

Week 3

C language:

Variable Names, Data Types and Sizes, Constants, Declarations,
Arithmetic Operators, Relational & Logical Operators, Increment & Decrement Operators, Bitwise Operators, Assignment Operators & Expressions, Type Conversions Conditional Expressions, Precedence and Order of Evaluation

C Language Components

The four main components of C language are

- 1) The Character Set.
- 2) Tokens
- 3) Variables
- 4) Data Types

1) The Character Set : Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other sign.

Letters AB.....Z ab.....z

Digits 0 1.....9

Special symbols # @* &.....

White spaces Blank spaces ,horizontal tabs, new line

2) Tokens: The smallest individual unit in a program is known as a token. Tokens are like individual words and punctuation marks in English language sentence.

C has six tokens

- i. Keywords
- ii. Identifiers
- iii. Constants
- iv. Strings
- v. Operators
- vi. Special symbols

KEYWORDS :

There are certain words, called keywords (reserved words) that have a predefined meaning in 'C' language. These keywords are only to be used for their intended purpose and not as identifiers. The following table shows standard 'C' keywords

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

IDENTIFIERS: Identifiers are the names given to various program elements such as variables, functions and arrays. These are user defined names consisting of sequence of letters and digits.

Rules for declaring identifiers:

- The first character must be an alphabet or underscore.
- It must consist of only letters, digits and underscore.
- Identifiers may have any length but only first 31 characters are significant.
- It must not contain white space or blank space.
- Do not use keywords as identifiers.
- Upper and lower case letters are different.

Example: ab Ab aB AB are treated differently

Examples of valid identifiers:

a, x, n, num, SUM, fact, grand_total, sum_of_digits, sum1

Examples of *Invalid identifiers*: \$amount, ³num', grand-total, sum of digits, 4num.

- \$amount : Special character is not permitted
- grand-total : hyphen is not permitted.
- sum of digits : blank spaces between the words are not allowed.
- 4num : should not start with a number (first character must be a letter or underscore

Note: Some compilers of C recognize only the first 8 characters only; because of this they are unable to distinguish identifiers with the words of length more than eight characters.

CONSTANTS:

Constants refer to values that do not change during the execution of a program.

Constants can be divided into two major categories:

1.Primary constants:

a)Numeric constants

Integer constants.

Floating-point (real) constants.

b)Character constants

Single character constants

String constants

2.Secondary constants:

Enumeration constants.

Symbolic constants.

Rules for declaring constants:

- 1.Commas and blank spaces are not permitted within the constant.
- 2.The constant can be preceded by minus (-) signed if required.
- 3.The value of a constant must be within its minimum bounds of its specified data type.

Integer constants: An integer constant is an integer-valued number. It consists of sequence of digits. Integer constants can be written in three different number systems:

- 1.Decimal integer (base 10).
- 2.Octal integer (base 8).
- 3.Hexadecimal (base 16).

Decimal integer constant: It consists of set of digits, 0 to 9.

Valid declaration: 0, 124, -56, + 67, 4567 etc.

Invalid declaration: \$245, 2.34, 34 345, 075.

23,345,00. it is also an invalid declaration.

Note: *Embedded spaces, commas, characters, special symbols are not allowed between digits*

They can be preceded by an optional + or \pm sign.

Octal integer: It consists of set of digits, 0 to 7.

Ex: 037, 0, 0765, 05557 etc. (valid representation)

It is a sequence of digits preceded by 0.

Ex: Invalid representations

0394: digit 9 is not permitted (digits 0 to 7 only)

235: does not begin with 0. (Leading number must be 0).

Floating point constants or Real constants : The numbers with fractional parts are called real constants.

These are the numbers with base-10 which contains either a decimal part or exponent (or both).

Representation: These numbers can be represented in either decimal notation or exponent notation (scientific notation).

Decimal notation: 1234.56, 75.098, 0.0002, -0.00674
(valid notations)

Exponent or scientific notation:

General form: Mantissa e exponent

Mantissa: It is a real number expressed in decimal notation or an integer notation.

Exponent: It is an integer number with an optional plus (+) or minus (-) sign.

E or e: The letter separating the mantissa and decimal part.

Ex: (Valid notations)

1.23456**E**+3 (1.23456×10^3)

7.5098 **e**+1 (7.5098×10^1)

2**E**-4 (2×10^{-4})

These exponential notations are useful for representing numbers that are either very large or very small.

Ex: 0.00000000987 is equivalent to **9.87e-9**

Character constants:-

Single character constants: It is character(or any symbol or digit) enclosed within single quotes.

Ex: 'a' '1' '*'

Every Character constants have integer values known as **ASCII** values

ASCII:- ASCII stands for American Standard Code for Information Interchange. Pronounced ask-ee, ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 255. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort.

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	`	127	7F	DEL

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	Ł	224	E0	α
129	81	ù	161	A1	í	193	C1	ł	225	E1	β
130	82	é	162	A2	ó	194	C2	ŧ	226	E2	Γ
131	83	â	163	A3	ú	195	C3	ţ	227	E3	π
132	84	à	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	á	165	A5	Ñ	197	C5	†	229	E5	σ
134	86	â	166	A6	*	198	C6	‡	230	E6	μ
135	87	ç	167	A7	°	199	C7	‡	231	E7	ι
136	88	ê	168	A8	Ł	200	C8	Ł	232	E8	Φ
137	89	ë	169	A9	ŕ	201	C9	ƒ	233	E9	Θ
138	8A	è	170	AA	˘	202	CA	Ł	234	EA	Ω
139	8B	ı	171	AB	½	203	CB	ƒ	235	EB	δ
140	8C	î	172	AC	¼	204	CC	ƒ	236	EC	=
141	8D	ı	173	AD	ı	205	CD	=	237	ED	φ
142	8E	Ä	174	AE	ı	206	CE	÷	238	EE	ε
143	8F	Å	175	AF	ı	207	CF	÷	239	EF	ŋ
144	90	Ê	176	B0	ı	208	D0	Ł	240	F0	≡
145	91	æ	177	B1	ı	209	D1	ƒ	241	F1	±
146	92	Æ	178	B2	ı	210	D2	ƒ	242	F2	≥
147	93	ô	179	B3	ı	211	D3	Ł	243	F3	≤
148	94	ó	180	B4	ı	212	D4	Ö	244	F4	
149	95	ò	181	B5	ı	213	D5	ƒ	245	F5	
150	96	ú	182	B6	ı	214	D6	ƒ	246	F6	+
151	97	û	183	B7	ı	215	D7	†	247	F7	~
152	98	ÿ	184	B8	ı	216	D8	†	248	F8	~
153	99	Û	185	B9	ı	217	D9	ı	249	F9	.
154	9A	Ü	186	BA	ı	218	DA	ı	250	FA	.
155	9B	Ÿ	187	BB	ı	219	DB	ı	251	FB	√
156	9C	£	188	BC	ı	220	DC	ı	252	FC	ˆ
157	9D	¥	189	BD	ı	221	DD	ı	253	FD	ˆ
158	9E	Ps	190	BE	ı	222	DE	ı	254	FE	■
159	9F	/	191	BF	ı	223	DF	ı	255	FF	

String constants or string literal:

String constant is a *sequence of zero or more characters* enclosed by double quotes.

Example:

“GITAM”

“12345”

“*)(&%”

Escape Sequences or Backslash Character Constants

C language supports some nonprintable characters, as well as backslash (\) which can be expressed as escape sequences. An escape sequence always starts with backslash followed by one or more special characters. For example, a new line character is represented “\n”

These are used in formatting output screen, i.e. escape sequence are used in output functions. Some escape sequences are given below:

Escape sequence	Character represented
\a	Alert (Beep, Bell)
\b	Backspace
\f	Formfeed Page Break
\n	Newline (Line Feed); see below
\t	Horizontal Tab
\v	Vertical Tab
\\	Backslash
\'	Apostrophe or single quotation mark
\"	Double quotation mark
\?	Question mark)

DATA TYPES:

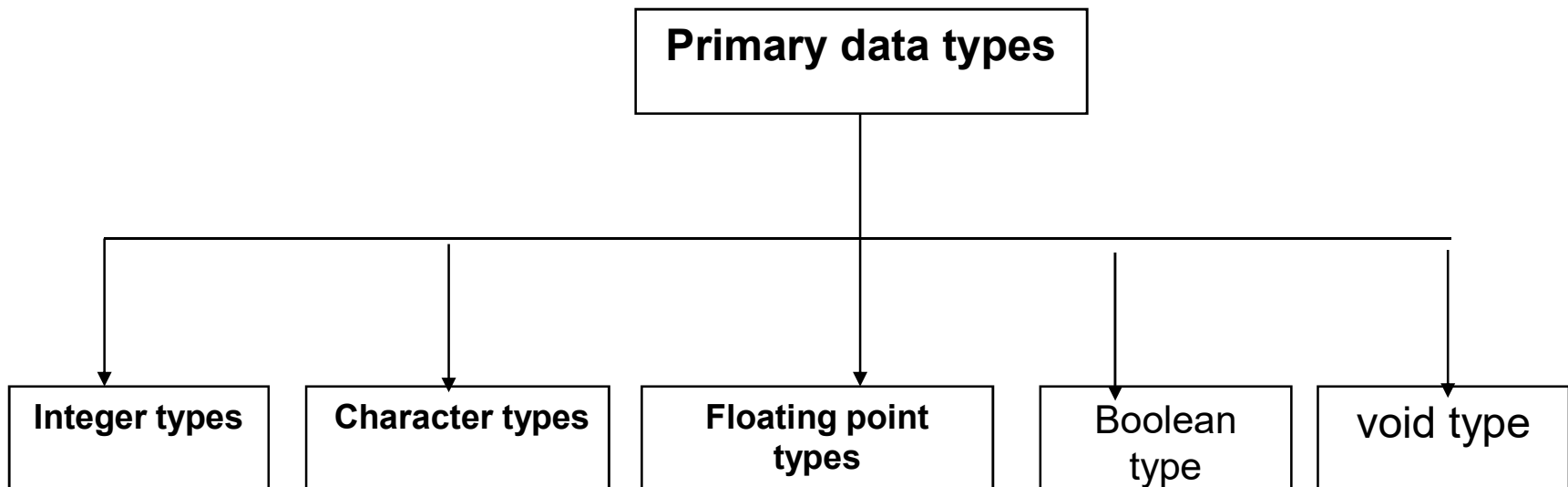
A data type is used to indicate the type of data value stored in a variable. All C compilers support a variety of data types. This variety of data types allows the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports the following classes of data types:

- 1.Primary (fundamental) data types.
- 2.Derived data types.
- 3.User-defined data types

Primary data types:

- 1.integer data type**
- 2.character data type**
- 3.float point data type**
- 4.Boolean data type**
- 5.void data type**



integer data type(int):-

This data type is used to store whole numbers. These numbers do not contain the decimal part. The size of the integer depends upon the word length of a machine (16-bit or 32-bit). On a 16-bit machine, the range of integer values is -32,768 to +32,767.

integer variables are declared by keyword **int**.

C provides control over range of integer values and storage space occupied by these values through the data types:

short int, int, long int in both signed and unsigned forms.

Signed integers: (16-bit machine):

A signed integer uses 1 bit for sign and 15 bits for the magnitude of the number

Size Qualifier :Size Qualifiers are prefixed to the primary data types to increase or decrease the space allocated to the variable.

Note:The size allocated to a data type depends on the compiler and the machine on which the compiler is installed.

short and long are the size qualifiers

Sign Qualifier It specifies whether a variable can hold a negative value or not. Sign qualifiers are used with int and char type.

C supports two sign qualifier, **signed and unsigned**.

A signed qualifier specifies a variable can hold both positive as well as negative integers.

An unsigned qualifier specifies a variable will only positive integers.

Note:

By Default any variable declared is signed

16 Bit Compiler

Data type	Memory occupied by Bytes	Range	Format Specifier
short int	2	-32768 to +32767 (-2 ¹⁵ to 2 ¹⁵ -1)	%hd
int	2	-32768 to +32767 (-2 ¹⁵ to 2 ¹⁵ -1)	%d
long int	4	-2147483648 to +2147483647 (-2 ³¹ to 2 ³¹ -1)	%ld

16 Bit Compiler

Data type	Memory occupied by Bytes	Range	Format Specifier
unsigned short int	2	0 to 65535 (0 to $2^{16}-1$)	%hu
unsigned int	2	0 to 65535 (0 to $2^{16}-1$)	%u
unsigned long int	4	0 to +4294967295 (0 to $2^{32}-1$)	%lu

32 Bit Compiler

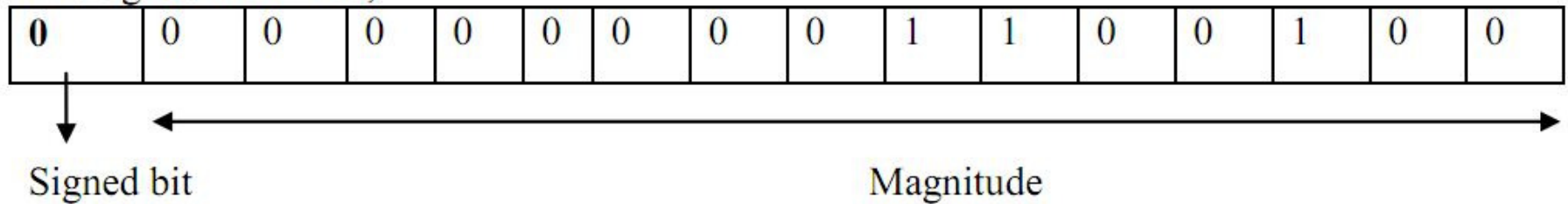
Data type	Memory occupied by Bytes	Range	Format Specifier
short int	2	-32768 to +32767 (-2 ¹⁵ to 2 ¹⁵ -1)	%hd
int	4	-2147483648 to +2147483648 (-2 ³¹ to 2 ³¹ -1)	%d
long int	4	-2147483648 to +2147483648 (-2 ³¹ to 2 ³¹ -1)	%ld
long long int	8	(-2 ⁶³ to 2 ⁶³ -1)	%Ld

32 Bit Compiler

Data type	Memory occupied by Bytes	Range	Format Specifier
unsigned short int	2	0 to 65535 (0 to $2^{16}-1$)	%hu
unsigned int	4	0 to +4294967295 (0 to $2^{32}-1$)	%u
unsigned long int	4	0 to +4294967295 (0 to $2^{32}-1$)	%lu
unsigned long long int	8	(0 to $2^{64}-1$)	%Lu

A signed integer uses 1 bit for sign and 15 bits for the magnitude of the number.
 (-2^{15} to $+2^{15}-1$).

Ex: signed int x=100;



MSB(most significant bit)

$$\begin{aligned}
 &0 \cdot 2^{15} + 0 \cdot 2^{14} + 0 \cdot 2^{13} + 0 \cdot 2^{12} + 0 \cdot 2^{11} + 0 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 \\
 &+ 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
 &= 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 64 + 32 + 0 + 0 + 4 + 0 + 0 \\
 &= 100_{(10)}
 \end{aligned}$$

$$100_{(10)} = 00000000001100100_{(2)}$$

Representation of negative number :

$$-100_{(10)} = 1111111110011100_{(2)}$$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0

$$\begin{aligned} & -1 * 2^{15} + 1 * 2^{14} + 1 * 2^{13} + 1 * 2^{12} + 1 * 2^{11} + 1 * 2^{10} + 1 * 2^9 + 1 * 2^8 + 1 * 2^7 \\ & + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 \\ & = -32768 + 16384 + 8192 + 4096 + 2048 + 1024 + 512 + \\ & 256 + 128 + 0 + 0 + 16 + 8 + 4 + 0 + 0 \\ & = -100_{(10)} \end{aligned}$$

NOTE: **Signed bit (MSB BIT):** 0 represents positive integer, 1 represents negative numbers

Turbo C/C++(16 Bit) is an Integrated Development Environment and compiler for the C programming language from Borland. First introduced in 1987, latest version is in 1992: Turbo C++ 3.0

GCC(32 bit): The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU tool chain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987 and the compiler was extended to compile C++ in December of that year

Character data type: (char) This data type is used to declare variables to store character data. Character data type occupies one byte of memory for storage of character. The qualifiers **signed** or **unsigned** can be applied on char data type. **char** is the key word used for declaring variables

Data type	Memory	Range	Format Specifier
char or signed char	1 Byte	-128 to 127 (-2^7 to $+2^7-1$)	%c
unsigned char	1 Byte	0 to 255 (0 to $+2^8-1$)	%c

Floating Point data Types:

Floating point number represents a real number with 6 digits precision occupies 4 bytes of memory. Floating point variables are declared by the keyword **float**.

Double floating point data type occupies 8 bytes of memory giving 14 digits of precision. These are also known as double precision numbers. Variables are declared by keyword **double** **long double** refers to a floating point data type that is often more precise than double precision.

Size and range of floating point data type is shown in the table:

Data type (key word)	Size (memory)	Range	format specifier
Float	32 bits (4 bytes)	3.4E-38 to 3.4E+38	%f
double	64 bits (8 bytes)	1.7E-308 to 1.7E +308	%lf
long double	80 bits (10 bytes)	3.4E-4932 to 1.1E+4932	%Lf

Note:The size of long double is 12 bytes in gcc compiler and other compilers in Linux operating system. Size also depends on operating system and System configuration

Boolean data type:-

Boolean or logical data type is a data type having two values (usually denoted true and false), intended to represent the truth values of logic and Boolean algebra. It is named after George Boole, who first defined an algebraic system of logic in the mid 19th century. The Boolean data type is the primary result of **conditional statements**, which allow different actions and change control flow depending on whether a programmer-specified Boolean condition evaluates to true or false.

C99 added a Boolean (true/false) type which is defined in the <stdbool.h> header

Boolean variable is defined by keyword **bool**;

Ex:

```
bool b;
```

where b is a variable which can store true(1) or false (0)

Void type

The void type has no values. This is usually used to specify the return type of functions. The type of the function said to be void when it does not return any value to the calling function. This is also used for declaring general purpose pointer called void pointer.

Derived data types.

Derived data types are used in 'C' to store a set of data values. Arrays , Structures , Union and pointer are examples for derived data types.

User-defined data types:

The data types defined by the user are known as the user-defined data types. C provides two identifiers ***typedef*** and ***enum*** to create new data type names.

Typedef: It allows the user to define an identifier that would represent an existing data type.

Syntax:

typedef Existing New_type;

For Example ,the declaration ,

typedef int Integer;

makes the name Integer a synonym of int.

Now the type Integer can be used in declarations ,type castings,etc like

Integer num1,num2;

Which will be treated by the C compiler as the declaration of num1,num2as int variables.

Enumerated data type:

An enumerated type (also called enumeration or enum) is a data type consisting of a set of named values called elements, members or enumerators.

The enumeration data type is defined as follows:

Syntax: enum identifier {value 1,value 2,.....value N};

Identifier: it is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (*enumeration constants*).

Declaration of variables of this new type:

Ex:

```
enum week{ sunday, monday, tuesday, wednesday, thursday, friday,
saturday};
```

Declaration of variable:

```
enum week today;  
today=wednesday;
```

Example of enumerated type

```
#include <stdio.h>  
enum week{ sunday, monday, tuesday, wednesday, thursday, friday,  
saturday};  
int main(){  
    enum week today;  
    today=wednesday;  
    printf("%d day",today+1);  
    return 0;  
}
```

Output

4 day

Variables:A named memory location is called variable.

OR

It is an identifier used to store the value of particular data type in the memory.

Since variable name is identifier we use following rules which are same as of identifier

Rules for declaring Variables names:

- The first character must be an alphabet or underscore.
- It must consist of only letters, digits and underscore.
- Identifiers may have any length but only first 31 characters are significant.
- It must not contain white space or blank space.
- We should not use keywords as identifiers.
- Upper and lower case letters are different.
- Variable names must be unique in the given scope
- Ex:int a,b,a;//is in valid

Int a,b;//is valid

Variable declaration: The declaration of variable gives the name for memory location and its size and specifies the range of value that can be stored in that location.

Syntax:

int a=10;

a

10

 2000

float x=2.3;

x

2.300000

 5000

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c programs", .

There are two ways to define constant in C programming.

1. **const keyword**
2. **#define preprocessor**

1) using **const keyword**:The const keyword is used to define constant in C programming.

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

```
    int main()
```

```
{
```

```
    const float PI=3.14;
```

```
    printf("The value of PI is: %f",PI);
```

```
    return 0;
```

```
}
```


2)Using #define preprocessor Directive :

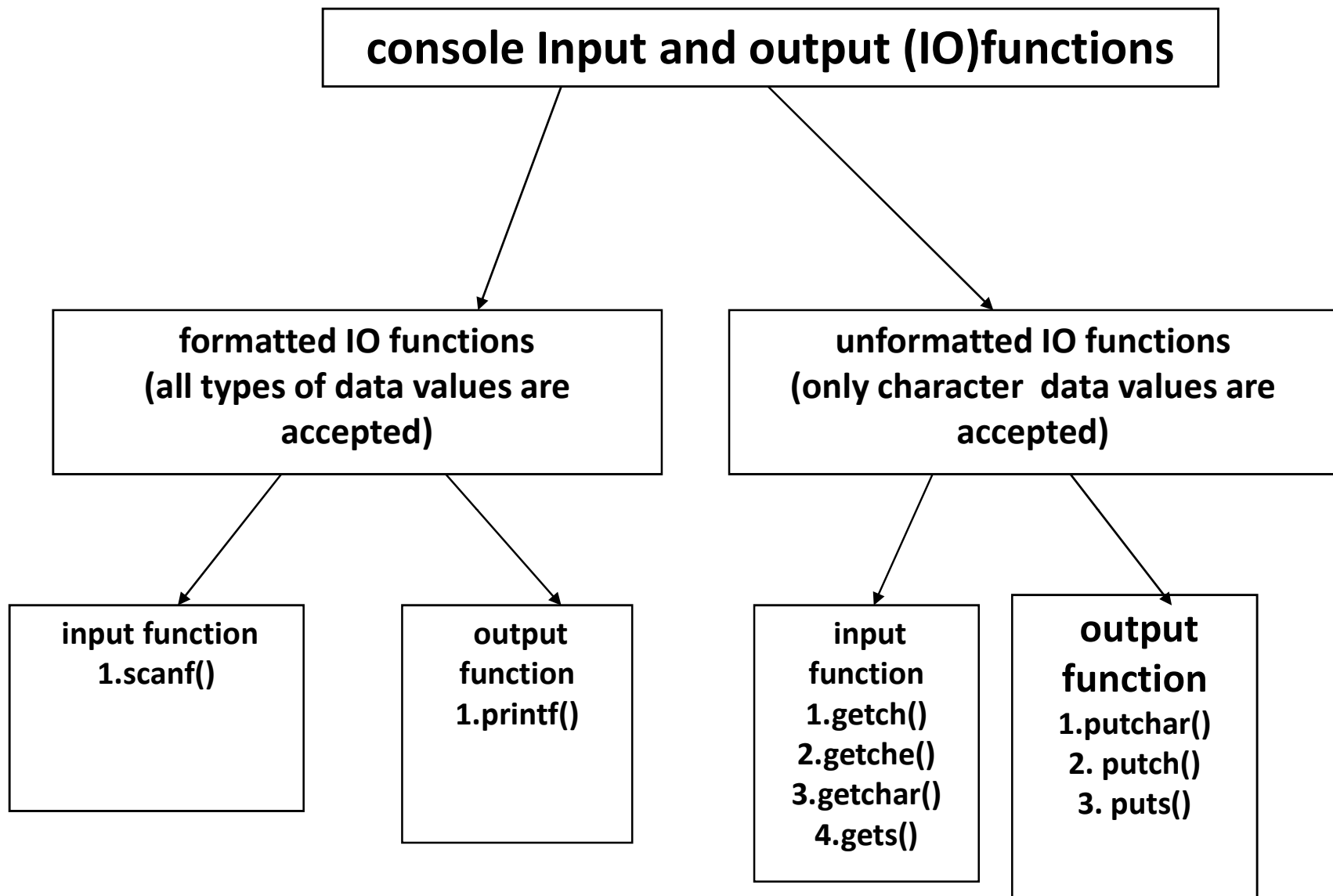
The #define preprocessor is also used to define constant.

```
#include<stdio.h>
#define PI 3.14
int main(){
    printf("The value of PI is: %f",PI);
return 0;
}
```

Basic Input / Output functions: Input & output operations on data can be done by using input and output functions. An input function reads data from keyboard and stores in the variable where as output function prints value in the variable on the screen

These functions are classified into two categories.

1. Formatted input/output functions
2. Unformatted input/output functions



Formatted Input function: scanf()

It accepts an input data through keyboard.

syntax

```
scanf("format string",&variable1..., &variableN);
```

scanf () consists of two parts.

1. format specifiers enclosed within double quotes. format specifier is used to specify the type of format accepted by the scanf to the specified variable. format specifier starts with "%" followed one or two characters

2. Corresponding Variable list preceded by & for each variable

ex: `scanf("%d%d%d",&v1,&v2,&v3);`

& is address operator which gives memory address of variable for storing data read from keyboard.

Formatted Output function: This prints data on the console(output screen).

printf() is the formatted output function.

printf (): This function prints all type of data values to the console.

it consists of two sections. first section contains format string and next part contains list for corresponding variables for specified format string.

Syntax:

printf("format String",v1,.....,vN);

Example:

ex: printf("%d%d%d",v1,v2,v3);

Format specifier	Data type
%hd	signed short int Or short int or short
%hu	unsigned short int
%d or %i	signed int Or int
%u	unsigned int
%ld	long int or signed long int
%lu	unsigned long int
%Ld	long long int Or signed long int
%Lu	unsigned long long int
%o or %O	Octal integer
%x or %X	Hexa decimal integer
%c	Char
%f	Float
%lf	Double
%Lf	long double
%s	String(group of characters)
%p	To print memory address of a variable

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *precision* and *code*

%	FLAG	WIDTH MODIFIER	precision	SIZE	CODE
----------	-------------	-----------------------	------------------	-------------	-------------

<i>flags</i>	description
-	Left-justify within the given field width; Right justification is the default (see <i>width</i> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.

<i>width</i>	description
(<i>number</i>)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

<i>Precision</i>	description
Dot (.)	Separates the fractional part. If the period is specified without an explicit value for <i>precision</i> , 0 is assumed.
<i>Size</i>	Allocates no of positions for fractional part

CODE is conversion symbol for data type


```

#include<stdio.h>
int main ( )
{
printf("\n%d",1234); //Without field width
printf("\n%6d",1234); // Right alignment
printf("\n%-6d ",1234); //Left alignment
printf("\n%012d",1234); //Padding with Zero
printf("\n%2d",1234); //Overriding the field width value.
return 0;
}

```

Output:

1	2	3	4								
		1	2	3	4						
1	2	3	4								
0	0	0	0	0	0	0	0	1	2	3	4
1	2	3	4								

For floating point Number:

The output of a real number may be displayed in two ways:

Decimal Notation ex: 2.345, 0.000005, -2.8976, 3456.987 etc.

Exponential Notation Output of Real Number in decimal notation: %w.p if Here both **w** and **p** are integers.

The integer **w** indicates the minimum number of positions that are to be used for display of the value.

The integer **p** indicates the number of digits to be displayed after decimal point (precision).

The value when displayed is rounded to p decimal places and printed right-justified in

```

#include<stdio.h>
int main()
{
float y= 98.7654;
    printf("\n%f", y);
    printf("\n%7.4f",y);
    printf("\n%7.2f",y);
    printf("\n%-7.2f",y);
return 0;
}

```

9	8	.	7	6	5	4
9	8	.	7	6	5	4
		9	8	.	7	7
9	8	.	7	7		

Printing of a single character:

The format specification for printing a character is:

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

```
int main()
```

```
{
```

```
    printf("%c", 'S');
```

```
    printf("\n%3c", 'S');
```

```
    printf("\n%6c", 'S');
```

```
return 0;
```

```
}
```

S

		S
--	--	---

					S
--	--	--	--	--	---

UNFORMATTED IO FUNCTIONS

INPUT FUNCTION

1.getchar()

2.gets()

3.getch()

4.getche()

Note:

getch() and **getche()** is a nonstandard function and is present in **conio.h** header file which is mostly used by MS-DOS compilers like Turbo C/C++.

It is not part of the C standard library or ISO C

1.getchar():-reads a character from keyboard and it requires pressing of enter key

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

2.gets():-reads a string from keyboard

```
char ch[10];// character Array  
gets(ch);
```

OUTPUT FUNCTIONS:-

1.putchar()

2.puts()

1.putchar():-prints a character on to the screen

`char ch;`

`ch=getchar();`

`putchar(ch);`

2.puts():- prints a string on to the screen

`char ch[10];//Character array`

`gets(ch);`

`puts();`

OPERATORS AND EXPRESSIONS

An ***operator*** is a symbol which represents a particular operation that can be performed on data.

An ***operand*** is variable on which an operation is performed.

By combining the operators and operands we form an *expression*. An ***expression*** is a sequence of operands and operators that reduces to a single value.

C operators can be classified as

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment or Decrement operators
6. Conditional operator
7. Bit wise operators
8. Special operators
- 9.unary operator

1. ARITHMETIC OPERATORS : All basic arithmetic operators are present in C.

operator meaning

+	add
-	subtract
*	multiply
/	division
%	modulo division(remainder)

/*C program on Integer Arithmetic Expressions*/

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

```
int main()
```

```
{ int a, b;
```

```
a=20;b=3;
```

```
    printf("a+b=%d", a+b);
```

```
    printf("\na-b=%d", a-b);
```

```
    printf("\na*b=%d", a*b);
```

```
    printf("\na/b=%d", a/b);
```

```
    printf("\n a mod b=%d", a%b);
```

```
return 0;
```

```
}
```

OUTPUT:

a+b=23

a-b=17

a*b=60

a/b=6

a mod b=2

2. RELATIONAL OPERATORS : We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

operator	meaning
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

/* C program on relational operators*/

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

```
int main()
```

```
{
```

```
int a,b;
```

```
    printf("Enter a, b values:");
```

```
    scanf("%d %d", &a, &b);
```

```
    printf("a>b:%d",a>b);
```

```
    printf("\na<b:%d",a<b);
```

```
    printf("\na>=a:%d",a>=a);
```

```
    printf("\na<=b:%d",a<=b);
```

```
    printf("\na==b:%d",a==b);
```

```
    printf("\na!=b:%d",a!=b);
```

```
return 0;
```

```
}
```

OUTPUT:

Enter a, b values: 5 9

a>b: 0 //false

a<b: 1 //true

a>=a: 1 //true

a<=b: 1 //true

a==b: 0 //false

a!=b: 1 //true

3.LOGICAL OPERATORS:

Logical Data: A piece of data is called logical if it conveys the idea of true or false. In C we use **int/bool** data type to represent logical data. If the data value is zero, it is considered as false. If it is non -zero (1 or any integer other than 0) it is considered as true. C has three logical operators for combining logical values and creating new logical values:

1.Logical AND (&&)

2.Logical OR (||)

3.Logical NOT(!)

Truth tables for AND (&&) and OR (||) operators:

Truth table for NOT (!) operator:

X	!X
0	1
1	0

X	Y	X&&Y	X Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

```

#include<stdio.h>
#include<stdbool.h>
int main()
{
int a,b;
a=0;b=0; /*logical and*/
printf("%d&&%d=%d\n",a,b,a&&b);
a=0;b=1;
printf("%d&&%d=%d\n",a,b,a&&b);
a=1;b=0;
printf("%d&&%d=%d\n",a,b,a&&b);
a=1;b=1;
printf("%d&&%d=%d\n",a,b,a&&b);

```

```

a=0;b=0; /*logical or*/
printf("%d || %d=%d\n",a,b,a || b);
a=0;b=1;
printf("%d || %d=%d\n",a,b,a || b);
a=1;b=0;
printf("%d || %d=%d\n",a,b,a || b);
a=1;b=1;
printf("%d || %d=%d\n",a,b,a || b);
a=0; /*logical not*/
printf("!%d =%d\n",a,!a );
a=1;
printf("!%d =%d\n",a,!a );
return 0;
}

```

OUTPUT:

$0 \& 0 = 0$

$0 \& 1 = 0$

$1 \& 0 = 0$

$1 \& 1 = 1$

$0 \mid 0 = 0$

$0 \mid 1 = 1$

$1 \mid 0 = 1$

$1 \mid 1 = 1$

$\neg 0 = 1$

$\neg 1 = 0$

4.ASSIGNMENT OPERATOR:

The assignment operator will evaluate the operand on the right side of the operator (=) and places its value in the variable on the left.

Note: The left operand in an assignment expression must be a single variable.

There are two forms of assignment:

- Simple assignment
- Compound assignment

Simple assignment :

In algebraic expressions we found these expressions.

Ex: $a=5$; $a=a+1$; $a=b+1$;

Here, the left side operand must be a variable but not a constant. The left side variable must be able to receive a value of the expression. If the left operand cannot receive a value and we assign one to it, we get a compile error.

Compound Assignment:

A compound assignment is a **shorthand notation** for a simple assignment. It requires that the left operand be repeated as a part of the right expression.

Syntax: variable operator=value

Ex:

$A+=1$; it is equivalent to $A=A+1$;

Some of the commonly used shorthand assignment operators are shown in the following table:

Statement with simple assignment operator	Statement with shorthand operator
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * 1$	$a *= 1$
$a = a / 1$	$a /= 1$
$a = a \% 1$	$a \% = 1$
$a = a * (n + 1)$	$a *= n + 1$

5.INCREMENT (++) AND DECREMENT (--) OPERATORS:

The operator ++ adds one to its operand where as the operator - - subtracts one from its operand. These operators are unary operators and take the following form:

Operator	Description
++a	Pre-increment
a++	Post-increment
--a	Pre-decrement
a--	Post-decrement

```
#include<stdio.h>
int main()
{
    int a=1;
    int b=5;
    ++a;
    printf("a=%d\n",a ) ;
    --b;
    printf("b=%d\n",b) ;
    printf("a=%d\n",a++ ) ;
    printf("a=%d\n",a);
    printf("b=%d\n",b-- ) ;
    printf("b=%d\n",b);
    return 0;
}
```

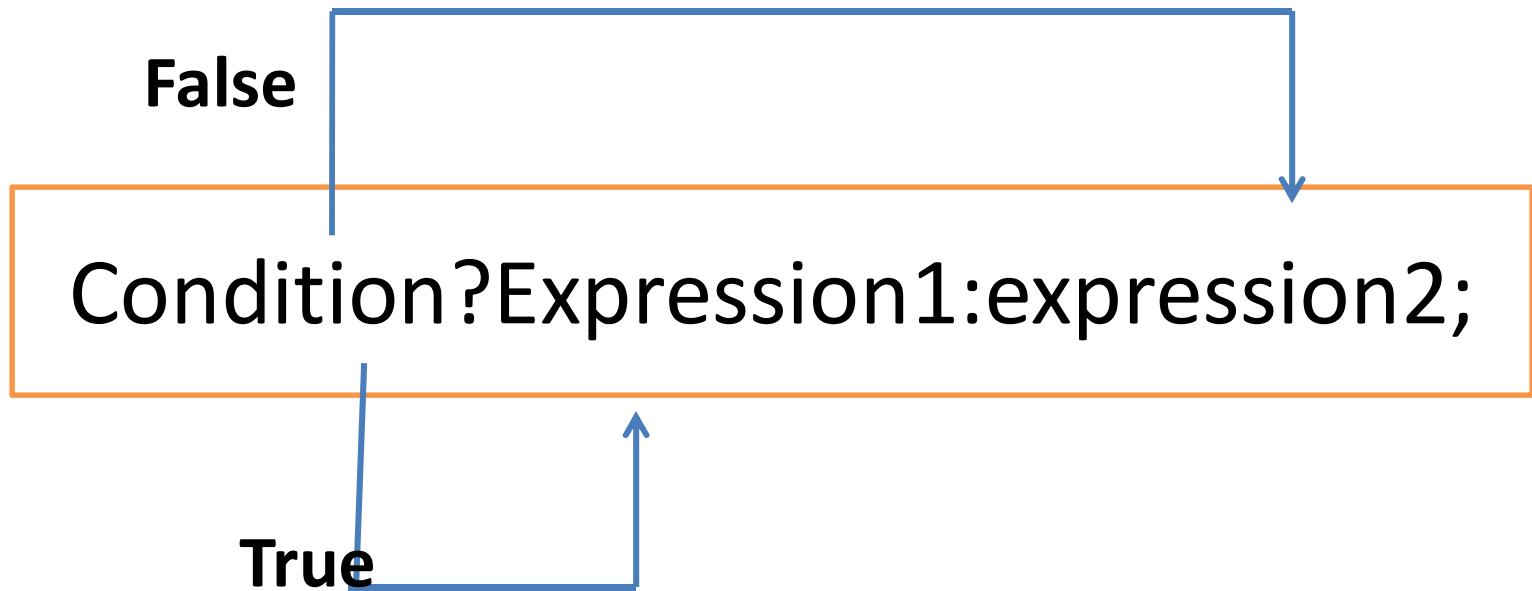
Output:

a=2
b=4
a=2
a=3
b=4
b=3

6.CONDITIONAL OPERATOR OR TERNARY OPERATOR:

A ternary operator requires two operands to operate

Syntax:



```
#include<stdio.h>
int main()
{
    int a, b,c;
        printf("Enter a and b values:");
        scanf("%d%d",&a,&b);
        c=a>b?a:b;
        printf("largest of a and b is %d",c);
return 0;
}
```

Enter a and b values:1 5

largest of a and b is 5

7. BIT WISE OPERATORS : C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.

operator	meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

Bitwise AND operator (&)

The bitwise AND operator is a binary operator it requires two integer operands. It operates on individual bits of the operands. It does a bitwise comparison as shown below:

Op1	OP2	OP1&OP2
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR operator (|)

The bitwise OR operator is a binary operator it requires two integer operands. It operates on individual bits of the operands. It does a bitwise comparison as shown below:

Op1	OP2	OP1 OP2
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise Exclusive OR operator (^)

The bitwise XOR operator is a binary operator it requires two integer operands. It operates on individual bits of the operands. It does a bitwise comparison as shown below:

Op1	OP2	OP1^OP2
0	0	0
0	1	1
1	0	1
1	1	0

Shift Operators

The shift operators move bits to the right or the left. These are of two types:

- Bitwise right shift operator (>>)
- Bitwise left shift operator (<<)

Bitwise right shift operator

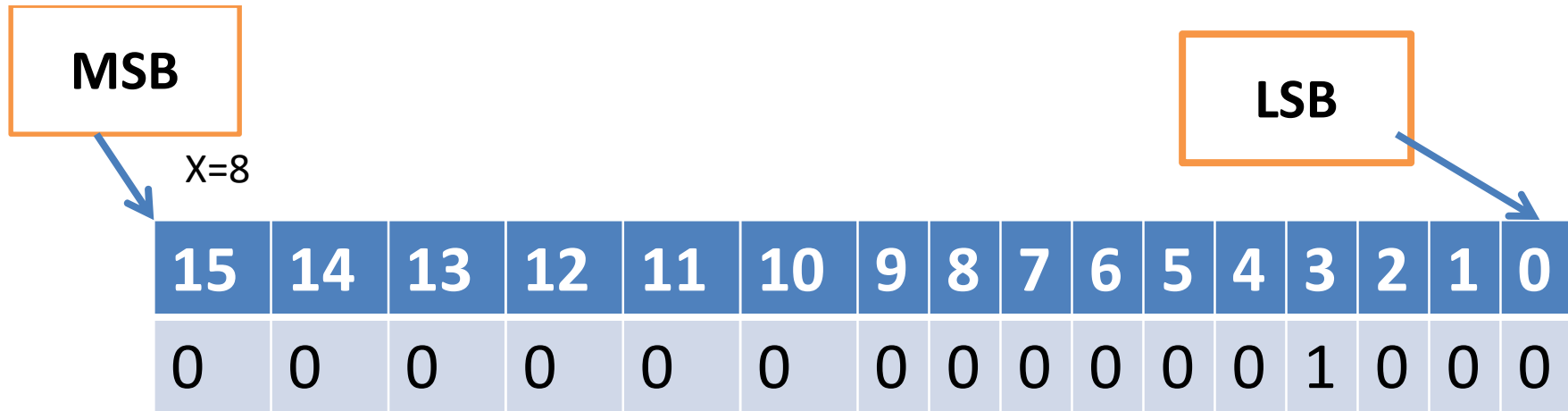
It is a binary operator it requires two integer operands. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted. Here bits are shifted towards right and Zeros are appended at MSB(most significant bit i.e at extreme left) and Extra bits obtained at LBS(least significant bit i.e at 0th bit) is discarded

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

    printf("Enter an integer:");
    scanf("%d",&x);
    x>>=2;  /*x=x>>2*/
    printf("After Right Shift x=%d",x);
return 0;
}
```

Output:

Enter an integer:8
After Right Shift x=2



$X = X \gg 2$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Now X = 2

Extra bits will
be discarded

Bitwise shift left operator (<<)

It is a binary operator it requires two integer operands. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted. Here bits are shifted towards left and Zeros are appended at LBS(least significant bit i.e at 0the bit) and Extra bits obtained at MSB(most significant bit i.e at extreme left) is discarded

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

    printf("Enter an integer:");
    scanf("%d",&x);
    x<<=2; /*x=x<<2*/
    printf("After Left Shift x=%d",x);
return 0;
}
```

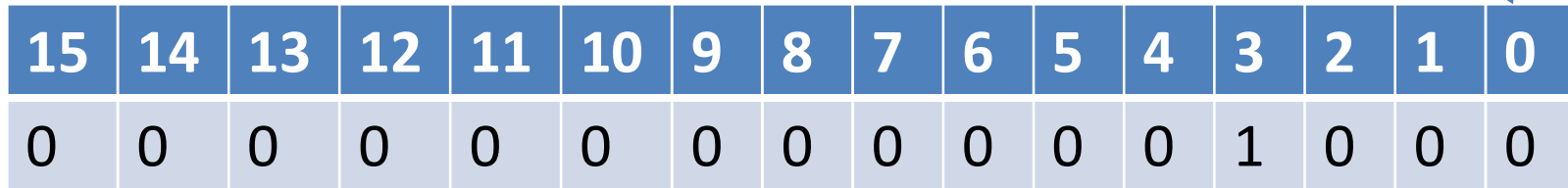
Output:

Enter an integer:2
After Left Shift X=8

X=8

MSB

LSB



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

$X = X \ll 2$

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Extra bits will be
discarded

Now
32

One's compliment or Bitwise Not(\sim (tilde)): It is a unary operator that performs logical negation on each bit.

Expression	\sim Expression
0	1
1	0

$X=32767$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

$X=\sim X$

$X= -32768$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
#include<stdio.h>
int main()
{
int x;
    printf("Enter an integer:");
    scanf("%d",&x);
    x=~x;
    printf("After Not X=%d",x);
return 0;
}
```

Enter an integer: 32767

After Not X= - 32768

8.SPECIAL OPERATORS

These operators which do not fit in any of the above classification are

1.comma operator (,)

2.sizeof()

3.Pointer operators (& and *)

4. member selection operators

dot (.)

arrow (->)

1.Coma operator : It is used to link the related expressions together . It is a binary operator. .It is left-associative.It evaluates expression from left to right and returns the value as a result. It has the **lowest precedence among all C Operators**.

```
#include<stdio.h>
int main()
{
int a,b,c;
c=(a=10,b=20,a+b);
printf("c=%d",c);
return 0;
}
```

Output:

C=30

2.sizeof():It returns no of bytes occupied by a variable in the memory;

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

```
int main()
```

```
{
```

```
printf("Memory occupied by char is %d\n",sizeof(char));
```

```
printf("Memory occupied by int is %d\n",sizeof(int));
```

```
printf("Memory occupied by long int is %d\n",sizeof(long int));
```

```
printf("Memory occupied by long long int is %d\n",sizeof(long long int));
```

```
printf("Memory occupied by float is %d\n",sizeof(float));
```

```
printf("Memory occupied by double is %d\n",sizeof(double));
```

```
printf("Memory occupied by long double is %d\n",sizeof(long double));
```

```
return 0;
```

```
}
```

Output:

Memory occupied by char is 1

Memory occupied by int is 4

Memory occupied by long int is 4

Memory occupied by long long int is 8

Memory occupied by float is 4

Memory occupied by double is 8

Memory occupied by long double is 12

3.Pointer operators(& and *)

Ampersand(&) is the address operator which gives the memory address of a variable

Asterisk (*): Used as a binary operator when used in binary arithmetic . When used as a Unary operator it a **dereference operator, also known as an indirection operator**, It operates on a pointer variable to return the value at an address pointed by pointer variable.

4.member selection operators (. and ->)

Dot operator(.) is used to access members of a structure or union

Arrow operator(→) is used to access members of a structure or union using pointers

9.UNARY OPERATOR: operator which operates on single operand is called unary operator

Operator	Description
+	Unary Plus
-	Unary Minus
++	Increment
--	Decrement
&	Address operator
~	One's Complement
Sizeof()	Memory occupied by any variable
Type	Type casting

TYPE CONVERSION IN EXPRESSIONS

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.. In C we have two types of conversions:

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversion (Automatic Conversion):

Implicit conversion is done automatically by the compiler when you assign a value of one type to another.

```
#include<stdio.h>
int main()
{
float c;
int a;
float b;
    a=10;
    b=20.1;
    c=a+b;
    printf("c=%f",c);
return 0;
}
C=30.100000
```

NOTE:

1. When a float is stored in an integer, the fraction part is dropped. If the integer part is larger than the maximum value that can be stored the results are invalid and unpredictable.
2. When we try to store a long double in a variable of type float, the results are valid if the value fits or invalid if it is too large.

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

```
int main()  
{ int i=23 ;  
  float f=67.5;  
    printf("i=%i\n ",i);  
    printf("f=%f\n ",f);  
    i=f;  
    printf(" i=%i ",i);  
return 0;  
}
```

```
i=23
```

```
f=67.500000
```

```
i=67
```

Explicit Conversion(Type casting): Explicit type conversion is a type conversion which is explicitly defined within a program (instead of being done by a compiler for implicit type conversion). Explicit conversion can be done using type cast operator and the general syntax for doing this is **(type) expression;**

Ex:

```
int a=10;
```

```
float b;
```

```
b=(float)a/3;
```

Precedence and Associativity

Precedence:-precedence is used to determine the order in which different complex expressions are evaluated.

An **expression is evaluated** in **left to right** and value is assigned to variable in left side of assignment operator.

An arithmetic expression without parenthesis will be evaluated from **left to right** using the rules of precedence of operators.

There are two distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + -

Rules for evaluation of expression

1. First parenthesized sub expression left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
4. The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.

Associativity :-It is used to determine the order in which operator with same precedence is evaluated in a complex expression

Associativity can be **left to right** or **from right to left**

Left to right Associativity:- It evaluates the expression by starting on left and moving the right

Ex: $3*8/4\%4*5$

In the above expression $*, /, \%, *$ are having same precedence

$(((3*8)/4)\%4)*5$

$((24/4)\%4)*5$

$((6\%4)*5)$

$(2*5)=10$

Right to left Associativity:

When more than one assignment operator occurs in an assignment expression, assignment operator must be interpreted from right to left

Ex:

$A += B * = C -= 5$ (consider $A=3, B=5, C=8$)

Evaluating above expression

$(A += (B * = (C -= 5)))$

$(A += (B * = (C = C - 5)))$

$(A += (B * = (C = 8 - 5)))$

$(A += (B = B * 3))$

$(A += (B = 5 * 3))$

$(A = A + 15)$

$A = 3 + 15$

$A = 18$

PRIORITY AND ASSOCIATIVITY OPERATORS:

Operators	Description	Associativity	Priority/precedence
()	Function call/ Parenthesis	Left to right	1
[]	Array expression		
->	Structure operator		
.	Structure operator		
+	Unary plus	Right to left	2
-	Unary minus		
++	Increment		
--	Decrement		
!	NOT operator		
~	Ones complement		
*	Pointer operator		
&	Address operator		
Sizeof	Size of operator		
Type	Type casting operator		

*	Multiplication	Left to right	3
/	Division		
%	Mod		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	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 operator	Right to left	13
=, +=, *=, /=, %=, &=, =, <<=, >>=, ^=, -=	Assignment operator	Right to left	14
,	Comma operator	Left to right	15

