Introduction

- Language: A way to communicate. A language is used to communicate between two persons.

- Programming Language: A language which is used to write a program is called programming language. A programming is used to communicate with a computer which can understand only binary numbers composed of 0 or 1.

- Program: A set of instructions given to the computer to perform some specific tasks is called program.

Why should we learn C language ?

C is a Basic of all other programming languages.

A computer can understand only machine language composed of 0 and 1. If a human wants to communicate with computer he requires a programming language like C language. But the instructions of C language is written in English which can not be understood by a computer directly. A compiler is required for C language and this compiler converts the C language to a machine language for a computer. Therefore C is a compiler based programming language.

In other words a program which is used to convert high level programming language to low level programming language is called compiler.

Importance of C language/ Use of C language

- Device drivers, Android core libraries, Oracle and MySQL Database software, Unix Operating Systems all are written in C language.

History of C Language

BCPL (Basic Combined Programming Language) language was developed by Martin Richards in 1966. The purpose of BCPL language was to write Compilers of other languages. Then Ken Thompson developed B language from BCPL language in 1969. Then in 1972 Dennis M. Ritchie developed C language at AT & T Bell Laboratory. It was standardized by ANSI (American National Standard Institute) in 1989. The C language was invented by Ritchie for the development of UNIX operating system.

Chapter 1

C Tokens

In C program the smallest individual units are known as C tokens. There are six types of C tokens.

1) Constants
2) Identifiers
3) Keywords
4) Strings
5) Operators
6) Special Symbols

1) Constants: A constant is a quantity that does not change during the execution of a program.

Types of constants:

1. **Integer constants** – Integer constants are the fixed values formed with whole numbers. There are three types of integer constants namely – decimal constant, octal constant and hexadecimal constant.
   - **Decimal constant** – Decimal constants are the numbers formed with a set of ten digits, 0 through 9. For example – 123, -321, 0, +78 etc.
     No spaces, comma or non-digit characters are allowed between the digits of a decimal constant. For example – 123 56 (invalid), 20,000 (invalid), $1000 (invalid)

   - **Octal constant** – Octal constants are the numbers formed with a set of eight digits, 0 through 7. In C language octal constants must be preceded with a leading zero. Some examples of octal integers are: 037, 0435, 0, 0557 etc.

   - **Hexadecimal constant** – Hexadecimal constants are the numbers formed with a set of sixteen digits, 0 through 9 and A through F. In C language hexadecimal constants must be preceded with 0x. Some examples of hexadecimal constants are: 0xA2, 0xff etc.

2. **Floating point constants** – The quantities are represented by numbers containing fractional parts are known as floating point constants or real constants. For example – 3.44, 0.75, 4.0, -0.75, +247.0, -.71, +.5, 215. etc.
   A floating point constant may also be expressed in exponential form. For example the real constant 215.65 can be represented by 2.1565e2 in exponential form where e indicates the exponential form. The general format of an exponential notation is given below.

   mantissa e exponent

   In the above mentioned example 2.1565 is named as mantissa and 2 is named as exponent. The mantissa is either a integer number or a real number with positive or negative sign and exponent is always a integer number with positive or negative sign. Some examples of floating point constants are given below:

Floating point constants	Floating point constants in exponential form
6500.0	0.65e4 / 0.65E4
0.12	12e-2 / 12E-2
-0.12	-1.2e-1 / -1.2E-1

Examples of some valid and some invalid integer and floating point constants are given below for your clarification.

Constants	Remarks	Reasons
698354L	Valid	Representing long integer
25,000	Invalid	Comma is not allowed
+5.0E3	Valid	ANSI C supports unary plus sign
3.5e-5	Valid	
7.1e 4	Invalid	No blank space is allowed
-4.5e-2	Valid	
1.5E+2.5	Invalid	Exponent must be an integer
$255	Invalid	$ symbol is not allowed
0x7B	valid	Hexadecimal constant

3. **Character constants** – A character constant is a single character enclosed within a pair of single quotation marks. For example – '5', 'X', ':', ' ' etc.

4. **String constant** – A string constant is sequence of characters enclosed within a double quotation marks. For example – “Hello”, “C Programming”, “5 + 3” etc.

2) Identifiers: Identifiers refer to the names of variables, functions and arrays. Basically the identifiers are the user-defined names which consist of a sequence of alphabets, digits and underscore symbols. Therefore variable is an identifier in C language.

Variables: A variable can be considered as a name given to the location in memory where a value is stored. Its value can be changed and can be reused many times. It is a way to represent memory location through some symbols so that it can be easily identified. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

General syntax for declaring of a variable in C program is given below:

Syntax: Data_Type Variable_Name;

Example: int a, b = 5;
char name;

Rules for constructing variable name

1) Variables can have alphabets, digits and underscore.
2) Variable should start with alphabets and underscore only.
3) Variable name should not start with digit
4) Blank space is not allowed in variable name
5) Keywords are not allowed in variable name
6) Length of a variable name should not exceed 8 characters
Examples of some valid and some invalid variable names are given below for your clarification.

Variable name	Remarks	Reasons
First_Var	Valid	
char	Invalid	char is a keyword
Price$	Invalid	Dolar sign ($) is not allowed
group one	Invalid	Blank space is not permitted
average_number	Valid	
int_type	Valid	int is a keyword, but it is a part of the entire variable name

Data types: The type of a variable determines how much space it occupies in storage. ANSI C supports normally four data types as follows.

1. Primary or Basic or fundamental data types
2. User defined data types
3. Derived data types

1. Basic or fundamental data types: There are different fundamental data types supported by many C compilers. The details regarding these basic data types are explained one by one. The size of a variable is defined as the number of bytes occupied by the variable in the memory and this size basically depends on the architecture of the computer processor. In case of 16-bit processor the size of an integer type variable becomes 2 bytes and the size of an integer type variable becomes 4 bytes for a 32-bit processor. The following table gives the size, ranges and specifiers of different kinds of integer data types in C language for a 16-bit computer.

Data Types	Size	Ranges	Specifiers
int or signed int	2 bytes i.e. 16 bits	-32,768 to 32,767 i.e. -2^{15} to $+(2^{15} - 1)$	%d
unsigned int	2 bytes i.e. 16 bits	0 to 65,535 i.e. 0 to $(2^{16} - 1)$	%u
short int or signed short int	1 byte i.e. 8 bits	-128 to 127 i.e. -2^{7} to $+(2^{7} - 1)$	%hd
unsigned short int	1 byte i.e. 8 bits	0 to 255 i.e. 0 to $(2^{8} - 1)$	%hu
long int or signed long int	4 bytes i.e. 32 bits	-2,147,483,648 to 2,147,483,647 i.e -2^{31} to $+(2^{31} - 1)$	%ld
unsigned long int	4 bytes i.e. 32 bits	0 to 4,294,967,295 i.e. 0 to $(2^{32} - 1)$	%lu

The following table gives the size, ranges and specifiers of different kinds of floating data types in C language for a 16-bit computer.

Data Types	Size	Ranges	Specifiers
float	4 bytes i.e. 32 bits	3.4E-38 to 3.4E+38	%f
double	8 bytes i.e. 64 bits	1.7E-308 to 1.7E+308	%lf
long double	10 byte i.e. 80 bits	3.4E-4932 to 3.4E+4932	%Lf

The following table gives the size, ranges and specifiers of different kinds of character data types in C language for a 16-bit computer.

Data Types	Size	Ranges	Specifiers
char or signed char	1 byte i.e. 8 bits	-128 to 127 i.e. -2^7 to $+(2^7 - 1)$	%c
unsigned char	1 byte i.e. 8 bits	0 to 255 i.e. 0 to $(2^8 - 1)$	%c

2. User defined data type: C supports a feature known as “type definition” which allows to define an identifier that represents an existing data type. This user defined data type identifier can be used to declare variables. The general form of user defined data type is given below:

Syntax: typedef type identifier;

For example –

```
typedef int temp_units;

temp_units centigrade, fahrenheit, kelvin;
```

3. Derived data type: Derived data types in C are extensions of fundamental data types, such as integers and floats. They are used to group similar data types together and represent multiple values in a program.

Examples of derived data type are arrays, pointers, functions, structures etc.

i) ***Arrays:*** A collection of similar data types, such as integers, chars, or doubles, stored in a contiguous memory location.

ii) ***Pointers:*** A variable that stores the address of another variable.

iii) ***Functions:*** A block of code that performs a specific task and returns a value of a specified data type.

iv) ***Structure:*** A collection of different types of data items stored in a contiguous memory locations.

3) Keyword: ANSI C has 32 number of fixed and unique words which have some specific meanings. These words are known as keywords in C language. Keywords serve the basic building blocks in C program. All the keywords in C are written in lowercase. All the keywords are listed in the following table.

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

4) String: A string is a sequence of characters enclosed within a double quotation marks and terminated with null character '\0'. The length of a string is the number of characters enclosed with a double quotation mark along with the null character placed at the end of the string. Here it is important to mention that the null character is invisible at the end of the string. For example – "Hello", "C Programming", "5 + 3" etc all are strings. Now if we want to calculate the length of the string "Hello", we have 5 characters ('H', 'e', 'l', 'l', 'o') along with the null character ('\0') at the end. Therefore the length of the string "Hello" will be 6 in C.

In C language a string can not be stored into a variable. In this case a char type one dimensional array is created to store a string, because C language does not have *string* data type to create a string variable. The char type array where the string is stored is displayed on the screen using printf() function wilh "%s" specifier.

For example:

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

In the above program the string "Hello World!" is stored in the char type array "greetings[]" and printed on the screen.

5) Operators: An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables. Normally an operators form a part of the mathematical or logical expressions. C operators are explained in the next section in details. For example - +, - , *, /, %, <, >, &&, | |, &, |, ^, ~, ++, - - etc are the operators in C language.

6) Special Symbols: Some special symbols are used for some particular purposes in a C program. Some special symbols along with their use in a C program are given below.

Special Symbols	Use
#	Pound sign is used in C program to import a header file such as - #include<stdio.h>
< >	It is used to import any header file in C like #include<math.h>
( )	Parenthesis are used in functions
{ }	Curly braces are used to construct the body of a function
[ ]	Third brackets are used to form any array in C
;	Semi-colon is used to terminate any statement in C program inside the body of a function.

Chapter 2

Operators and Expressions

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables. Normally an operators form a part of the mathematical or logical expressions.

C operators can be classified into a number of categories.

1) Arithmetic operator
2) Relational operator
3) Logical operator
4) Assignment operator
5) Increment and decrement operator
6) Conditional operator
7) Bitwise operator
8) Special operator

1) Arithmetic operator: There are five basic arithmetic operators which are supported by C language. The variables on which arithmetic operators are applied are known as operands. In the expression "a + b" a and b (two variables) are called operands and the + sign is the operator.

SL.	Operator	Meaning
1.	+	Addition or unary plus
2.	-	Subtraction or unary minus
3.	*	Multiplication
4.	/	Division
5.	%	Modulo division

a) Integer Arithmetic: When both the operands in a single arithmetic expression are integers, the expression is called an integer expression and the operation is called integer arithmetic.

Important points to remember:

1. An integer arithmetic always gives an integer value as a result.
   For example – If a and b are two integers and a = 14, b = 4 then
   a + b = 18 (Integer)
   a – b = 10 (Integer)
   a * b = 56 (Integer)
   a / b = 3 (Integer)
   a % b = 2 (Integer)

2. During division of two integers the result will be the quotient integer only without any fractional part. For example – If a and b are two integers and a = 10 and b = 6 then
   a / b = 1 (Integer)

3. Modulo division produces the remainder of an integer division. It can not be used for floating point data.

4. During modulo division the sign of the result is always the sign of the first operand i.e. the dividend.

For example – -14 % 3 = -2
-14 % -3 = -2
14 % -3 = 2

b) Real Arithmetic: An arithmetic operation involving only real operands is known as real arithmetic. A real operand may assume values either in decimal or exponential notation. In case of real arithmetic the floating point value will be rounded to the number of significant digits permissible. Normally C language supports six permissible digits after the decimal point to represent a floating point number. That's why the floating point values will be rounded to the six permissible digits. For example – If x is declared as a floating point variable and if it is assigned a constant (6.0 / 7.0) then the variable x will hold the floating point value 0.857143 instead of 0.857142857, because the sixth digit after the decimal point will be rounded to 3 from 2.

float x, y, z;

Expression	Approximate value	Actual value
x = 6.0 / 7.0	x = 0.857143	X = 0.857142857
y = 1.0 / 3.0	y = 0.333333	y = 0.333333333
z = -2.0 / 3.0	z = -0.666667	z = -0.666666666

Modulo division i.e. the operator % can not be used with real operands.

c) Mixed-mode Arithmetic: When one of the operands is real and others are integer, the expression is called mixed-mode arithmetic expression. If any one operand is of the real type, then only the real operation is performed and the result is always a real number. For example –

15 / 10.0 = 1.5 whereas 15 / 10 = 1

2) Relational operator: When we compare two quantities and depending on their relation we take certain decision, then relational operators are used. An expression containing relational operator is known as relational expression. A relational expression always returns a value either 0 or 1. When a relational expression is satisfied or becomes true, it returns 1 and when a relational expression is not satisfied or becomes false, it gives 0.

The relational operators in C are given in the following table.

SL.	Operator	Meaning
1.	<	Less than
2.	>	Greater than
3.	<=	Less than or equal to
4.	>=	Greater than or equal to
5.	==	Equal to
6.	!=	Not equal to

Some examples of relational expressions are given below.

Relational expression	Remarks	Returned value
4.5 <= 10	Expression is true	1
4.5 < -10	Expression is false	0
-35 >= 0	Expression is false	0
10 < 7+5	Expression is true	1
int a = 10, b = 20, c = 12, d = 18; a + b == c + d	Expression is true	1
int a = 10, b = 20, c = 12, d = 18; a + c == b + d	Expression is false	0

When arithmetic expressions are used on either side of a relational operator or both sides of a relational operator, the arithmetic expressions will be evaluated first and then the results will be compared. Due to this reason ***arithmetic operators have higher priority over relational operators.***

3) Logical operator: C has the following logical operators.

SL.	Operator	Meaning
1.	&&	Logical AND
2.	||	Logical OR
3.	!	Logical NOT

Logical operators && and || are used to test more than one condition and make decision depending upon the result of the test. For example – a > b && x == 10.

An expression which combines two or more relational expressions are known as logical expression. In the above example the expression (a > b && x == 10) is an example of logical expression. Like relational expression, a logical expression also returns either 0 or 1 as per the following truth table. Here it is important to mention that in C language any non-zero value is teated as true value i.e. 1.

OP-1	OP-2	Value returned by logical expression OP-1 && OP-2
0	0	0
0	Non-zero	0
Non-zero	0	0
Non-zero	Non-zero	1

OP-1	OP-2	Value returned by logical expression OP-1 && OP-2
0	0	0
0	Non-zero	1
Non-zero	0	1
Non-zero	Non-zero	1

For example –

```
int a, b, x;
 a = 6;
b = 5;
x = 10;
```

a < b	**x == 20**	**Value returned by logical expression a < b && x == 20**
0	0	0
a < b	**x == 10**	**Value returned by logical expression a < b && x == 10**
0	1	0
a > b	**x == 20**	**Value returned by logical expression a > b && x == 20**
1	0	0
a > b	**x == 10**	**Value returned by logical expression a > b && x == 10**
1	1	1

4) Assignment operator: Assignment operator '=' is used to assign the result of an expression to a variable. In addition C has a set of shorthand assignment operator of the following form.

Variable Operator = Expression;

In the following table we have given a set of examples of assignment operator with expression and the corresponding shorthand assignment operator with expression.

Assignment operator with expression	**Shorthand assignment operator with expression**
x = x + y + 1;	x + = y + 1;
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 shorthand assignment operators:

1. What appears on the left-hand side of the assignment operator need not be repeated and therefore it becomes easier to write the shorthand assignment expression.

2. The statement of shorthand assignment operator becomes concise.

3. The statement is more efficient.

5) Increment and decrement operator: C has two very useful operators which are called increment operator and decrement operator.

SL.	Operator	Meaning
1.	++	Increment operator: It adds 1 to the variable and the incremented value is assigned again to the same variable.
2.	--	Decrement operator: It subtracts 1 from the variable and the decremented value is assigned again to the same variable.

If m is variable, then m++ or ++m is equivalent to m = m + 1 or m + = 1.

Similarly m-- or --m is equivalent to m = m – 1 or m - = 1. The m++ and ++m performs the same mathematical operation, but they behaves differently in an expression on the right-hand side of the assignment operator.

int m = 5, y; y = ++m;	int m = 5, y; y = m++;
In the above case, the value of m is incremented first and then the incremented value of m is assigned to y. Therefore after the execution of the above statements we have the following values of the variables m and y. m = 6 and y = 6	In the above case, the value of m is assigned to y first and then incremented. Therefore after the execution of the above statements we have the following values of the variables m and y. m = 6 and y = 5

For example:

```
int n = 15, j = 1, m;
m = n++ - j + 10;
```

After the execution of the above statements we have the following results.

```
n = 16
j = 1
m = 24
```

6) Conditional operator: A ternary operator pair “? :” is available in C to construct the conditional expressions which have the following generalized form.

Exp1 ? Exp2 : Exp3

Exp1 is evaluated first. If it is non-zero i.e. true then the expression EXP2 is evaluated and becomes the value of the expression. If Exp1 is false, Exp3 is evaluated and becomes the value of the expression. Note that only one of the expressions (either Exp2 or Exp3) is evaluated.

For example:

```
int a = 10;
int b = 15;
x = (a > b) ? a : b;
```

As a is smaller than b, the condition (a > b) becomes false. Therefore the value of b will be assigned to the variable x. Hence x will be 15.

```
int a = 20;
int b = 15;
x = (a > b) ? a : b;
```

As a is larger than b, the condition (a > b) becomes true. Therefore the value of a will be assigned to the variable x. Hence x will be 20.

7) Bit-wise operator: Bit-wise operators are used for manipulation of data at bit level. This operator is used for testing the bits or shifting them right or left. Bit-wise operator is not applicable for float or double type data.

SL.	Operator	Meaning
1.	&	Bit-wise AND
2.	|	Bit-wise OR
3.	^	Bit-wise EXOR
4.	<<	Shift left
5.	>>	Shift right
6.	~	One's complement

Note: In the following two examples x, y, z, a, b, all are integer type variables and having 2 bytes of size considering 16-bit computer. That's why they are represented by 16 bits in binary.

Example1:

```
int x = 10, y = 6;
int z;
z = x | y;
w = x & y;
```

In the above case x is 10 in decimal i.e. 1010 in binary and y is 6 in decimal i.e. 0110 in binary.

```
   x: 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
   y: 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
   ___________________________________
z = x | y: 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 i.e. 14 in decimal

   x: 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
   y: 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
   ___________________________________
w = x & y: 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 i.e. 2 in decimal
```

Therefore the value of z and w will be 14 and 2 respectively.

Example2:

```
int a, b;
b = a^a;
```

In the above statements a and b both are declared as integer variables, but not assigned any value. Therefore a and b, both the variables will hold garbage values. Suppose the variable 'a' is holding a garbage value of 1011000001011111.

a: 1 0 1 1 0 0 0 0 0 1 0 1 1 1 1 1
a: 1 0 1 1 0 0 0 0 0 1 0 1 1 1 1 1

b = a ^ a: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 i.e. 0 in decimal

Therefore we can see that the variable a will hold a garbage value, but the variable b will hold 0 after the execution of the above two instructions.

Note: In the following two examples a and b are integer type variables and having 4 bytes of size considering 32-bit computer. That's why they are represented by 32 bits in binary.

Example3:

```
int a = 1073741826;
int b;
b = a << 3;
```

In this case the binary value of a will be shifted left for 3 times as shown below.

a : 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 (Left shifted once)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 (Left shifted twice)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 (Left shifted thrice)

b = a << 3: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 i.e. 16 in decimal

Example4:

```
int a = 1073741826;
int b;
b = a << 1;
```

In this case the binary value of a will be shifted left for 1 time as shown below.

a : 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 (Left shifted once)

b = a << 1: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

Here a and b are declared as integer or signed integer type variables. The MSB (most significant bit) is used for representing sign i.e. if MSB = 0 the number is positive and if MSB = 1 the number is negative in 2’s complement form. In this case the MSB of the variable b is 1 which is showing the number to be negative. We have to perform 2's complement on the value of b to get the negative value in decimal.

2's complement of b = 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 i.e. 2147483644 in decimal. Therefore after the execution of the above three statements the value will be – 2147483644.

If we consider the shift right operator (>>) the bits of the binary value will be shifted right, just the opposite case of shift left shown above.

8) Special operator: C supports special operators such as comma operator (,), sizeof operator, pointer operator (& and *) and member selection operator (. and ->). The discussion of pointer operator and member selection operator is out of scope here and will be later. In this section we will discuss only the comma operator and sizeof operator in details.

Comma operator: A comma operator is normally used to link two or more number of expression together. For example when multiple variables are declared in a single line they are separated with comma operator (,) like int a = 10, b = 20, c = 30, sum. In addition to this, a comma-linked expression is evaluated left to right and the value of the right-most expression is the value of the comma-linked expression. For example in the following statement the comma-linked expression placed right-hand side of the assignment operator is evaluated left to right.

```
int x, y;
Int value = (x = 10, y = 20, x + y);
```

In the above case the left-most expression x = 10 will be executed first, then the expression y = 20 will be executed and lastly the right-most expression (x + y) will be evaluated and assigned to the variable value. Hence the value of the variable will be 30 after the execution of the entire combined expression.

In for loops:

```
for(i = 1, i < 10, i++)      (left to right evaluated as it is comma separated)
```

sizeof operator: The sizeof is a compile time operator and when used with an operand, it returns the number of bytes an operand occupies in the memory. The operand may be a variable, a constant or a date qualifier.

Example:

```
#include<stdio.h>

int main()
{
        int value = 10;
        unsigned int sizeofvar;
        sizeofvar = sizeof(value);

        printf("The size of the variable = %u", sizeofvar);
        return 0;
}
```

Output

```
The size of the variable = 4
```

As the variable value is declared as integer type variable for 32-bit computer, the size of the variable will be 4 bytes. Therefore 4 will be printed on the screen to display the size of value.

The sizeof operator is normally used to determine the lengths of arrays and structures when their size are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of a program.

Precedence and Associativity of Operators

Precedence of operators come into picture when in an expression we need to decide which operator will be evaluated first. In this situation the operator with higher precedence or higher priority will be evaluated first.

For example – To evaluate the expression "2 + 3 * 5" we can determine the value of the expression in the following two ways.

1. (2 + 3) * 5 where (2 + 3) will be evaluated first and the result of (2 + 3) will be multiplied by 5 next. In this case 2 + 3 * 5 = 25.

2. 2 + (3 * 5) where (3 * 5) will be evaluated first and the result of (3 * 5) will be added with 2 next. In this case 2 + 3 * 5 = 17

But which one of the above two ways is correct in C language. In C multiplication (*) has higher priority than the addition (+). Therefore second way to evaluate the expression is correct as per the rules of precedence of operators in C language.

Associativity of operators come into the picture when precedence of operators become same and we need to decide which operator will be evaluated first.

For example – To evaluate the expression "10 / 2 * 5" we can determine the value of the expression in the following two ways.

1. *Left to Right evaluation:* (10 / 2) * 5 where (10 / 2) will be evaluated first and the result of (10 / 2) will be multiplied by 5 next. In this case 10 / 2 * 5 = 25.

2. *Right to Left evaluation:* 10 / (2 * 5) where (2 * 5) will be evaluated first and then 10 will be divided by the result of (2 * 5). In this case 10 / 2 * 5 = 1.

But which one of the above two ways is correct in C language. In C for same precedences we have to evaluate the expression from left to right for arithmetic operators. Therefore the first way to evaluate the expression i.e. left to right evaluation is correct in this case as per the rules of associativity of operators in C. The following table provides a complete list of operators, their precedence levels and their rules of association.

Precedence and Associativity Table

Rank	Category	Operators	Associativity
1	Parenthesis/ Brackets	( ) → Function call or parenthesis in expression [ ] → Array element reference -> → Member access operator . → Member access operator ++ → Postfix increment - - → Postfix decrement	Left to Right
2	Unary	+ → Unary plus - → Unary minus ++ → Prefix increment - - → Prefix decrement ! → Logical NOT / negation ~ → One's complement * → Pointer reference & → Address sizeof → Size of an object (type) → Type cast (conversion)	Right to Left
3	Multiplicative	* → Multiplication / → Division % → Modulo division	Left to Right
4	Additive	+ → Addition - → Subtraction	Left to Right
5	Bitwise shift	<< → Left shift >> → Right shift	Left to Right
6	Relational	< → Less than > → Greater than <= → Less than or equal to >= → Greater than or equal to	Left to Right
7	Equality	== → Equal to != → Not equal to	Left to Right
8	Bitwise AND	& → Bitwise AND	Left to Right
9	Bitwise XOR	^ → Bitwise XOR	Left to Right
10	Bitwise OR	| → Bitwise OR	Left to Right
11	Logical AND	&& → Logical AND	Left to Right
12	Logical OR	|| → Logical OR	Left to Right
13	Conditional	? : → Conditional operator	Right to Left

Rank	Category	Operators	Associativity
14	Assignment	= → Assignment operator += → Short-hand assignment operator with addition -= → Short-hand assignment operator with subtraction *= → Short-hand assignment operator with multiplication /= → Short-hand assignment operator with division %= → Short-hand assignment operator with modulo division &= → Short-hand assignment operator with bitwise AND ^= → Short-hand assignment operator with bitwise EXOR |= → Short-hand assignment operator with bitwise OR <<= → Short-hand assignment operator with shift left >>= → Short-hand assignment operator with shift right	Right to Left
15	Comma	, → Comma operator	Left to Right

The concept of precedence and associativity will be explained with the help of the following examples.

Example1: Evaluate the following expression using operator precedence and associativity.
*int Value = 2 * 3 / 4 + 4 / 4 + 8 – 2 + 5 / 8*

Steps	Evaluation of the expression	Explanation
1.	Value = 2 * 3 / 4 + 4 / 4 + 8 – 2 + 5 / 8	As multiplication *, division / have the higher priority than addition +, subtraction – , they will be evaluated before the addition and subtraction. Here multiple multiplications and divisions are present. Therefore they will be evaluated from left to right as per the rules of associativity.
2.	Value = 6 / 4 + 4 / 4 + 8 – 2 + 5 / 8	Left-most multiplication (2 * 3 = 6) is done.
3.	Value = 1 + 4 / 4 + 8 – 2 + 5 / 8	Division (6 / 4 = 1) is done.
4.	Value = 1 + 1 + 8 – 2 + 5 / 8	Division (4 / 4 = 1) is done.
5.	Value = 1 + 1 + 8 – 2 + 0	Division (5 / 8 = 0) is done.
6.	Value = 2 + 8 – 2 + 0	Multiple addition and subtraction are present. They will be evaluated from left to right as per the rules of associativity. So addition (1 + 1 = 2) is done.
7.	Value = 10 – 2 + 0	Addition (2 + 8 = 10) is done.
8.	Value = 8 + 0	Subtraction (10- 2 = 8) is done.
9.	Value = 8	Addition (8 + 0 = 8) is done.

Note: In the above example if the variable Value is declared as float, then the value of the expression will be 8, but the variable Value will be 8.000000, because 6 digits after the decimal point is permissible for a floating point value in C.

Example2: Evaluate the following expression using operator precedence and associativity.

*float a = 7 / 22 * (3.14 + 2) * 3 / 5*

Steps	Evaluation of the expression	Explanation
1.	a = 7 / 22 * (3.14 + 2) * 3 / 5	As parenthesis ( ) has the higher priority than multiplication * and division /, (3.14 + 2) will be evaluated first. Here multiple multiplications and divisions are present. Therefore they will be evaluated from left to right as per the rules of associativity.
2.	a = 7 / 22 * 5.140000 * 3 / 5	(3.14 + 2 = 5.140000) is done first.
3.	a = 0 * 5.140000 * 3 / 5	Left-most division (7 / 22 = 0) is done.
4.	a = 0.000000 * 3 / 5	Multiplication (0 * 5.140000 = 0.000000) is done.
5.	a = 0.000000 / 5	Multiplication (0.000000 * 3 = 0.000000) is done.
6.	a = 0.000000	Division (0.000000 / 5 = 0.000000) is done.

Example3: Evaluate the following expression using operator precedence and associativity.

int a = 4 + 2 % -8

Steps	Evaluation of the expression	Explanation
1.	a = 4 + 2 % -8	As modulo division % has the higher priority than addition + and division /, (2 % -8) will be evaluated first. Here dividend 2 is positive, so the result of the modulo division will be positive also.
2.	a = 4 + 2	(2 % -8 = 2) is done first.
3.	a = 6	Addition (4 + 2 = 6) is done.

Example4: Determine the value of the variable ans after the execution of the following expression using operator precedence and associativity.

*int a = 5, ans = ++a * (3 + 8) % 35 – 28 / 7;*

Steps	Evaluation of the expression	Explanation
1.	int a = 5, ans = ++a * (3 + 8) % 35 – 28 / 7;	The two expressions are comma separated and we know left to right evaluation of the expressions will be preformed for comma operator. Therefore the expression a = 5 will be executed first and the expression ans = ++a * (3 + 8) % 35 – 28 / 7 will be evaluated next.
2.	a = 5	The integer type variable a is assigned to 5
3.	ans = ++a * (3 + 8) % 35 – 28 / 7	In this expression the parenthesis ( ) has the highest priority, then prefix increment operator ++ has the next priority, then multiplication * and division /

		has the next priority and lastly subtraction has the lowest priority. The evaluation of the operators will be done from highest priority to lowest priority.
4.	ans = ++a * 11 % 35 – 28 / 7	(3 + 8 = 11) is done.
5.	ans = 6 * 11 % 35 – 28 / 7	During the execution of ++a the variable a is incremented by one from 5 to 6 and returns the value 6.
6.	ans = 66 % 35 – 28 / 7	Multiplication *, division / and modulo division % has the same precedence, hence will be evaluated left to right. That's why (6 * 11 = 66) is evaluated.
7.	ans = 31 – 28 / 7	(66 % 35 = 31) is evaluated.
8.	ans = 31 – 4	(28 / 7 = 4) is evaluated.
9.	ans = 27	(31 – 4 = 27) is executed.

Example5: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int ans;
        ans = 5, 10;

        printf("Value = %d\n", ans);
        return 0;
}
```

Explanation: In the above program the variable ans is declared as integer type variable. The statement ans = 5, 10; contains assignment operator = and comma operator ,. We know assignment operator has higher precedence over comma operator. Therefore the expression ans = 5 will be executed first and then 10 will be executed.

Output

```
Value = 5
```

Example6: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int ans;
        ans = (5, 10);
        printf("Value = %d\n", ans);
        return 0;
}
```

Explanation: In the above program the variable ans is declared as integer type variable. The statement ans = (5, 10); contains assignment operator = , parenthesis ( ) and comma operator ,. We know parenthesis operator has the highest precedence and will be evaluated first. Therefore the expression (5,10) will be evaluated left to right and returns the value 10. Now this value 10 will be assigned to the variable ans . Therefore the value of ans will be 10.

Output

Value = 10

Example7: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int ans;
        ans = (ans = 5, 10);
        printf("Value = %d\n", ans);
        return 0;
}
```

Explanation: The statement ans = (ans = 5, 10); contains assignment operator = , parenthesis ( ) and comma operator ,. We know parenthesis operator has the highest precedence and will be evaluated first. That's why (ans = 5, 10) will be executed where assignment operator will be executed making the value of ans to be 5 and then 10 will be executed. As a result the expression (ans = 5, 10) returns 10 which will be assigned to the variable ans again and thus the previous value of ans i.e. 5 is replaced by 10.

Output

Value = 10

Example8: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int a, b, c, d;
        a = 15, b = 10;
        c = ++a – b;
        d = b++ + a;

        printf("Value of c = %d\n Value of d = %d\n", c, d);
        return 0;
}
```

Explanation: In the expression (c = ++a – b) the ++a will be evaluated first and then the value of b will be subtracted from the incremented value of a, because ++ operator has higher precedence over – operator. After the execution of ++ a, the value of a becomes 16 from which the value of b i.e. 6 is subtracted. Thus the value of c becomes 6. Similarly in the next expression (d = b++ + a), b++ returns the current value 10 and then incremented to 11. Now 10 will be added to the value of a i.e. 16 to make the value of d to be 26.

Output
Value of c = 6 Value of d = 26

Example9: Evaluate the following expression using operator precedence and associativity and determine the value of the variable 'Value' after the execution of it.

int x = 20, y = 5, Value = x == 10 + 15 && y < 10;

Steps	Evaluation of the expression	Explanation
1.	int x = 20, y = 5, Value = x == 10 + 15 && y < 10;	The expressions are comma separated and will be evaluated left to right.
2.	x = 20, y = 5, Value = x == 10 + 15 && y < 10;	(x = 20) is executed and the value of x is 20.
3.	x = 20, y = 5, Value = x == 25 && y < 10;	(y = 5) is executed and the value of y is 5.
4.	x = 20, y = 5, Value = x == 25 && 1;	The expression Value = x == 25 && y < 10 contains assignment operator, equal to operator, less than operator and logical operator. Among these less than operator has the highest precedence. That's why the expression (y < 10) returns 1.
5.	x = 20, y = 5, Value = 0 && 1;	The relational expression (x == 25) returns 0.
6.	x = 20, y = 5, Value = 0;	The logical expression (0 && 1) returns 0 and assigned to the variable Value.

Example10: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i++ + i++));
        printf(“Value of i = %d\n”, i);
        i = 1;
        printf("Value of the expression = %d\n", (i++ - i++));
        printf(“Value of i = %d\n”, i);
        return 0;
}
```

Output of the Program
Value of the expression = 3
Value of i = 3
Value of the expression = -1
Value of i = 3

Explanation: In the above program the expression (i++ + i++) has post-increment operator (i++) and addition operator among which ++ operator has the higher precedence over + operator. As there are two post-increment operator, a question may arise which ++ operator will be executed first. According to the rules of associativity left to right evaluation will be performed. During the evaluation of left-most i++, the present value of i i.e. 1 will be returned first and then the value of i will be incremented to 2. Now during the execution of the second operator i++, the present value of i i.e. 2 will be returned and the i will be incremented to 3. Therefore the expression (i++ + i++) will return 3 i.e. (1 + 2).

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ + i++ = 1 + i++	Incremented value of i = 2
2.	Present value of i = 2 1 + i++ = 1 + 2 = 3	Incremented value of i = 3
	After the execution the expression (i++ + i++) will return 3 and the value of i will be 3.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ - i++ = 1 - i++	Incremented value of i = 2
2.	Present value of i = 2 1 - i++ = 1 - 2 = -1	Incremented value of i = 3
	After the execution the expression (i++ - i++) will return -1 and the value of i will be 3.	

Example11: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i++ + ++i));
        printf("Value of i = %d\n", i);
        i = 1;
        printf("Value of the expression = %d\n", (i++ - ++i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program
Value of the expression = 4
Value of i = 3
Value of the expression = -2
Value of i = 3

Explanation: In the above program the expression (i++ + ++i) has post-increment operator (i++), pre-increment operator (++i) and addition operator among which post-increment operator has the higher precedence over + operator and pre-increment operator. During the evaluation of left-most i++, the present value of i i.e. 1 will be returned first and then the value of i will be incremented to 2. Now during the execution of the second operator ++i, the i will be incremented to 3 first and then ++i will return 3. Therefore the expression (i++ + ++i) will return 4 i.e. (1 + 3).

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ + ++i = 1 + ++i	Incremented value of i = 2
2.	Present value of i = 2 1 + ++i = 1 + 3 = 4	Incremented value of i = 3
	After the execution the expression (i++ + ++i) will return 4 and the value of i will be 3.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ - ++i = 1 - ++i	Incremented value of i = 2
2.	Present value of i = 2 1 - ++i = 1 - 3 = -2	Incremented value of i = 3
	After the execution the expression (i++ - ++i) will return -2 and the value of i will be 3.	

Example12: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (++i + i++));
        printf("Value of i = %d\n", i);
        i = 1;
        printf("Value of the expression = %d\n", (++i - i++));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program
Value of the expression = 5
Value of i = 3
Value of the expression = 1
Value of i = 3

Explanation: In the above program the expression (++i + i++) has pre-increment operator (++i), post-increment operator (i++) and addition operator (+) among which post-increment operator has the higher precedence over + operator and pre-increment operator. During the evaluation of left-most ++i, the value of i will be incremented to 2, but ++i will not return 2 as i++ with higher precedence is placed next to ++i. Now i++ will be evaluated and during the execution of i++ the present value of i i.e. 2 will be returned and then i will be incremented to 3. As the value of i is 3, the ++i will return 3. Therefore the expression (++i + i++) will return 5 i.e. (3 + 2).

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i + i++ = ++i (value not returned) + i++	Incremented value of i = 2
2.	Present value of i = 2 ++i (value not returned) + i++ = ++i (value not returned) + 2	Incremented value of i = 3
3.	Present value of i = 3 ++i (value not returned) + 2 = 3 + 2 = 5	i = 3
	After the execution the expression (++i + i++) will return 5 and the value of i will be 3.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i - i++ = ++i (value not returned) - i++	Incremented value of i = 2
2.	Present value of i = 2 ++i (value not returned) - i++ = ++i (value not returned) - 2	Incremented value of i = 3
3.	Present value of i = 3 ++i (value not returned) - 2 = 3 - 2 = 1	i = 3
	After the execution the expression (++i - i++) will return 1 and the value of i will be 3.	

Example13: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (++i + ++i));
        printf("Value of i = %d\n", i);
        i = 1;
        printf("Value of the expression = %d\n", (++i - ++i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

Value of the expression = 6
Value of i = 3
Value of the expression = 0
Value of i = 3

Explanation: In the above program the expression (++i + ++i) has two pre-increment operators (++i) and addition operator (+) among which pre-increment operator has the higher precedence over + operator. Moreover two pre-increment operators (++i) have same precedence. Therefore right to left evaluation of pre-increment operators will be followed as per the rules of associativity. During the evaluation of left-most ++i, the value of i will be incremented to 2, but ++i will not return 2 as right-most preincrement operator should return first according to associativity. Now right-most ++i will be evaluated and during the execution of this ++i the present value of i is 2. Therefore ++i will increment the value of i from 2 to 3 and 3 will be returned by the right-most ++i. As the value of i is 3, the left-most ++i will also return 3. Therefore the expression (++i + ++i) will return 6 i.e. (3 + 3).

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i + ++i = ++i (value not returned) + ++i	Incremented value of i = 2
2.	Present value of i = 2 ++i (value not returned) + ++i = ++i (value not returned) + 3	Incremented value of i = 3
3.	Present value of i = 3 ++i (value not returned) + 3 = 3 + 3 = 6	i = 3
	After the execution the expression (++i + ++i) will return 6 and the value of i will be 3.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i - ++i = ++i (value not returned) - ++i	Incremented value of i = 2
2.	Present value of i = 2 ++i (value not returned) - ++i = ++i (value not returned) - 3	Incremented value of i = 3
3.	Present value of i = 3 ++i (value not returned) - 3 = 3 - 3 = 6	i = 3
	After the execution the expression (++i + ++i) will return 0 and the value of i will be 3.	

Example14: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i + i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i - i++));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

```
Value of the expression = 3
Value of i = 2
Value of the expression = 1
Value of i = 2
```

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i + i++ = i + 1	Incremented value of i = 2
2.	Present value of i = 2 i + 1 = 2 + 1 = 3	i = 2
	After the execution the expression (i + i++) will return 3 and the value of i will be 2.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i - i++ = i - 1	Incremented value of i = 2
2.	Present value of i = 2 i - 1 = 2 - 1 = 1	i = 2
	After the execution the expression (i - i++) will return 1 and the value of i will be 2.	

Example15: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i++ + i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ - i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

Value of the expression = 3
Value of i = 2
Value of the expression = -1
Value of i = 2

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ + i = 1 + i	Incremented value of i = 2
2.	Present value of i = 2 1 + i = 1 + 2 = 3	i = 2
	After the execution the expression (i++ + i) will return 3 and the value of i will be 2.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i++ - i = 1 - i	Incremented value of i = 2
2.	Present value of i = 2 1 - i = 1 - 2 = -1	i = 2
	After the execution the expression (i++ - i) will return -1 and the value of i will be 2.	

Example16: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i + ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i - ++i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

Value of the expression = 4
Value of i = 2
Value of the expression = 0
Value of i = 2

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i + ++i = i + 2	Incremented value of i = 2
2.	Present value of i = 2 i + 2 = 2 + 2 = 4	i = 2
	After the execution the expression (i + ++i) will return 4 and the value of i will be 2.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 i - ++i = i - 2	Incremented value of i = 2
2.	Present value of i = 2 i - 2 = 2 - 2 = 0	i = 2
	After the execution the expression (i - ++i) will return 0 and the value of i will be 2.	

Example17: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (++i + i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i - i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program
Value of the expression = 4
Value of i = 2
Value of the expression = 0
Value of i = 2

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i + i = 2 + i	Incremented value of i = 2
2.	Present value of i = 2 2 + i = 2 + 2 = 4	i = 2
	After the execution the expression (++i + i) will return 4 and the value of i will be 2.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 ++i - i = 2 - i	Incremented value of i = 2
2.	Present value of i = 2 2 - i = 2 - 2 = 0	i = 2
	After the execution the expression (++i - i) will return 0 and the value of i will be 2.	

Example18: Determine the output of the following program.

```
#include<stdio.h>
int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i++ + i++ + i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ + i++ - i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ - i++ + i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ - i++ - i++));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

```
Value of the expression = 6
Value of i = 4
Value of the expression = 0
Value of i = 4
Value of the expression = 2
Value of i = 4
Value of the expression = -4
Value of i = 4
```

Explanation: When no of ++ operator is three or more, the evaluation of the expression will be done from left to right. For this expression (i++ + i++ + i++) the left-most expression (i++ + i++) will be evaluated first, then the remaining i++ operand will be evaluated and finally added with the result of (i++ + i++) expression as shown in the following table step-by-step. The same procedure for the evaluation of the remaining expressions like (i++ + i++ - i++), (i++ - i++ + i++) and (i++ - i++ - i++) will be followed as shown in the next following tables.

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ + i++) + i++ = (1 + i++) + i++	Incremented value of i = 2
2.	Present value of i = 2 (1 + i++) + i++ = (1 + 2) + i++ = 3 + i++	Incremented value of i = 3
3.	Present value of i = 3 3 + i++ = 3 + 3 = 6	Incremented value of i = 4
	After the execution the expression (i++ + i++ + i++) will return 6 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ + i++) - i++ = (1 + i++) - i++	Incremented value of i = 2
2.	Present value of i = 2 (1 + i++) - i++ = (1 + 2) - i++ = 3 - i++	Incremented value of i = 3
3.	Present value of i = 3 3 - i++ = 3 - 3 = 0	Incremented value of i = 4
	After the execution the expression (i++ + i++ - i++) will return 0 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ - i++) + i++ = (1 - i++) + i++	Incremented value of i = 2
2.	Present value of i = 2 (1 - i++) + i++ = (1 - 2) + i++ = -1 + i++	Incremented value of i = 3
3.	Present value of i = 3 -1 + i++ = -1 + 3 = 2	Incremented value of i = 4
	After the execution the expression (i++ - i++ + i++) will return 2 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ - i++) - i++ = (1 - i++) - i++	Incremented value of i = 2
2.	Present value of i = 2 (1 - i++) - i++ = (1 - 2) - i++ = -1 - i++	Incremented value of i = 3
3.	Present value of i = 3 -1 - i++ = -1 - 3 = -4	Incremented value of i = 4
	After the execution the expression (i++ - i++ - i++) will return -4 and the value of i will be 4.	

Example19: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (i++ + i++ + ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ + i++ - ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (i++ - i++ + ++i));
        printf("Value of i = %d\n", i);


        i = 1;
        printf("Value of the expression = %d\n", (i++ - i++ - ++i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

```
Value of the expression = 7
Value of i = 4
Value of the expression = -1
Value of i = 4
Value of the expression = 3
Value of i = 4
Value of the expression = -5
Value of i = 4
```

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ + i++) + ++i = (1 + i++) + ++i	Incremented value of i = 2
2.	Present value of i = 2 (1 + i++) + ++i = (1 + 2) + ++i = 3 + ++i	Incremented value of i = 3
3.	Present value of i = 3 3 + ++i = 3 + 4 = 7	Incremented value of i = 4
	After the execution the expression (i++ + i++ + ++i) will return 7 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ + i++) - ++i = (1 + i++) - ++i	Incremented value of i = 2
2.	Present value of i = 2 (1 + i++) - ++i = (1 + 2) - ++i = 3 - ++i	Incremented value of i = 3
3.	Present value of i = 3 3 - ++i = 3 - 4 = -1	Incremented value of i = 4
	After the execution the expression (i++ + i++ - ++i) will return -1 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ - i++) + ++i = (1 - i++) + ++i	Incremented value of i = 2
2.	Present value of i = 2 (1 - i++) + ++i = (1 - 2) + ++i = -1 + ++i	Incremented value of i = 3
3.	Present value of i = 3 -1 + ++i = -1 + 4 = 3	Incremented value of i = 4
	After the execution the expression (i++ - i++ + ++i) will return 3 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (i++ - i++) - ++i = (1 - i++) - ++i	Incremented value of i = 2
2.	Present value of i = 2 (1 - i++) - ++i = (1 - 2) - ++i = -1 - ++i	Incremented value of i = 3
3.	Present value of i = 3 -1 - ++i = -1 - 4 = -5	Incremented value of i = 4
	After the execution the expression (i++ - i++ - ++i) will return -5 and the value of i will be 4.	

Example20: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (++i + ++i + i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i + ++i - i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i - ++i + i++));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i - ++i - i++));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

```
Value of the expression = 9
Value of i = 4
Value of the expression = 3
Value of i = 4
Value of the expression = 3
Value of i = 4
Value of the expression = -3
Value of i = 4
```

Explanation:

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (++i + ++i) + i++ = (++i + ++i) + i++	Incremented value of i = 2
2.	Present value of i = 2 (i++ + ++i) + i++ = (i++ + 3) + i++ = (3 + 3) + i++ = 6 + i++	Incremented value of i = 3
3.	Present value of i = 3 6 + i++ = 6 + 3 = 9	Incremented value of i = 4
	After the execution the expression (++i + ++i + i++) will return 9 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (++i + ++i) - i++ = (++i + ++i) - i++	Incremented value of i = 2
2.	Present value of i = 2 (i++ + ++i) - i++ = (i++ + 3) - i++ = (3 + 3) - i++ = 6 - i++	Incremented value of i = 3
3.	Present value of i = 3 6 - i++ = 6 - 3 = 3	Incremented value of i = 4
	After the execution the expression (++i + ++i - i++) will return 3 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (++i - ++i) + i++ = (++i - ++i) + i++	Incremented value of i = 2
2.	Present value of i = 2 (i++ - ++i) + i++ = (i++ - 3) + i++ = (3 - 3) + i++ = 0 + i++	Incremented value of i = 3
3.	Present value of i = 3 0 + i++ = 0 + 3 = 3	Incremented value of i = 4
	After the execution the expression (++i - ++i + i++) will return 3 and the value of i will be 4.	

Steps	Evaluation of the expression	Value of i
1.	Present value of i = 1 (++i - ++i) - i++ = (++i - ++i) - i++	Incremented value of i = 2
2.	Present value of i = 2 (i++ - ++i) - i++ = (i++ - 3) - i++ = (3 - 3) - i++ = 0 - i++	Incremented value of i = 3
3.	Present value of i = 3 0 - i++ = 0 - 3 = -3	Incremented value of i = 4
	After the execution the expression (++i - ++i + i++) will return -3 and the value of i will be 4.	

Example21: Determine the output of the following program.

```
#include<stdio.h>

int main()
{
        int i;
        i = 1;
        printf("Value of the expression = %d\n", (++i + i++ + ++i + ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i - i++ - ++i + ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i + i++ - ++i - ++i));
        printf("Value of i = %d\n", i);

        i = 1;
        printf("Value of the expression = %d\n", (++i - i++ - ++i - ++i));
        printf("Value of i = %d\n", i);
        return 0;
}
```

Output of the Program

```
Value of the expression = 14
Value of i = 5
Value of the expression = 2
Value of i = 5
Value of the expression = -4
Value of i = 5
Value of the expression = -8
Value of i = 5
```

Explanation: The reason behind the output of the above program is self-explanatory as per the rules of Example18, Example19 and Example20.

Example22: Determine the output of the following program.

```
#include<stdio.h>
int main()
{
        int x, y;
        x = 1;
        y = 2;
        printf("Value of the expression = %d\n", (x + y + x++ + y++ + ++x + ++y));
        printf("Value of x = %d\n", x);
        printf("Value of y = %d\n", y);

        x = 1;
        y = 2;
        printf("Value of the expression = %d\n", (x++ + y++ + x + y + ++x + ++y));
        printf("Value of x = %d\n", x);
        printf("Value of y = %d\n", y);

        x = 1;
        y = 2;
        printf("Value of the expression = %d\n", (++x + ++y + x++ + y++ + x + y));
        printf("Value of x = %d\n", x);
        printf("Value of y = %d\n", y);

        x = 1;
        y = 2;
        printf("Value of the expression = %d\n", (x + ++x + x++ + y + ++y + y++));
        printf("Value of x = %d\n", x);
        printf("Value of y = %d\n", y);
        return 0;
}
```

Output of the Program

```
Value of the expression = 13
Value of x = 3
Value of x = 4
Value of the expression = 15
Value of x = 3
Value of x = 4
Value of the expression = 17
Value of x = 3
Value of x = 4
Value of the expression = 14
Value of x = 3
Value of x = 4
```

Explanation:

Steps	Evaluation of the expression	Value of x and y
1.	Present value of x = 1 Present value of y = 2 (x + y) + x++ + y++ + ++x + ++y = (1 + 2) + x++ + y++ + ++x + ++y = 3 + x++ + y++ + ++x + ++y	value of x = 1 value of y = 2
2.	Present value of x = 1 Present value of y = 2 (3 + x++) + y++ + ++x + ++y = (3 + 1) + y++ + ++x + ++y = 4 + y++ + ++x + ++y	Incremented value of x = 2 value of y = 2
3.	Present value of x = 2 Present value of y = 2 (4 + y++) + ++x + ++y = (4 + 2) + ++x + ++y = 6 + ++x + ++y	value of x = 2 Incremented value of y = 3
4.	Present value of x = 2 Present value of y = 3 (6 + ++x) + ++y = (6 + 3) + ++y = 9 + ++y	Incremented value of x = 3 value of y = 3
5.	Present value of x = 3 Present value of y = 3 9 + ++y = 9 + 4 = 13	value of x = 3 Incremented value of y = 4
	After the execution the expression *(x + y + x++ + y++ + ++x + ++y)* will return 13 and the values of x and y will be 3 and 4 respectively.	

Steps	Evaluation of the expression	Value of x and y
1.	Present value of x = 1 Present value of y = 2 *(x++ + y++) + x + y + ++x + ++y* = (1 + 2) + x + y + ++x + ++y = 3 + x + y + ++x + ++y	Incremented value of x = 2 Incremented value of y = 3
2.	Present value of x = 2 Present value of y = 3 (3 + x) + y + ++x + ++y = (3 + 2) + y + ++x + ++y = 5 + y + ++x + ++y	value of x = 2 value of y = 3
3.	Present value of x = 2 Present value of y = 3 (5 + y) + ++x + ++y = (5 + 3) + ++x + ++y = 8 + ++x + ++y	value of x = 2 value of y = 3
4.	Present value of x = 2 Present value of y = 3 (8 + ++x) + ++y = (8 + 3) + ++y = 11 + ++y	Incremented value of x = 3 value of y = 3
5.	Present value of x = 3 Present value of y = 3 11 + ++y = 11 + 4 = 15	value of x = 3 Incremented value of y = 4
	After the execution the expression *(x++ + y++ + x + y + ++x + ++y)* will return 15 and the values of x and y will be 3 and 4 respectively.	

Steps	Evaluation of the expression	Value of x and y
1.	Present value of x = 1 Present value of y = 2 *(++x + ++y) + x++ + y++ + x + y* = (2 + 3) + x++ + y++ + x + y = 5 + x++ + y++ + x + y	Incremented value of x = 2 Incremented value of y = 3
2.	Present value of x = 2 Present value of y = 3 (5 + x++) + y++ + x + y = (5 + 2) + y++ + x + y = 7 + y++ + x + y	Incremented value of x = 3 value of y = 3
3.	Present value of x = 3 Present value of y = 3 (7 + y++) + x + y = (7 + 3) + x + y = 10 + x + y	value of x = 3 Incremented value of y = 4
4.	Present value of x = 3 Present value of y = 4 (10 + x) + y = (10 + 3) + y = 13 + y	value of x = 3 value of y = 4
5.	Present value of x = 3 Present value of y = 4 13 + y = 13 + 4 = 17	value of x = 3 value of y = 4
	After the execution the expression *(++x + ++y + x++ + y++ + x + y)* will return 17 and the values of x and y will be 3 and 4 respectively.	

Steps	Evaluation of the expression	Value of x and y
1.	Present value of x = 1 Present value of y = 2 *(x + ++x) + x++ + y + ++y + y++* = (x + 2) + x++ + y + ++y + y++ = (2 + 2) + x++ + y + ++y + y++ = 4 + x++ + y + ++y + y++	Incremented value of x = 2 value of y = 2
2.	Present value of x = 2 Present value of y = 2 (4 + x++) + y + ++y + y++ = (4 + 2) + y + ++y + y++ = 6 + y + ++y + y++	Incremented value of x = 3 value of y = 2
3.	Present value of x = 3 Present value of y = 2 (6 + y) + ++y + y++ = (6 + 2) + ++y + y++ = 8 + ++y + y++	value of x = 3 value of y = 2
4.	Present value of x = 3 Present value of y = 2 (8 + ++y) + y++ = (8 + 3) + y++ = 11 + y++	value of x = 3 Incremented value of y = 3
5.	Present value of x = 3 Present value of y = 3 11 + y++ = 11 + 3 = 14	value of x = 3 Incremented value of y = 4
	After the execution the expression *(x + ++x + x++ + y + ++y + y++)* will return 14 and the values of x and y will be 3 and 4 respectively.	

Note: The procedure of evaluation of expressions for Example10 to Example21 is followed by GCC compiler. Therefore we can observe that the execution of an expression will be started from left to right always for GCC compiler, but "when the value will be returned ?" depends on the rules of operator precedence and associativity. It is important to mention that other compilers may follow other procedure to evaluate an expression and may give different results for Example10 to Example21.

Example23: Determine the output of the following program.

```
#include<stdio.h>
int main()
{
        int x;
        x = 10;
        x = x++;
        printf("Value of x = %d\n", x);

        x = 10;
        x = ++x;
        printf("Value of x = %d\n", x);
        return 0;
}
```

Output of the Program
Value of x = 10
Value of x = 11

Explanation: For the first expression x = x++, the post-increment operator has higher precedence than assignment operator. Therefore x++ will return 10 first and then the value of x becomes 11 after increment. Now 10 will be assigned to x using '=' operator. Therefore the previous value of x i.e. 11 will be replaced by 10. Finally the value of x will be 10.

For the second expression x = ++x, the pre-increment operator has higher precedence than assignment operator. Therefore ++x will increment the value of x to 11 first and then the incremented value 11 will be returned by ++x. Now 11 will be assigned to x using '=' operator. Finally the value of x will be 11.

Example24: Determine the output of the following program.

```
#include<stdio.h>
int main()
{
        int x, y;
        x = 1;
        y = (++x) + (x = 10);
        printf("Value of y = %d\n", y);

        x = 1;
        y = (x++) + (x = 10);
        printf("Value of y = %d\n", y);

        x = 1;
        y = (++x) + (x = 10) + (x = 20);
        printf("Value of y = %d\n", y);

        x = 1;
        y = (x++) + (x = 10) + (x = 20);
        printf("Value of y = %d\n", y);
        return 0;
}
```

Output of the Program
Value of y = 20
Value of y = 11
Value of y = 40
Value of y = 31

Explanation:

a) For y = (++x) + (x = 10) : In this expression two pair of parenthesis has highest priority but same associativity. Hence left parenthesis (++x) will increment the value of x to 2, but the value of x will not be returned until (x = 10) will be evaluated. The expression (x = 10) will return 10 making the value of x to be 10. As the value of x is 10, (++x) will return 10 also. Finally the value of y will be (10 + 10) i.e. 20.

b) For y = (x++) + (x = 10) : In this expression two pair of parenthesis has highest priority but same associativity. Hence left parenthesis (x++) will return 1 first, then increment the value of x to 2. Now the expression (x = 10) will return 10 making the value of x to be 10. Finally the value of y will be (1 + 10) i.e. 21.

c) For y = (++x) + (x = 10) + (x = 20) :
((++x) + (x = 10)) + (x = 20)
= ((++x) + 10) + (x = 20)
= (10 + 10) + (x = 20)
= 20 + (x = 20)
= 20 + 20
= 40

d) For y = (x++) + (x = 10) + (x = 20) :
((x++) + (x = 10)) + (x = 20)
= (1 + (x = 10)) + (x = 20)
= (1 + 10) + (x = 20)
= 11 + (x = 20)
= 11 + 20
= 31

Example24: Determine the output of the following program.

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

Output of the Program

Values of a b c d: 1 0 1 1

Explanation:

Steps	Evaluation of the expression	Value of x and y
1.	Present value of a = 1 Present value of b = 1 c = ++a || b++ c = 2 || b++ (Not executed) c = 1 || b++ c = 1 (TRUE)	Incremented value of a = 2 As ++a expression has already returned 2 (TRUE), || operator does not require the value of b++. Whatever may be the value of b++, the expression ++a || b++ will return 1. Therefore b++ will not be executed here and the value of b = 1
2.	Present value of a = 2 Present value of b = 1 d = b-- && --a d = 1 && 1 d = 1	decremented value of a = 1 decremented value of b = 0
	After the execution the program the following values of the variables a, b, c, d will be displayed. a = 1 b = 0 c = 1 d = 1	

Declaring a variable as global and local

When a variable is declared before the main function, it is called a global variable. The name ‘global’ suggests that it can be used in all functions in the program without declaring it inside any function. A global variable is also known as external variable.

When a variable is declared inside any function, then the variable can be accessed only inside that function where it has been declared and its value can not be used in other functions. This type of variable is called local variable.

An example program is given below to explain this.

```
#include<stdio.h>

int gvar = 10;

void function1()
{
        printf(“The value of the global variable \” gvar \” is accessed in function1.\n”);
        printf(“ The value of gvar: %d”,gvar);
}

void function2()
{
        printf(“The value of the global variable \” gvar \” is accessed in function2.\n”);
        printf(“ The value of gvar: %d”,gvar);
}

int main()
{
        int var1 = 20;
        int sum;
        sum = gvar + var1;
         printf(“Sum with global variable = %d”, sum);

        function1();
        function2();
        return 0;
}
```

Output on the screen

Sum with global variable = 30

The value of the global variable gvar is accessed in function1.
The value of gvar: 10

The value of the global variable gvar is accessed in function2.
The value of gvar: 10

Explanation: In this program the variable “gvar” is declared before all the functions namely function1(), function2() and main(). So the variable “gvar” is a global variable in this case. As it is a global variable, it can be accessed in all the functions without declaring it inside any function. Hence the value of gvar is used and printed in function1(), function2() and main() respectively. On the other hand the variable var1 is declared inside the function main(). So its value can only be used in the function main() not from other functions namely function1() and function2(). Here the variable var1 is an example of local variable.

Defining Symbolic Constants

We often use certain unique constants in a program and this constant may be used repeatedly in different places in the program. We face two problems in this case if we want to modify the value of that particular constant.

1. Problem in modification of the constant value
2. Problem in understanding the constant value in the program

Modifiability: If we want to change the value of the constant we have to search throughout the program and explicitly change the value of the constant wherever it has been used. If the constant is used 10 times in the program, we have to change the value of the variable 10 times in 10 places. This is very much time consuming and prone to error also because if any value is left unchanged by mistake, the program will give a wrong result.

Understandability: when a constant value appears in a program, its use is not always clear. Assignment of such constants to a *symbolic name* gives us better understanding. For example if we use a constant value 3.14 to different places in a program to represent the value of “pi”, it will not be so clear. Therefore it is better to give a symbolic name *PI* to the constant value 3.14 and use that name PI in place of the value 3.14 wherever required in a program.

A constant can be defined by symbolic name using the following general form.

#define symbolic-name value of constant

For example – **#define PI 3.14**

Rules for constructing symbolic constants

1. Symbolic names have the same form as variable names. Symbolic names are written in CAPITAL letters normally to visually distinguish from the normal variables which are generally written in lowercase letters in C language.

2. No blank space is allowed between the pound/ hash sign (#) and the word define.

3. '#' must be the first character in the line consisting the word 'define'.

4. A blank space must be placed between #define and symbolic name and between symbolic name and the constant.

5. #define statement should not be terminated with semicolon (;).

6. After defining a symbolic constant, it can not be assigned any other value by using an assignment statement within the program.

7. Symbolic names are not declared for data types.

Examples of some invalid #define statements are given below for clarification.

#define statements	Remarks	Reasons
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No blank space between # and define is allowed
#define N 25;	Invalid	';' sign is not allowed
#define N 5, M 10	Invalid	A #define statement can only contain one symbolic name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in symbolic name

Chapter 3

Array

Suppose we have to store the marks of 100 students in an institution. If we want to do this job using variables in C language, we need to declare 100 separate variables with different names, which is a tedious and inefficient way. To get rid off this problem, C language offers a data structure known as array. Using the array we can store plenty of values in a well organized manner in the memory of the computer. Moreover any value stored inside an array may be accessed and modified very easily. In a word it can be said that an array is a collection of variables of same type. Therefore the concept of array plays a vital role in C language. The definition of an array is given as follows.

An array is a static linear data structure where the elements of same data type are stored in consecutive memory locations and each elements are accessed by an index.

Declaration of an array - In C, arrays are declared using the following syntax or more precisely it can be said that one dimensional array is declared using the following syntax.

data_type ArrayName[size];

In the above declaration *data_type* denotes the type of the values that can be stored in the array, *ArrayName* indicates the name of the array which is assigned by the programmer and the *size* enclosed with square brackets denotes the fixed or static size of the array which can not be changed throughout the program.

For example: int marks[100];

The above statement declares an array named marks that contains 100 elements which are stored in 100 consecutive memory locations as shown in Fig.3.1. Here the elements of the array contains the marks of different students.

1D Array 'marks'	marks[0] 75	marks[1] 86	marks[2] 91	……..	Marks[98] 85	Marks[99] 67
index→	0	1	2	……..	98	99
Starting Address→	1000	1004	1008	……..	1392	1396

Fig.3.1: Array stores the marks of 100 students

In Fig.3.1 it is being observed that every element of the array has its predecessor and successor except 1st element and 100th (last) element. That's why array is a linear data structure.

Understanding the definition of array – Here the significance of the important terms in the definition of an array will be explained in details.

I) *Data structure:* Data structure is a format of organizing and storing multiple data in an efficient way so that every data can be accessed easily. Array also stores plenty of data in some consecutive memory locations which gives the flexibility to read and modify those data very easily with the help of the indices of that particular array. That's why array is called data structure.

II) *Static data structure:* Why array is called static data structure is illustrated here. If the format of the declaration of an array is noticed, it will be seen that the size of the array enclosed by square brackets is made constant at the very beginning of any program in C. This size is basically the maximum size of the array. If an array is declared as "int a[50]", it can store maximum 50 integer values. Hence if we are going to store 51st value inside this array, we can not store it. Moreover this size can not be altered anywhere after this declaration in C. That's why the array data structure is called static. This creates a serious problem where the number of values to be stored in an array can not be known in advance. In this situation the maximum size of the array is taken very large in the declaration to protect the array to be exhausted. On the contrary due to the declaration of the array with large size plenty of array space may not be utilized.

III) *Linear data structure:* In case of an array every element has one predecessor and one successor element except the first and the last element of the array. In addition to this an array is organized in a sequential manner. Therefore an array is a linear data structure.

IV) *Elements of same data type:* All the elements or the values inside an array must have same data type either integer or floating point or character type. An array can not accommodate different types of data. For example it is not possible to store integer values and floating point values in the same array. Due to this reason the entire array is declared by a data type initially, which gives the information regarding the type of values to be stored inside the array.

V) *Consecutive memory locations:* The elements of an array are stored in successive memory locations. If an array 'a' is declared as 'int' in a program, each element will take 4 bytes of memory for 32-bit architecture and here 1st, 2nd, 3rd and so on elements will be stored with the starting addresses 1000, 1004, 1008 and so on respectively provided the starting address or the base address of the array is 1000. The memory organization of an array declared as 'int' is shown below in Fig.3.2 for clarification.

1D Array 'a' of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.2: Memory representation of an integer type array with five elements

1) One dimensional array or 1D array – 1D array is an array where the elements are arranged sequentially in one direction. In this case only one row exists and all the elements are positioned one after another row-wise. The elements of a 1D array are accessed by one index only such as - 1st element of a 1D array is accessed by a[0], 2nd element by a[1], 3rd element by a[2] and so on where 0, 1, 2 enclosed by square brackets are the indices. In the above we have discussed one dimensional array mainly to demonstrate the definition of an array. A 1D array is shown in Fig.3.3 along with its memory representation.

1D Array 'a' of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.3: 1D array of five elements along with index values and memory representation

Declaration of 1D array - A one dimensional array is declared as follows in C.

data_type ArrayName[size];

- data type—the kind of values it can store, for example, int , char , float , double .
- ArrayName—to identify the array.
- size—the maximum number of values that the array can hold.

Accessing the elements of 1D array – To access a particular element of a 1D array we have to write the array name followed by the index of the element enclosed within a square bracket like ArrayName[index]. In case of array in C language the index starts from 0 and ends to (size – 1). That means, the index of first element will be 0 and the index of the last element will be (size – 1).

If we want to access 1st element of an array named 'a', we should use a[0]. Similarly 2nd, 3rd, 4th, 5th element can be accessed by a[1], a[2], a[3], a[4] respectively.

Initialization of 1D array – There are four methods by which a 1D array can be initialized to some values.

Method 1: In this method the 1D array is initialized using the following syntax either during the declaration or after the declaration as given below.

```
int a[5] = {10, 20, 30, 40, 50};        [During array declaration]

            OR

int a[5];
a[5] = {10, 20, 30, 40, 50};            [After array declaration]
```

In the above statement, the 1st element is initialized to 10, 2nd element is initialized with 20 and so on. Therefore we can say a[0] = 10, a[1] = 20, a[2] = 30, a[3] = 40 and a[4] = 50.

Method 2: In this method the 1D array is initialized with some values during the declaration as follows.

```
int a[] = {10, 20, 30, 40, 50};          [During array declaration]
```

Like Method 1 the 1st element is initialized to 10, 2nd element is initialized with 20 and so on i.e. a[0] = 10, a[1] = 20, a[2] = 30, a[3] = 40 and a[4] = 50. But Method 2 is better than Method 1, because in Method 2 the size of the 1D array is not specified into the square brackets. Here the maximum size of the 1D array becomes equal to the number of elements initialized in the array declaration. For example – the size of the array is automatically fixed to 5 by the C compiler in the above statement. It gives the opportunity to the programmer to add as many elements as he wish to add into the 1D array.

Method 3: In this technique we can initialize the elements of the array individually with the help of the index as mentioned below.

```
int a[5];

a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 50;
```

But the above mentioned procedure is not efficient, because here every elements are being initialized separately using the assignment operator. Hence if 100 elements of an array are to be initialized using this Method 3, we have to use 100 assignments for 100 elements one by one which is a cumbersome task. Therefore this method is not accepted for array initialization.

Method 4: Here the initialization of the elements are done using a loop as given below.

```
int a[5], i;
int j = 10;

for(i=0; i<5; i++)
{
        a[i] = j;
        j = j + 10;
}
```

Among the above mentioned four methods Method 1 and Method 3 are not used due their limitations. Therefore Method 2 and Method 4 are generally used in various C programs.

- *What happens if the number of elements initialized in an array is less than the size of the array ?*

Solution: Suppose three elements of an array 'a' is initialized to some values where the size of the array is declared as 5 using the following statement.

```
int a[5] = {10, 20, 30};
```

Here it can be seen that only 3 elements (1st, 2nd and 3rd element) of the array are initialized with 10, 20 and 30 respectively instead of 5 elements. Now a question comes obviously that what will be the values of 4th and 5th element after this initialization. In this situation the remaining elements will be automatically filled by zero. Therefore 4th element (a[3]) and 5th element (a[4]) will become zero.

- *How are all the elements of a 1D array of size 100 initialized with zero ?*

Solution: To initialize a 1D array of size 100 with the value zero we have to follow the method given below.

```
int a[100] = {0};
```

When the above statement is complied the 1st element i.e. a[0] is made zero and the rest of the all 99 elements become zero automatically by the compiler of C. In this way it is possible to make all the elements of a 1D array to be zero very easily.

Note:
1. int a[10] = { }; is illegal, because we have to specify atleast one value inside the curly braces.
2. int a[5] = {10, 20, 30, 40, 50, 60}; is also illegal, because the number of elements initialized is larger than the size of the array.

Designated Initialization of 1D array – If we want to initialize some elements to be initialized with some non-zero values randomly (not sequentially) and other remaining elements to be zero, then it is better to use designated initialization of an array. For example 1st, 3rd and 7th elements are to be initialized with 45, 36 and 21 respectively and other elements like 2nd, 4th, 5th, 6th, 8th, 9th, 10th elements should be zero. In this case we have to use the following statement to accomplish this.

```
int a[10] = {[0] = 45, [2] = 36, [6] = 21};
```

In the above syntax the values inside the square brackets indicates the indices of 1st, 3rd and 7th element of the 1D array. This way of initialization is called designated initialization and each number in the square bracket is said to be a designator.

- *An array without any specified size is initialized using the statement "int a[ ] = {[0] = 45, [2] = 36, [6] = 21}". What will be the size of the array ?*

  Solution: It is being observed that the size of the array 'a' is not mentioned here and all the initialization are done designated initialization. Here the compiler will deduce the size of the 1D array from the largest designator in the list. Therefore the size of the array will be 7 in the above case, as the largest designator here is [6].

We can mix up the designated initialization and the normal initialization as follows.

```
int a[ ] = {1, 7, 5, [5] = 90, 6, [8] = 4};
```

The above statement is equivalent to the following initialization.

```
int a[ ] = (1, 7, 5, 0, 0, 90, 6, 0, 4};
```

The above two initialization implies that a[0] = 1, a[1] = 7, a[2] = 5, a[3] = 0, a[4] = 0, a[5] = 90, a[6] = 6, a[7] = 0, a[8] = 4.

- *Consider the initialization "int a[ ] = {1, 2, 3, [2] = 4, [6] = 45};" for a 1D array. What will be the value of third element of the array after this initialization ?*

Solution: Here the important point is that, there is a coincidence between normal initialization and designated initialization. Consider the above mentioned initialization where both the normal and designated initialization are used at a time.

```
int a[ ] = {1, 2, 3, [2] = 4, [6] = 45};
```

In the above case the third element (a[2]) is made 3 using normal initialization and at the same time it is assigned with a value of 4 using designated initialization. Naturally the question arises, which value will be initialized for the 3rd element. Here designated initialization will get the priority. That means, the value of the 3rd element will become 4 instead of 3. Therefore the equivalent statement for the above initialization will be given as:

```
int a[ ] = {1, 2, 4, 0, 0, 0, 45};
```

Advantages of designated initialization:

- No need to bother about the entries containing zero.

  ```
  int a[10] = {45, 0, 36, 0, 0, 0, 21, 0, 0, 0};int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  ```

  The initialization can be performed in simpler way using designated initialization as follows.

  ```
  int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  ```

- No need to bother about the order at all.

  ```
  int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  int a[10] = {[2] = 36, [6] = 21, [0] = 45};
  ```

  The above two statements give the same result.

Memory representation of 1D array – We know that the elements are placed consecutively into the memory in case of a 1D array. If a 1D array is declared as int type, each element of the array will occupy 4 bytes in the memory space for a 32-bit architecture. If the starting address or the base address of the 1D array is 1000, the first element of the array will be stored from the address 1000 to 1003, the second element will be stored from 1004 to 1007, third element will occupy from the address 1008 to 1011 etc. Hence it is clear that the starting addresses of the consecutive elements in a 1D array will be incremented by 4 for a 1D array declared as int. Similarly the starting addresses of the successive elements will be incremented by 1 if the 1D array is declared as char. It implies that how much the addresses of the consecutive elements will be increased in a 1D array, that depends on the data type of the 1D array. A pictorial representation of the memory organization of an integer type 1D array as well as char type 1D array has been depicted in Fig.3.4(a) and Fig.3.4(b) respectively for better understanding of the above mentioned phenomenon.

1D Array a of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.4(a): Memory organization of int type 1D array of size = 5 and base address = 1000

1D Array b of char type	b[0]	b[1]	b[2]	b[3]	b[4]
index→	0	1	2	3	4
Starting Address→	2000	2004	2008	2012	2016

Fig.3.4(b): Memory organization of char type 1D array of size = 5 and base address = 2000

Now we can determine the starting address of any element of a 1D array ‘a’ using the following formula and this formula can verified for the 1D array ‘a’ and ‘b’ in Fig.3.4.

Starting Address of $a[i] = B + (i - i_0) \times W$ Where B = Base address of 1D array
W = Storage size of each element in the array in bytes
i = Index of the element in the array
i_0 = Starting index of the 1D array

Now the storage size of every element in an array (W) depends on the date type of the array. Different values of W depending on the data type of the array are given below.

W = 4 bytes for int data type
W = 8 bytes for long int data type
W = 4 bytes for float data type
W = 8 bytes for double data type
W = 1 byte for char data type

In C language the index of a 1D array starts from 0 always. That’s why $i_0 = 0$ for the above mentioned formula in C.

2) Two dimensional array or 2D array – 2D array is an array where the elements are arranged sequentially in two directions – along the row and along the column. The elements of a 2D array is placed like a 2D matrix. The elements of a 2D array are accessed by two indices, one index gives the row-wise position and another index gives the column-wise position. Hence any element of a 2D array may be demoted by a[i][j] where i is the row index and j is the column index. The pictorial view of a 2D array with 5 rows and 6 columns is shown in Fig.3.5.

j ↓ i→	0	1	2	3	4	5
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]	a[3][5]
4	a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]	a[4][5]

Fig.3.5: 2D array with 5 rows and 6 columns

In Fig.3.5 we can see that every elements of the 2D array is represented by the array notation in C language. For example – a[0][0] is the element with 1st row and 1st column, a[4][2] is the element with 5th row and 3rd column. The 1st index inside the square bracket denotes the row of the array whereas 2nd index denotes the column of the array. Therefore in general a[i][j] denotes (i + 1)th row and (j + 1)th column of the 2D array. Therefore a 2D array with m no. of rows and n no. of columns can hold maximum (m × n) no. of elements.

Declaration of 2D array - A two dimensional array is declared as follows in C.

data_type ArrayName[SizeOfRow][SizeOfColumn];

- data type – the kind of values it can store, for example, int , char , float , double .
- ArrayName – to identify the array.
- SizeOfRow – the maximum number of rows that the array can hold.
- SizeOfColumn – the maximum number of columns that the array can hold.

For example, an integer type 2D array with 4 rows and 5 columns is declared as follows.

```
int a[4][5];
```

Accessing the elements of 2D array – To access a particular element of a 2D array we have to write the array name followed by the row index of the element enclosed within a square bracket and the column index of the element enclosed with another square bracket like ArrayName[i][j] where i and j are the row index and the column index of the particular element of the array. If we want to access 1st element of the 2D array named ‘a’, we should use a[0][0] where both zeros indicates the row index and the column index of the array.

Initialization of 2D array – There are two methods by which a 2D array can be initialized to some values.

Method 1: In this method a 2D array is initialized as follows.

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
```

In the above case the number of rows is 2 and the number of columns is 3. Therefore total number of elements present in the 2D array is (2 × 3) = 6. Now the first three values will be assigned to the elements of the first row sequentially which results a[0][0] = 1, a[0][1] = 2, a[0][2] = 3. Similarly the values 4, 5 and 6 will be stored into the elements a[1][0], a[1][1] and a[1][2] respectively. The pictorial view of the 2D array is shown in Fig.3.6 after the initialization.

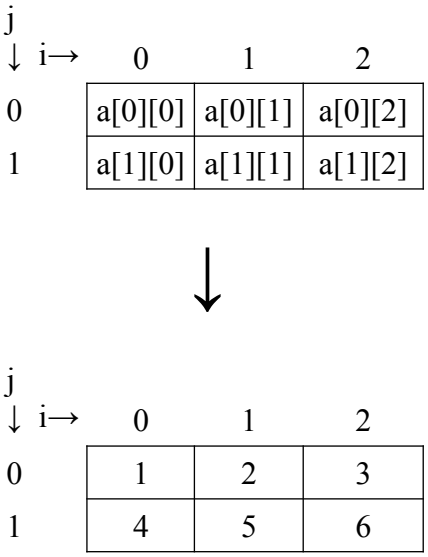


Fig.3.6: 2D array after initialization

Method 2: This method is better than Method 1 as the initialization of the elements can be visualized more clearly than Method 1. In this method the initialization is done by the following way.

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

From the above statement it is clear that first curly braces denotes the first row and the second curly braces denotes the 2nd row of the 2D array. Here we shall get the same results after initialization as shown in Fig.3.6.

Memory representation of 2D array – There are basically two types of conventions by which a 2D array may be represented namely row major and column major.

1. Row major ordering - A 2D array can be considered as the combination of some 1D arrays, because each row of a 2D array may be considered as a 1D array. Therefore we can assume that there are m number of 1D arrays of size = n in a 2D array of m no. of rows and n no. of columns. If we consider a 2D array of size = 2 × 3, there will be two 1D arrays with 3 elements each as shown in Fig.3.7.

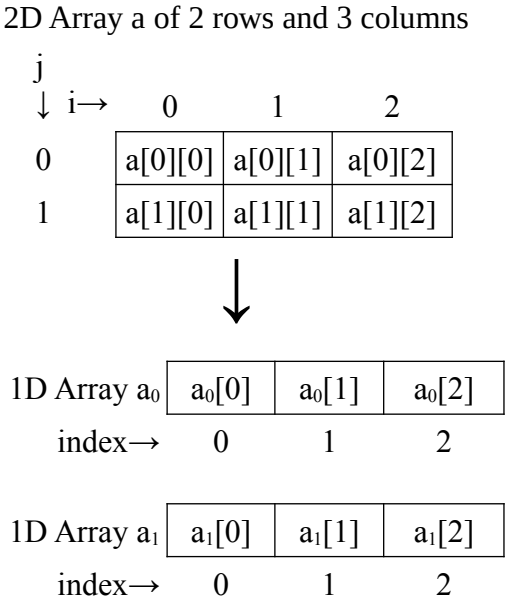


Fig.3.7: 2D array of size = 2 × 3 represented by two 1D arrays

We know that the elements are placed consecutively into the memory in case of a 1D array. If a 1D array is declared as int type, each element of the array will occupy 4 bytes in the memory space for a 32-bit architecture. According to the memory mapping of 1D array which has already been discussed previously the starting addresses of the consecutive elements in an integer type 1D array will be incremented by 4.

Memory mapping of a 2D array starts with the memory mapping of the first 1D array, then the second 1D array, after that third 1D array and so on sequentially. That means after the last element of 1st 1D array the 1st element of 2nd 1D array will be memory-mapped and after the last element of 2nd 1D array first element of 3rd 1D array comes. In this way the entire 2D array is memory-mapped row-wise as shown in Fig.3.8. This type of ordering of memory addresses in case of a 2D array is called as row major order. It is important to mention that C language follows row major form to represent the memory organization of a 2D array.

j ↓ i→	0	1	2	3	4	5
0	a[0][0] *1000*	a[0][1] *1004*	A[0][2] *1008*	A[0][3] *1012*	a[0][4] *1016*	a[0][5] *1020*
1	a[1][0] *1024*	a[1][1] *1028*	A[1][2] *1032*	A[1][3] *1036*	a[1][4] *1040*	a[1][5] *1044*
2	a[2][0] *1048*	a[2][1] *1052*	A[2][2] *1056*	A[2][3] *1060*	a[2][4] *1064*	a[2][5] *1068*

Nate: The number written in italics under each element of the 2D array in every cell are the starting address of every element.

Fig.3.8: Row major ordering for memory representation of a 2D array of size (3 × 6)

The following formula determines the address of any element in a 2D array of size (m × n) using row major ordering.

Address of a[i][j] = B + W × [(i – i_0) × n + (j – j_0)]

B = Base address of the 2D array
W = Storage size of each element in the 2D array in bytes
n = Maximum number of columns in the 2D array
i = Row index of the element a[i][j]
i_0 = Starting row index of the 2D array
j = Column index of the element a[i][j]
j_0 = Starting column index of the 2D array

Now we shall verify the above mentioned formula of row major ordering with the help of the 2D array shown in Fig.3.8. Suppose the address of the element a[2][3] is to be determined using this formula. In this case we have the values of the following parameters.

B = 1000, W = 4 bytes, n = 6, i = 2, i_0 = 0, j = 3, j_0 = 0

∴ Address of the element a[2][3] = B + W × [(i – i_0) × n + (j – j_0)]
= 1000 + 4 × [(2 – 0) × 6 + (3 – 0)]
= 1000 + 4 × [12 + 3]
= 1060

If the memory mapping of the 2D array shown in Fig.3.8 is compared with this result, it is exactly the same as before.

2. Column major ordering – In case of column major ordering the addresses of the successive elements are incremented by 4 column-wise. It says that the memory mapping is done for 1D array corresponding to 1st column, then for the 1D array corresponding to 2nd column and so on. After the address of the last element from the 1st column array the address of the first element of the 2nd column array comes. Column major ordering for a (3 × 6) 2D array is shown in Fig.3.9.

j ↓ i→	0	1	2	3	4	5
0	a[0][0] *1000*	a[0][1] *1012*	A[0][2] *1024*	A[0][3] *1036*	a[0][4] *1048*	a[0][5] *1060*
1	a[1][0] *1004*	a[1][1] *1016*	A[1][2] *1028*	A[1][3] *1040*	a[1][4] *1052*	a[1][5] *1064*
2	a[2][0] *1008*	a[2][1] *1020*	A[2][2] *1032*	A[2][3] *1044*	a[2][4] *1056*	a[2][5] *1068*

Nate: The number written in italics under each element of the 2D array in every cell are the starting address of every element.

Fig.3.9: Column major ordering for memory representation of a 2D array of size (3 × 6)

The following formula determines the address of any element in a 2D array of size (m × n) using column major ordering.

Address of a[i][j] = B + W × [(i – i_0) + (j – j_0) × m]

B = Base address of the 2D array
W = Storage size of each element in the 2D array in bytes
m = Maximum number of rows in the 2D array
i = Row index of the element a[i][j]
i_0 = Starting row index of the 2D array
j = Column index of the element a[i][j]
j_0 = Starting column index of the 2D array

Now we shall verify the above mentioned formula of column major ordering with the help of the 2D array shown in Fig.3.9. Suppose the address of the element a[2][3] is to be determined using this formula. In this case we have the values of the following parameters.

B = 1000, W = 4 bytes, m = 3, i = 2, i_0 = 0, j = 3, j_0 = 0

∴ Address of the element a[2][3] = B + W × [(i – i_0) + (j – j_0) × m]
= 1000 + 4 × [(2 – 0) + (3 – 0) × 3]
= 1000 + 4 × [2 + 9] = 1044 *(Verified from Fig.3.9)*

Here one point is important that the address of the last element of a 2D array will be same for both of the row major and the column major ordering.

Limitations of Array:

1) Arrays are declared as fixed size. If it is required to handle more number of elements than the size of the array, it is not possible to do that. Due to this reason array is referred as static data structure.
2) Data elements of an array are stored in continuous memory locations which may not be available always.
3) Insertion and deletion of element in case of array becomes problematic because of shifting of elements from their positions.

Different operation on a 1D array – In this section we shall discuss few operations which may be performed on a 1D array..

1. Insertion of an element into a 1D array – This operation inserts a new element at a specified position of the 1D array. If the position 'pos' is specified, then all the elements right of the position 'pos' and the element at position 'pos' will be shifted one position right to make a vacant space for the insertion of a new element.

Time complexity for the best case: When the new element is inserted at the last position, it is required to shift the last element only from (n – 1)th index to nth index of the array. Therefore only one shifting is required and this becomes the bast case. So the time complexity here will be O(1).

Time complexity for the worst case: When the new element is to be inserted at the 1st position, all the elements of the array (from 1st position to the last position) should be shifted right. Hence there will be n no. of shifting from the element a[0] to a[n-1], which becomes the worst case. Therefore time complexity for the worst case becomes O(n).

Algorithm to insert an element at a specific position of an array:

Step 1: Start

Step 2: Input the element which is to be inserted into the variable 'element'.

Step 3: Input the position at which the element is to be inserted into the variable 'pos'.

Step 4: Set i = n – 1 *[n is the size of the 1D array]*

Step 5: Repeat Step 6 to Step 7 while i >= pos – 1.

Step 6: Set a[i + 1] = a[i]

Step 7: Set i = i – 1

Step 8: Set a[pos – 1] = element *[To insert the element at specific position]*

Step 9: Set n = n + 1

Step 10: Stop

User defined function in C to insert an element at a specific position of the array.

```
int Insert(int a[],int n)
{
        int i, element, pos;

        printf("Enter the element to be inserted: ");
        scanf("%d", &element);

        printf("Enter the position for insertion: ");
        scanf("%d", &pos);

        for(i=n-1; i>=pos-1; i--)
                a[i+1] = a[i];

        a[pos - 1] = element;

        return n+1;
}
```

2. Deletion of an element from a 1D array – This operation will remove an existing element from the array. For this purpose the position 'pos' at which the element will be deleted is taken as input and all the elements beyond the index (pos – 1) will be shifted left by one position, which will replace the element at index (pos – 1) by the element at index pos. Thus the element at the specified position 'pos' will be removed.

Time complexity for the best case: The best case happens when the element at the last position is deleted. Here no left shifting is required to remove the last element, only the length of the array is decreases from n to (n – 1). Therefore the time complexity becomes O(1).

Time complexity for the worst case: The worst case occurs when 1st element is to be removed from the array. In this case all the elements starting from 2nd position to nth position are shifted left by one position. Thus we require (n – 1) no. of left shifting to accomplish this task. Therefore time complexity for the worst case becomes O(n).

Algorithm to delete an element at a specific position of an array:

Step 1: Start

Step 2: Input the position at which the element is to be deleted into the variable 'pos'.

Step 3: Set i = pos – 1.

Step 4: Repeat Step 5 to Step 6 while i <= n – 2 *[n is the size of the 1D array]*

Step 5: Set a[i] = a[i+1].

Step 6: Set i = i + 1

Step 7: Set n = n – 1

Step 8: Stop

User defined function in C to delete an element at a specific position of the array.

```
int Delete(int arr[],int n)
{
        int i, pos;

        printf("Enter the position for deletion: ");
        scanf("%d", &pos);

        for(i=pos-1; i<n-1; i++)
                arr[i] = arr[i+1];

        return n-1;
}
```

The entire C program to insert an element to a specified position of an array and also delete an element from a specified position of the same array is given below.

C program to insert an element to a specified position of an array and also delete an element from a specified position of the same array.

```
#include<stdio.h>
#include<stdlib.h>
#define Min 10
#define Max 100
#define ArrayLen 100

void Create(int [],int );
void Display(int [],int );
int Insert(int [],int );
int Delete(int [],int );

int main()
{
	int a[ArrayLen], n, option;
	int element, pos;

	printf("Enter the number of elements to create the array: ");
	scanf("%d", &n);

	Create(a, n);

	printf("The elements of the created array:\n");
	Display(a, n);

	while(1)
	{
		printf("*******************************\n");
		printf("Press 1 to insert an element.\n");
		printf("Press 2 to delete an element.\n");
		printf("Press 0 to exit the program.\n");
		printf("*******************************\n");
		printf("Enter your option: ");
		scanf("%d", &option);

		switch(option)
		{
			case 0: exit(1);

			case 1: n = Insert(a, n);
				printf("The elements of the array after insertion:\n");
				Display(a, n);
				break;
```

```
                        case 2: n = Delete(a, n);
                                printf("The elements of the array after deletion:\n");
                                Display(a, n);
                                break;

                        default: printf("Wrong option is selected.\n");
                                 break;
                }
        }

        return 0;
}

void Create(int arr[],int n)
{
        int i;

        for(i=0; i<n; i++)
                arr[i] = rand() % (Max - Min + 1) + Min;
}

void Display(int arr[],int n)
{
        int i;

        for(i=0; i<n; i++)
                printf("%d  ", arr[i]);

        printf("\n");
}

int Insert(int arr[],int n)
{
        int i, element, pos;

        printf("Enter the element to be inserted: ");
        scanf("%d", &element);
        printf("Enter the position for insertion: ");
        scanf("%d", &pos);

        for(i=n-1; i>=pos-1; i--)
                arr[i+1] = arr[i];

        arr[pos - 1] = element;

        return n+1;
}
```

```
int Delete(int arr[],int n)
{
        int i, pos;

        printf("Enter the position for deletion: ");
        scanf("%d", &pos);

        for(i=pos-1; i<n-1; i++)
                arr[i] = arr[i+1];

        return n-1;
}
```

Programming for Problem Solving

ES-CS201

Dr. S. K. Pradhan, ECE Dept.

nit-1: Introduction to ramming

Syllabus:

- Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.) - **(1 lecture).**
- Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm: Flowchart/Pseudocode with examples. **(1 lecture)**
- From algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and Logical Errors in compilation, object and executable code- **(2 lectures)**

omputer

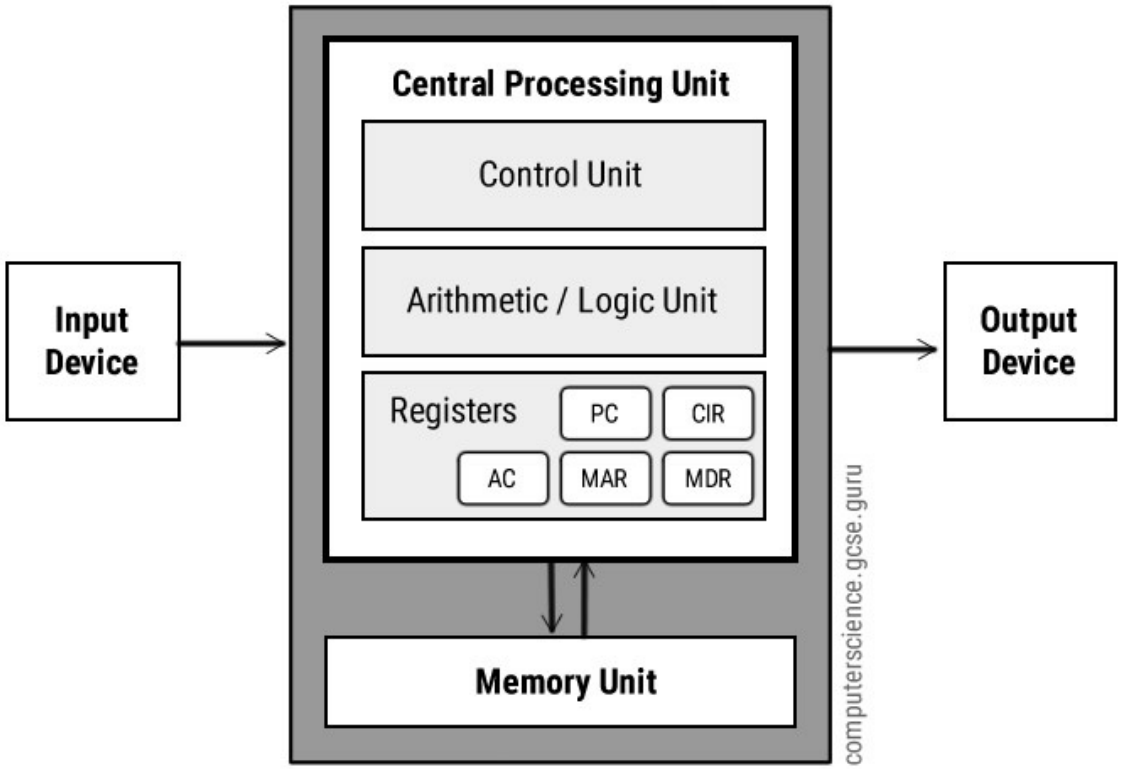
What is a computer?

- A computer is an electronic machine that accepts data, performs operations, displays results, and stores data/results in memory.
- It is a combination of **hardware and software** that provides various functionalities to the user.
- Hardware is the physical components of a computer like a processor, memory devices, monitor, keyboard, etc.
- While the software is a set of programs that are required by the hardware resources to function properly.

omponents of A Computer

Main components of a computer are shown in Figure:

- Input Unit
- Central Processing Unit
- Output Unit and
- Memory Unit



put Unit

- The input unit consists of input devices that are attached to the computer.
- Some of the common input devices are keyboard, mouse, joystick, scanner etc.
- A user inputs data and instructions through input devices such as a keyboard, mouse, etc.
- The input unit is used to provide data to the processor for further processing.

Central Processing Unit (CPU)

- Once the information is entered into the computer by the input device, the processor processes it.
- The CPU is called the brain of the computer because it is the control centre of the computer.
- It first fetches instructions from memory, then interprets them and finally executes or performs the required computation, and then either stores the output or displays it on the output device.
- The CPU has three main components, which are responsible for different functions: Arithmetic Logic Unit (ALU), Control Unit (CU) and Registers.

rithmetic and Logic Unit (ALU)

- Arithmetic and Logic Unit is a digital circuit that is used to perform arithmetic and logical operations.
- Arithmetic operations include addition, subtraction, multiplication and division.
- Logical operations include right and left shifts, comparison, Boolean operations like OR, Ex-OR, AND, NOT, NAND, NOR etc.
- ALU is the main component of the CPU and often called Kitchen of the processor.

egister

- A register is a temporary memory which resides inside the CPU.
- These are used to store data directly used by the processor.
- Registers can be of different sizes (16-bit, 32-bit, 64-bit and so on) and each register inside the CPU has a specific function, like storing data, storing an instruction, storing address of a location in memory etc.
- The user registers can be used by an assembly language programmer for storing operands, intermediate results etc.

Output Unit

- The output unit consists of output devices that are attached to the computer.
- It converts the binary data coming from the CPU to human understandable form.
- The common output devices are monitor, printer, plotter, etc.
- The output unit displays or prints the processed data from the CPU in a user-friendly format.

emory Unit

- Memory in a computer systems stores data and programs and are of two types: Primary and Secondary.
- Primary memory also called main memory directly communicates with the CPU.
- It is the place from where the program (set of instructions) is fetched for execution by the CPU.
- It also stores data that are required for execution of the instructions.
- Primary memory is volatile in nature and is normally populated by RAM called random access memory.

emory Unit

- Programs, both operating system and application, are stored in non-volatile memory called secondary memory like hard disk, CD, DVD, USB etc.
- Window, Linux etc. are operating systems which coordinate among the computer hardware, user and the application software.
- Application software allows users to create reports, letters, to compose emails, to play games etc.
- During booting up, the computer loads the operating system into its main memory.
- Application programs are also loaded into main memory while they are invoked.
- So, program is always run from its main or primary memory.

gorithm

- Computer programs can be written in a number of languages like assembly language, C, C++, Python, Java etc.
- A computer program is a set of instructions of a particular computer language which when executed will give some result/output.
- Before writing a computer program a planning called ***algorithm*** must be made.
- Writing a program without algorithm is like a house to build without a plan – both may lead to an inefficient program or a bad construction.

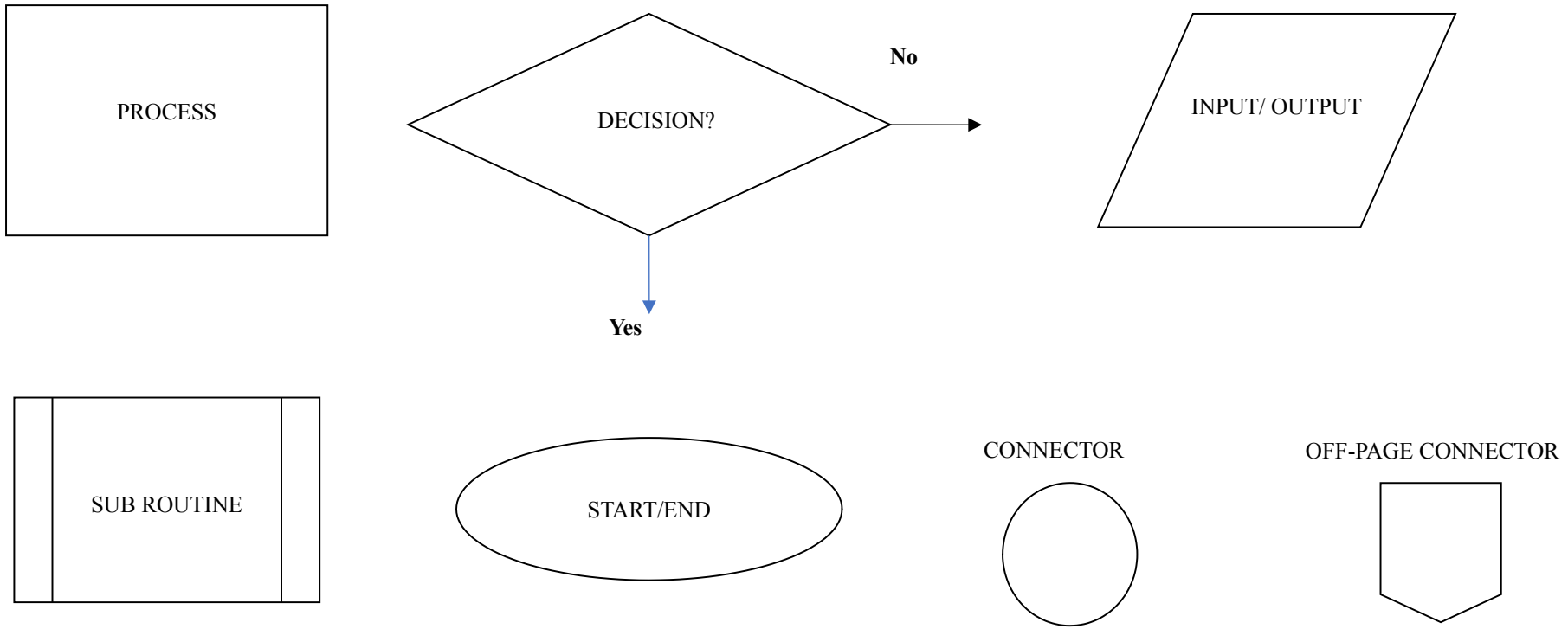
gorithm

- Thus, an algorithm is a step-by-step procedure that defines a set of instructions that must be carried out in a specific order to produce the desired result.
- Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming languages.
- Unambiguity, fineness, effectiveness, and language independence are some of the characteristics of an algorithm.

gorithm

- An algorithm can be developed in two ways, flowchart and pseudocode.
- Algorithm written through flowchart uses graphic shapes to represent different types of operations.
- Pseudocode, on the other hand, is an informal way of writing a program for better human understanding. It is written in simple English, making the complex program easier to understand.
- Pseudocode cannot be compiled or interpreted. It doesn't follow the programming language's syntax.

owchart


PROCESS
DECISION?
No
Yes
INPUT/ OUTPUT
SUB ROUTINE
START/END
CONNECTOR
OFF-PAGE CONNECTOR

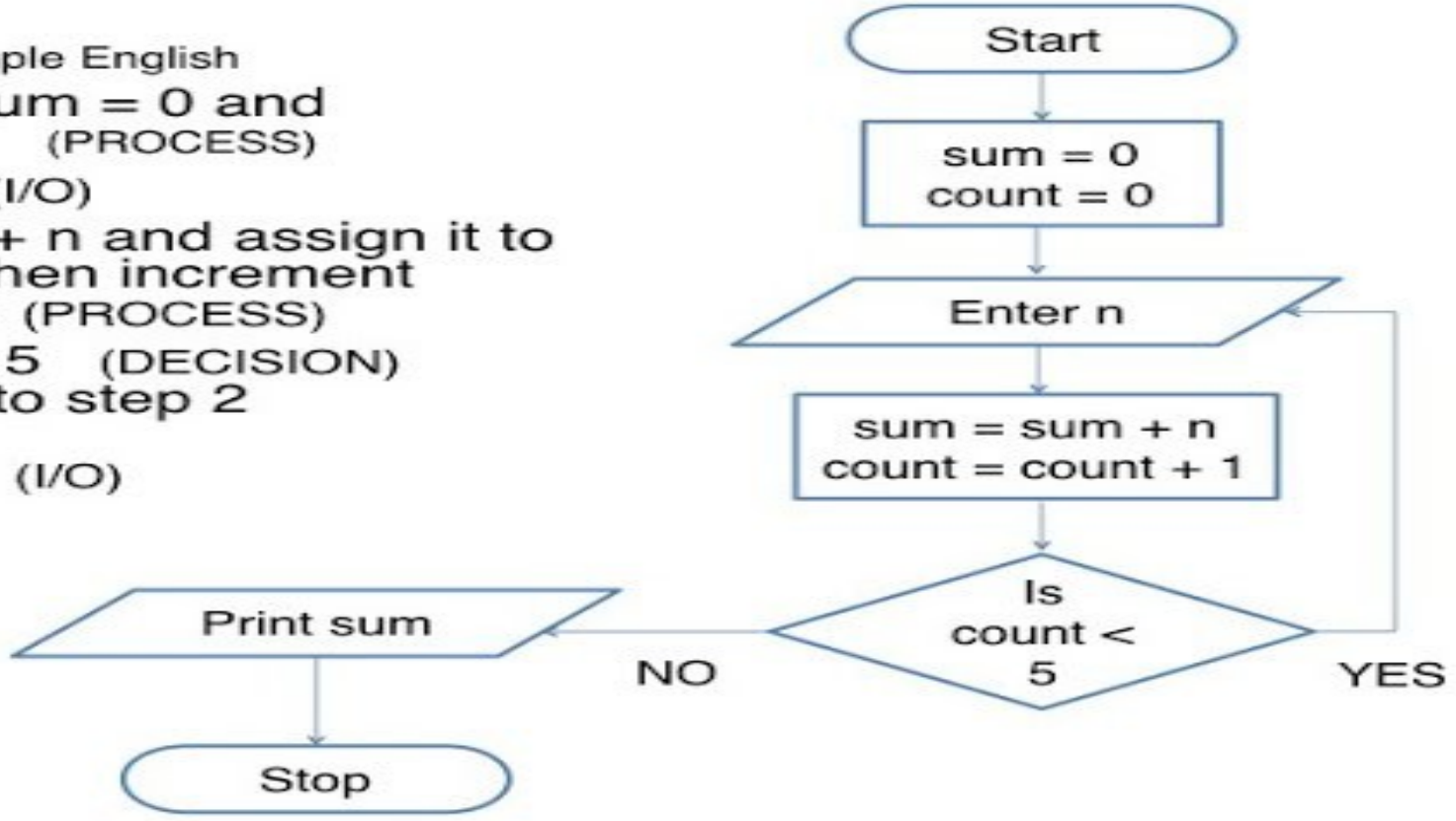
owchart Example

Find the sum of 5 numbers

Algorithm in simple English

1. Initialize sum = 0 and count = 0 (PROCESS)
2. Enter n (I/O)
3. Find sum + n and assign it to sum and then increment count by 1 (PROCESS)
4. Is count < 5 (DECISION)
   if YES go to step 2
   else
   Print sum (I/O)

Flowchart



Pseudocode

- Pseudocode is an informal way of programming description that does not require any strict programming language syntax.
- It is used for creating an outline or a rough draft of a program.
- System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.

Pseudocode Example: To make a Cup of Tea

Step-1: Start (get ready with the ingredients)

Step-2: Put the Teabag in a cup

Step-3: Fill the kettle with water

Step-4: Boil the water

Step-5: Pour some boiling water into the cup

Step-6: Add milk to the cup

Step-7: Add sugar to the cup

Step-8: Stir the tea

Step-9: Stop (ready to drink)

hat is C?

- C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.
- It is a very popular language, despite being old.
- C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

hy Learn C?

- It is one of the most popular programming language in the world.
- If you know C, you will have no problem learning other popular programming languages such as Java, Python, C++, C#, etc, as the syntax is similar.
- C is very fast, compared to other programming languages, like Java and Python.
- C is very versatile; it can be used in both applications and technologies.

fference between C and C++

- C++ was developed as an extension of C, and both languages have almost the same syntax.
- The main difference between C and C++ is that C++ support classes and objects, while C does not.

et Started With C

To start using C, you need two things:

- A text editor, like Notepad, to write C code
- A compiler, like GCC, to translate the C code into a language that the computer will understand

E: Integrated Development onment

- An IDE is used to edit and compile the code.
- Popular IDE's include Code::Blocks, Eclipse, and Visual Studio.
- These are all free, and they can be used to both edit and debug C code.
- Web-based IDE's can work as well, but functionality is limited.
- **Code::Blocks** is a good IDE to start with.
- You can find the latest version of Codeblocks at http://www.codeblocks.org/

  .
- Download the mingw-setup.exe file, which will install the text editor with a compiler

Quick Start

Let's create our first C file.

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

xplanation

- **Line-1:** #include <stdio.h> is a **header file library** that allows us work with input and output functions, such as printf() (used in line 4). Header files add functionality to C programs.
- **Line 2:** A blank line. C ignores white space. But we use it to make the code more readable.
- **Line 3:** Another thing that always appear in a C program, is main(). This is called a **function**. Any code inside its curly brackets {} will be executed.

xplanation

- **Line 4:** printf() is a **function** used to output/print text to the screen. In our example it will output "Hello World”
- **Line 5:** return 0 ends the main() function.
- **Line 6:** Do not forget to add the closing curly bracket } to actually end the main function.

serting New Line: \n Character

You can insert as many printf() function as you want but it does not insert a new line at the end of the line:

```
#include <stdio.h>
int main() {
printf("Hello World!");
printf("I am learning C.");
return 0;
}
```

Output:
Hello World!I'm learning C.

serting New Line: \n Character

You can insert a new line by using the \n character:

```
#include <stdio.h>
int main() {
printf("Hello World!\n");
printf("I am learning C.");
return 0;
}
```

Output:
Hello World!
I'm learning C.

serting New Line: \n Character

You can also use a single printf() to display multiple lines of text:

```
#include <stdio.h>
int main() {
printf("Hello World!\nI am learning C\nAnd it is awesome!");
return 0;
}
```

Output:
Hello World!
I'm learning C.
And it is awesome!

serting New Line: \n Character

Two consecutive \n insert a blank line:

```
#include <stdio.h>
int main() {
printf("Hello World!\n\n");
printf("I am learning C.");
return 0;
}
```

Output:

```
Hello World!


I'm learning C.
```

cape Sequence

- The new line character \n is called an escape sequence.
- It forces the cursor to change its position to the beginning of the next line. Examples of the other valid escape sequences are:

\t	Creates a horizontal tab
\\	Inserts a backslash character (\)
\”	Inserts a double quote character

Comments in C

- Comments can be used to explain code, and to make it more readable.
- It can also be used to prevent execution when testing alternative code.
- Comments can be **single-lined** or **multi-lined**.
- Single line comment starts with //
- Multi-line comments are written between /* Comments */
- Comments are ignored by the compiler.

omments in C

Example

```
/* The code below will print the greetings Good Morning Friends!

On the Monitor */

printf("Good Morning Friends!);
```

Variables

- Variables are used to store values.
- There are different data types in C and defined with different keywords. For example:
  - Int – stores integers (whole numbers) without decimals, such as 56, -98 etc.
  - Float – stores floating point numbers with decimal such as 99.34 or -23.45.
  - Char – stores single characters such as ‘a’ or ‘D’ surrounded by single quotes.

riable Declarations

- Variables in C should be declared (defined) before they will be used in a program.
- Syntax: type variableName = value;
- Examples: int myAge = 57;
- or without a value and assign the value later
- Example: int myAge; myAge = 57;

riable Declarations

- Variable will hold the latest data assigned to it replacing the old data.
- **Example:** int myAge = 57; // myAge is 57

```
myAge = 60;      // Now myAge is 60
```

rmat Specifiers

- If you want to display variable, you have to know “format specifiers”.
- Format specifiers are used with printf() function to tell the compiler what type of data the variable is storing.
- It is basically a Placeholder for the variable value.
- A format specifier starts with a percentage sign % followed by a character.
- For example, %d or %i- to output the value of an int variable etc.

rmat Specifiers

- Example:

```
int myAge = 57;
printf("My age is: %d", myAge);
```

Output:

My age is: 57

rmat Specifiers

- To print other types, use %c for char and %f for float. For example:

```
int myAge = 57; float myFloat = 9.34; char myLetter = 'D';
printf("My age is: %d\n", myAge);
printf("My Floating Number is: %f\n", myFloat);
printf("My Letter is: %c\n", myLetter);
```

Output:

```
My age is: 57
My Floating Number is: 9.34
My Letter is: D
```

rmat Specifiers

- Different types can be printed by a single printf() function as follows:

```
int myAge = 57;
char myLetter = 'D';
printf("My age is %d yrs. and my letter is %c", myAge, myLetter);
```

Output:

My age is 57 yrs. and my letter is: D

dd Variables Together

- In C, variables can be added together by using + operator. Example:

```
int x = 57;
int y = 65;
int sum = x + y;
printf("The sum is: %d", sum);
```

Output:

The sum is: 122

eclare Multiple Variables ther

- To declare more than one variables of same type, use a comma separated list as follows:

```
int x = 57, y = 65, sum;
sum = x + y;

printf("The sum of %d and %d is: %d", x, y, sum);
```

Output:

The sum of 57 and 65 is: 122

eclare Multiple Variables ther

- You can also assign same value to multiple variables of same type:

```
int x, y, z;
x = y = z = 12;

printf("Sum =  %d", x + y + z);
```

Output:

Sum = 36

Variable Names

- A C variable must be identified by a unique name and is called ***identifier***.
- Identifiers can be short names like x, y, z etc.
- They can be more descriptive like myAge, sum, totalVale etc.
- It is recommended to use descriptive names in order to create understandable and maintainable code.
- Example: ***int minutesPerHr = 60*** is preferable than ***int m = 60***

Variable Names – General Rules

Variable names or identifiers follow the following rules:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (***myage*** and ***myAge*** are different variables)
- Names cannot contain whitespace or special characters like **!, #, %** etc.
- Reserved words such as ***int***, ***char*** cannot be used as names

onstants

If you want a variable will not be overwritten, use ***const*** keyword which makes the variable unchangeable and read-only.

Example:

```
const int minutesPerHr = 60;

const float PI = 3.14;
```

Note: When a constant variable is declared, it must be assigned with a value.

```
const float PI;

PI = 3.14; // Not correct
```

onstants

Good Practice

It is a good practice to declare the constants with UPPERCASE. It is not required but useful for code readability and common for C programmers.

Example:

```
const int BIRTHYEAR = 1987;
const float PI = 3.14;
```

perators

- Operators are used to operate on variables and values

- + operator adds two or more values and variables

Example

```
int sum1 = 10 + 5;          //15 (10 + 5)
Int sum2 = sum1 + 20;       //35 (15 + 20)
Int sum3 = sum1 + sum2;     //50 (15 + 35)
```

perators

C operators are of following types:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

rithmetic Operators

Arithmetic operators are used to perform arithmetic operations like:

Operator	Name	Description
+	Addition	Adds values and variables
-	Subtraction	Subtracts one value from the other
*	Multiplication	Multiplies two values
/	Division	Divides one value by the other
%	Modulus	Returns the remainder of division operations
++	Increment	Increases the value of a variable by one
--	Decrement	Decrements the value of a variable by one

ssignment Operators

Assignment operators (=) are used to assign values to variables

Example: int x = 10;

The ***addition assignment operator*** (+=) adds a value to a variable and assigns back to it.

Example: int x = 10; x + = 5; //x = x + 5 = 15

ssignment Operators

List of assignment operators are as follows

Operator Name	Operator	Example	Same As
Basic assignment	=	X = 15	X = 15
Addition assignment	+=	X += 5	X = X + 5
Subtraction assignment	-=	X -= 5	X = X – 5
Multiplication assignment	*=	X *= 5	X = X * 5
Division assignment	/=	X /= 5	X = X / 5
Modulo assignment	%=	X %= 5	X = X % 5

ssignment Operators

List of assignment operators are as follows (continued.....)

Operator Name	Operator	Example	Same As
Bitwise AND assignment	&=	X & = Y	X = X & Y
Bitwise OR assignment	|=	X |= Y	X = X | Y
Bitwise XOR assignment	^=	X ^= Y	X = X ^ Y
Bitwise left shift assignment	<<=	X <<= Y	X = X << Y (X left shifted by Y times)
Bitwise right shift assignment	>>=	X >>= Y	X = X >> Y (X right shifted by Y times)

omparison Operators

Comparison operators are used to compare two values which returns either true (1) or false (0). List of comparison operators are:
