statement.

The goto statement:

- ***goto*** statement allows to **branch unconditionally from one point to another in the program.**
- The ***goto*** requires a **label** to identify the place where the branch is to be made.
- **A label can be any valid variable name, and must be followed by a colon.**
- A goto breaks the normal sequential execution of the program.
- Following are the general forms of goto and label statements:



Forward jump



Backward jump

- The label: can be anywhere in the program either before or after the goto label; statement.
- Eg:

```
if (x<10)
        goto error;
        ............
        error: printf ("Error in input");
```

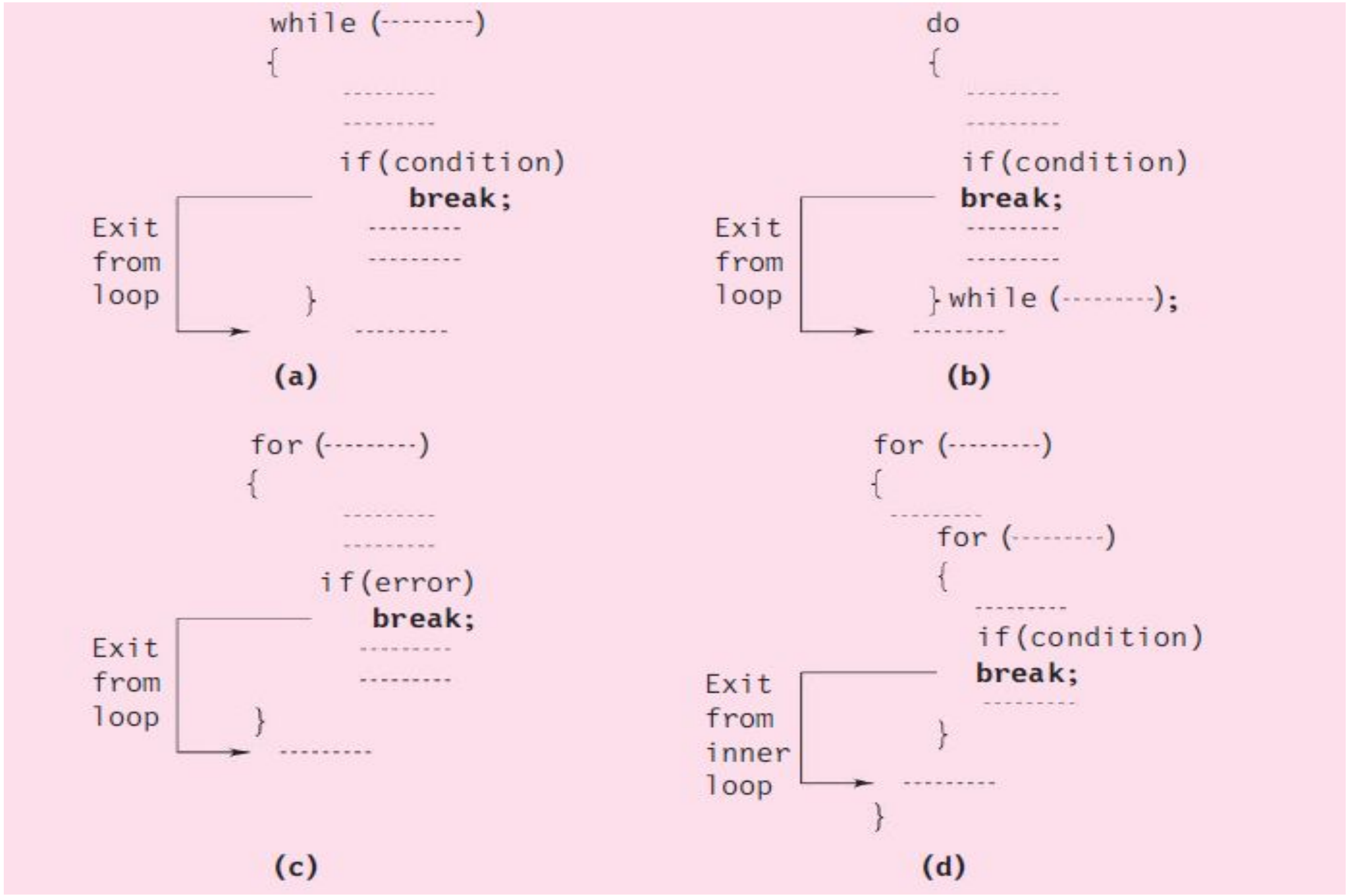
- When the goto statement is encountered, the control is transferred to the labelled statement.
- error is the label here.
- Then, subsequent statements are executed.

- **Forward jump:** If the **label: is placed after the goto label;** some statements will be skipped and the jump is known as forward jump.
- **Backward jump:** If the **label: is placed before the goto label;** a loop will be formed and statements will be executed repeatedly. Such a jump is known as backward jump.
- A goto is often used at the end of a program to direct the control to go to the input statement, to read further data.

- Another use of goto statements is to transfer the control out of a loop when certain conditions are encountered.
- The use of goto statements may make many confusions and complications in the program rather than making a clarity.
- So, it is highly recommended to avoid the usage of goto statements if possible.

The break statement:

- The *break* statement **allows to jump out of a loop**.
- **When a *break* statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.**
- **When the loops are nested, the *break* would exit from the loop containing it.**
    - ie, break would exit only a single loop.



```
while (--------)
{
      --------
      --------
     if(condition)
         break;
       --------
       --------
}
      --------
```

(a)

```
do
{
      --------
      --------
     if(condition)
     break;
      --------
      --------
}while(--------);
--------
```

(b)

```
for (--------)
{
      --------
      --------
     if(error)
        break;
       --------
       --------
}
  --------
```

(c)

```
for (--------)
{
  --------
    for (--------)
    {
      --------
      if(condition)
      break;
      --------
    }
  --------
}
```

(d)

- Illustrating how break statement is used within *while*, *for* and *do while* loops.

The continue statement:

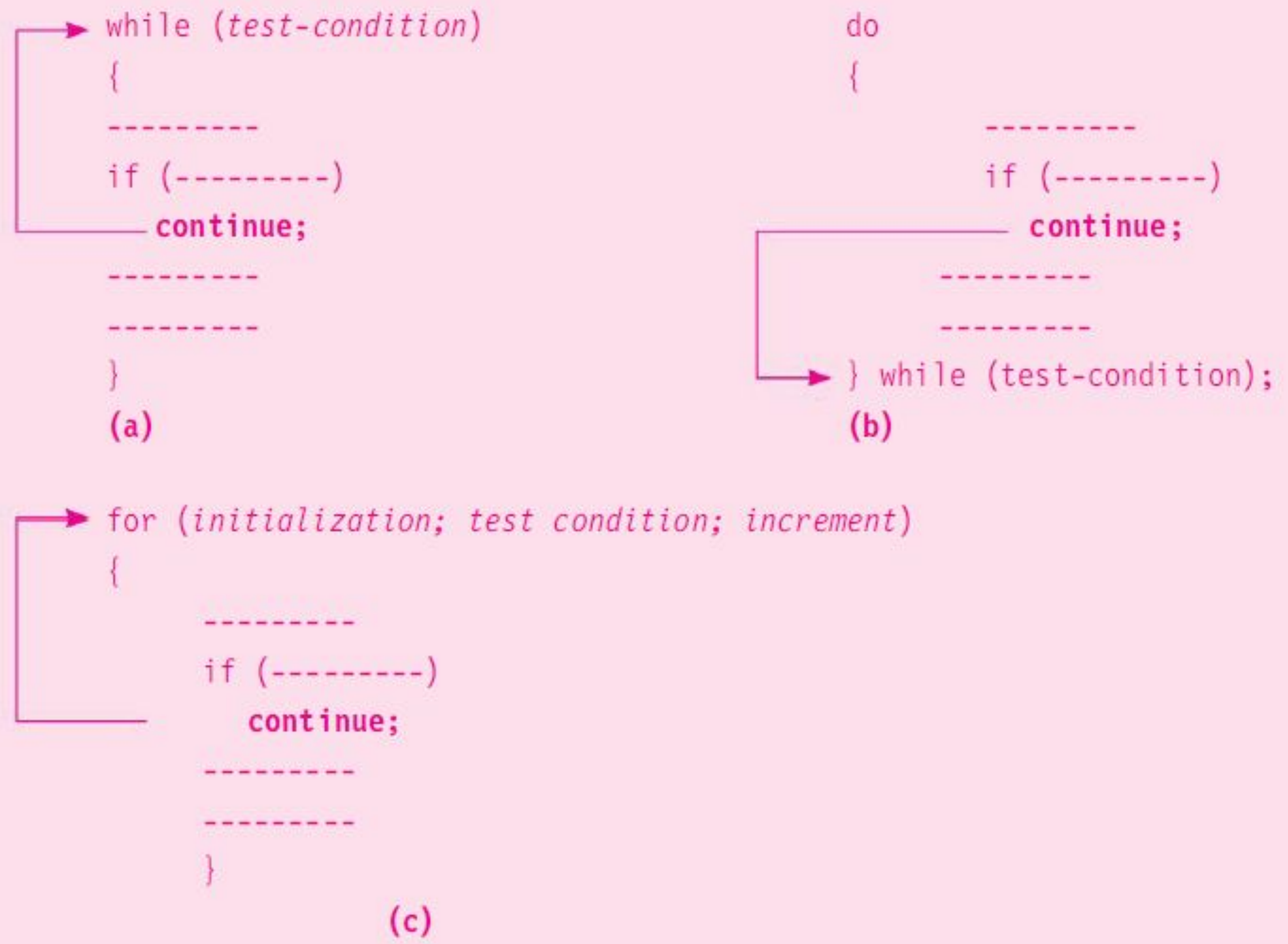
- The ***continue*** statement causes the loop to be continued and after skipping any statements in between.
- ie, the ***continue*** statement tells the compiler;

  "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION"

- The format of continue statement is simply

  continue;

- In *while* and *do* loops, continue causes the control to go directly to the test condition and then to continue the iteration process.
- In the *for* loop, the increment section of loop is executed before the test condition is evaluated.



```
while (test-condition)
{
---------
if (---------)
  continue;
---------
---------
}
```

(a)

```
do
{
        ---------
        if (---------)
          continue;
      ---------
      ---------
} while (test-condition);
```

(b)

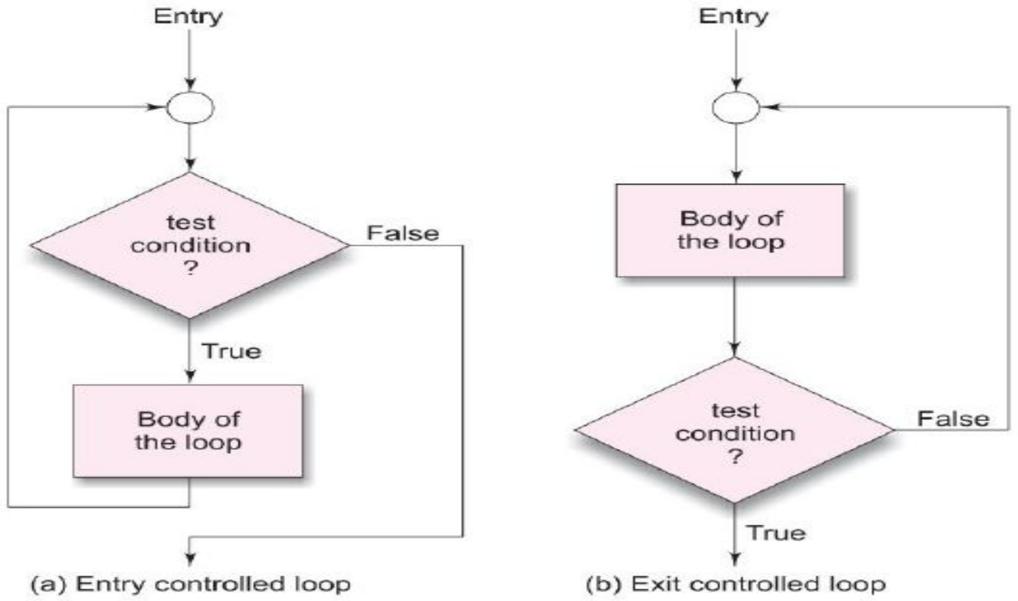
```
for (initialization; test condition; increment)
{
      ---------
      if (---------)
        continue;
      ---------
      ---------
      }
```

(c)

Looping statements in C:

- In looping, **a sequence of statements are executed until some conditions for termination of the loop are satisfied.**
- A program loop thus has 2 segments;
    - **Body of the loop.**
    - **The control statement.**
- The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.
- Control structures can be classified into two based on the position of the control statement in the loop;
    - **Entry-controlled loop** OR **pre-test loop.**
    - **Exit-controlled loop** OR **post-test loop.**

- In the entry-controlled loop,
  - The control conditions are tested before the start of the loop execution.
  - If the conditions are not satisfied, then the body of the loop will not be executed.
- In an exit-controlled loop,
  - The test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.



- In general, a looping process includes the following 4 steps:
  - Setting and initialization of a condition variable.
  - Execution of statements in the loop.
  - Test for specified value of the condition variable for execution of the loop.
  - Incrementing or updating the condition variable.
- The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.
- C language provides three constructs for performing loop operations:
  - The *while* statement.
  - The *do* statement.
  - The *for* statement.

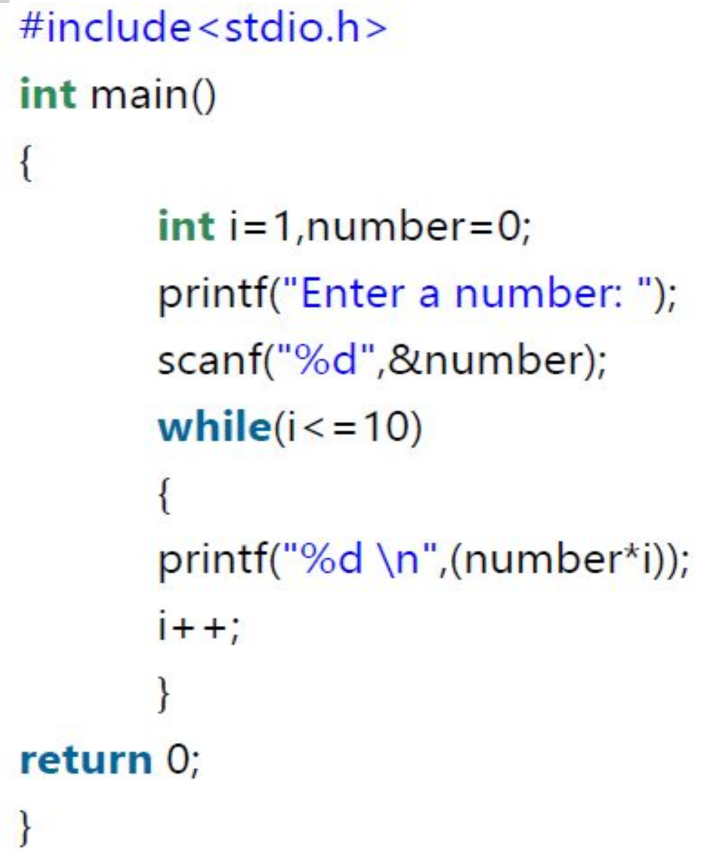
The ***while*** statement:

- The simplest of all the looping structures in C.
- The basic format of the *while* statement is ;

```
        while (test condition)

        {

            body of the loop

        }
```

- The *while* is an **entry-controlled loop statement**.
- The *test condition* is evaluated and if the condition is true, then the *body of the loop* is executed.
- After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again.
- This process is repeated until the test condition finally becomes false and the control is transferred out of the loop.
- On exit, the program continues with the statement immediately after the body of the loop.

- The body of the loop may have one or more statements.
- The braces are needed only if the body contains two or more statements.
- Eg: C program to print the multiples of a given number using while loop in C



```
#include<stdio.h>
int main()
{
        int i=1,number=0;
        printf("Enter a number: ");
        scanf("%d",&number);
        while(i<=10)
        {
        printf("%d \n",(number*i));
        i++;
        }
return 0;
}
```

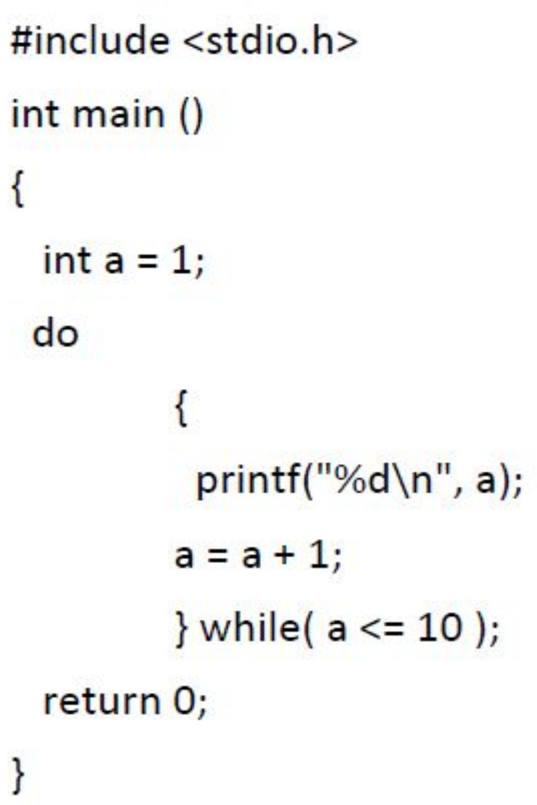
The ***do*** statement:

- Certain occasion demands the **execution of body of the loop before performing the test.**
    - In *while* loop construct, test condition is checked before the loop execution.
- *do* statements are used in such scenarios.
- *do….while* construct provides an **exit-controlled loop** and therefore **the body of the loop is always executed at least once.**
- Following is the general form of *do* statement;

```
do
{
    body of the loop
}
while (test-condition);
```

- On reaching the ***do*** statement, the program proceeds to evaluate the body of the loop first.
- At the end of the loop, the *test-condition* in the *while* statement is evaluated.
- If the condition is true, the program continues to evaluate the body of the loop once again.
- This process continues as long as the condition is true.
- When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the *while* statement.

- Eg: C program to print numbers from 1 to 10 using do...while loop,



```
#include <stdio.h>
int main ()
{
  int a = 1;
 do
          {
            printf("%d\n", a);
          a = a + 1;
          } while( a <= 10 );
  return 0;
}
```

The ***for*** statement:

- It is an **entry-controlled loop** that provides more concise loop control structure.
- The general form of the for loop is;

```
for (initialization ; test-condition ; increment )
{
          body of the loop
}
```

- The execution of the *for* statement is as follows;
    - **Initialization of the control variables** is done first, using assignment statements such as i=1 and count=0. The variables i and count are called *loop control variables*.

- **The value of the control variable is tested using the test condition**. The test condition is a relational expression, such as i < 10 that determines when the loop will exit.
  - If the condition is true, the body of the loop is executed.
  - If the condition is false, the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the *for* statement after evaluating the last statement in the loop.
  - Now, the control variable is incremented using an assignment statement such as i = i +1 and the new value of the control variable is again tested to see whether it satisfies the loop condition.
  - If the condition is satisfied, the body of the loop is again executed.
  - Till the value of the control variable fails to satisfy the test condition, this process continues.

- Eg: C program segment to print digits from 0 to 10 in a line.

```
for ( i = 0 ;  i <= 10 ; i = i+1 )
      {
            printf("%d", i);
      }
      printf("\n");
```

- The *for* loop allows negative increments too. The before mentioned *for* loop can be modified as follows to print the digits 9 to 0 in a line.

```
for ( i = 9 ;  i >= 0 ; i = i-1 )
      {
            printf("%d", i);
      }
      printf("\n");
```

Additional features of *for* loop:

- The for loop in C has several capabilities that are not found in other loop constructs.
    - More than one variable can be initialized at a time in the for statement.
        - The multiple arguments in the initialization section are separated by commas.
    - More than one variable can be incremented in the for statement.
        - The multiple arguments in the increment section are separated by commas.
    - The test condition may have any compound relation and the testing need not be limited only to the loop control variable.
    - If necessary, one or more sections of *for* loop can be omitted.
        - However semicolons separating the sections must remain.