3. Programming Language and Its Applications (ACtE03)

3.1 Introduction to C programming: C Tokens, Operators, Formatted/Unformatted Input/output, Control Statements, Looping, User-defined functions, Recursive functions, Array (1-D, 2-D, Multi-dimensional), and String manipulations. (ACtE0301)

3.2 Pointers, structure and data files in C programming: Pointer Arithmetic, Pointer and array, passing pointer to function, Structure vs Union, array of structure, passing structure to function, structure and pointer, Input/output operations on files, and Sequential and Random Access to File. (ACtE0302)

3.3 C++ language constructs with objects and classes: Namespace, Function Overloading, Inline functions, Default Argument, Pass/Return by reference, introduction to Class and object, Access Specifiers, Objects and the Member Access, Defining Member Function, Constructor and its type, and Destructor, Dynamic memory allocation for objects and object array, this Pointer, static Data Member and static Function, Constant Member Functions and Constant Objects, Friend Function and Friend Classes. (ACtE0301)

3.4 Features of object-oriented programming: Operator overloading (unary, binary), data conversion, Inheritance (single, multiple, multilevel, hybrid, multipath), constructor/destructor in single/multilevel inheritances. (ACtE0304)

3.5 Pure virtual function and file handling: Virtual function, dynamic binding, defining opening and closing a file, Input / Output operations on files, Error handling during input/output operations, Stream Class Hierarchy for Console Input /Output, Unformatted Input /Output Formatted Input /Output with ios Member functions and Flags, Formatting with Manipulators. (ACtE0305)

3.6 Generic programming and exception handling: Function Template, Overloading Function Template, Class Template, Function Definition of Class Template, Standard Template Library (Containers, Algorithms, Iterators), Exception Handling Constructs (try, catch, throw), Multiple Exception Handling, Rethrowing Exception, Catching All Exceptions, Exception with Arguments, Exceptions Specification for Function, Handling Uncaught and Unexpected Exceptions. (ACtE0306)

Nepal Institute of Engineering

Welcome To C Programming World

Topics to be covered

- C Tokens
- Operator
- Formatted/ Unformatted I/O
- Control Statement
- Looping
- User defined function
- Recursive function
- Array[1-D, 2-D, Multidimensional]
- String manipulation

Features of C

- It is a robust language with rich **set of built-in functions** and **operators** that can be used to write any complex program.
- The C compiler combines the **capabilities of an assembly language** with **features of a high-level language.**
- Programs Written in C are **efficient and fast**.
- C is **highly portable** this means that programs **once written can be run on another machines** with little or no modification.
- Another important feature of C program, is its ability to **extend** itself.
- A C program is basically a collection of functions that are supported by C library. We can also create our **own function** and add it to C library.
- C language is the most widely used language in operating systems and embedded system development today.

Example

```
// Program to add two numbers
#include<stdio.h>
#include<conio.h>
int sum(int,int);
int s;
void main()
{
      int a,b;
      a=2;
      b=3;
      s=sum(a,b);
      printf("Sum is %d",s);
      getch();
}
int sum(int x, int y)
{
      return(x+y);
}
```

Document Section

Link Section

Global Declaration Section

Declaration Part

Execution Part

Main Function Section

User Defined Function

Character Sets

- **Alphabets**

  -- Uppercase letters A - Z

  -- Lowercase letters a – z

- **Digits**

  -- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- **Special Characters**

  ~ → tilde
  | → vertical bar
  + → plus sign
  _ → underscore
  > → greater than
  % → percent sign
  @ → at symbol
  < → less than
  - → minus sign
  ^ → caret etc…..

- **White Space**

  \b → blank space
  \v → vertical tab
  \f → form feed
  \t → horizontal tab
  \r → carriage return
  \n → new line etc…

C Tokens

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as C tokens.

C tokens are of six types:

- Keywords — **Example**: int, while etc…
- Identifiers — **Example** : main, total etc…
- Constants — **Example** : 10, 20 etc…
- Strings — **Example** : “total”, “hello” etc…
- Special symbols — **Example** : (), {} etc..
- Operators — **Example** : +, /,-,* etc..

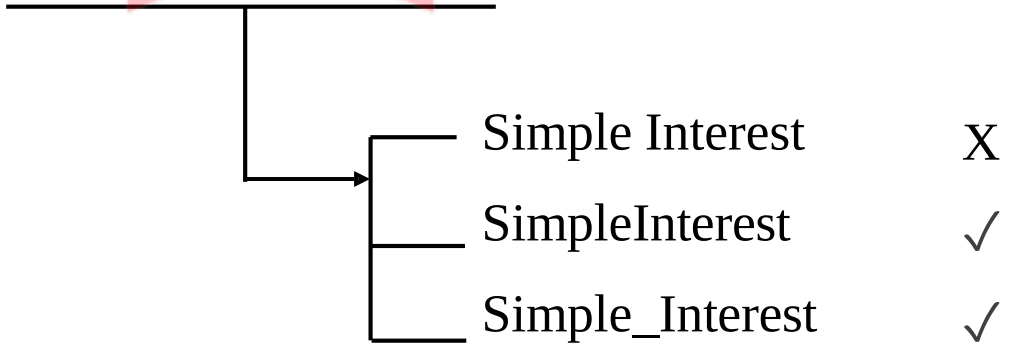
Keywords

- There are certain **reserved** words, called keywords that have standard, predefined meanings in C.
- They cannot be used as programmer-defined identifiers.
- There are 32 keywords in c programming they are:

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

Identifiers

- Identifiers are names that are given to various program elements, such as variables, functions and arrays.
- Rules for identifiers
  - First character must be an alphabet (or Underscore).
  - Must consist of only letters, digits or underscore.
  - Only first 31 characters are significant.
  - Cannot use a keyword.
  - Must not contain white space.



Data types

- The data type in **C** defines the amount of storage allocated to variables, the values that they can accept, and the operation that can be performed on those variables
- There are following type of data types supported by c programming

1. Primary Data Type
2. Derived Data Type
3. User Defined Data Type

Primary(Fundamental) Datatypes

- Character type
- Integers type
- Floating type
- Void type

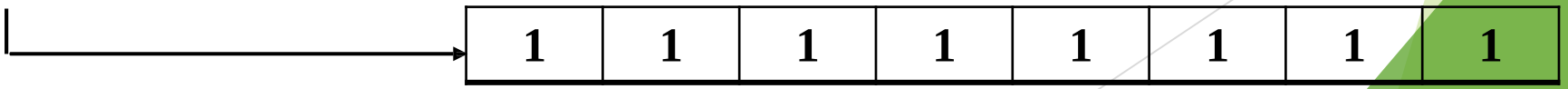
Character Type

- A single character can be defined as a character type data.
- Characters are stored in 8 bits (1 Byte).
- It is represented by a keyword “char”.
- Its conversion character is %c.

Type	Memory Required	Conversion Character	Value Range
Char	1 byte	%c	-128 to 127 i..e -2^7 to 2^7-1
Unsigned char	1 byte	%c	0 to 255 i..e 0 to 2^8-1
Signed char	1 byte	%c	-128 to 127 i..e -2^7 to 2^7-1

- Note 1 Byte= 8 bit
- Largest possible value that can be represented in 8 bit is (2^8-1) i..e 255

Binary of 255



Integer Type

- ✓ Integers are the whole numbers, i.e. non-fractional numbers.
- ✓ Integers are stored in 16 bits (2 Byte).
- ✓ Generally it is represented by a keyword “int”.
- ✓ Integer are of following type.

Type	Represented by	Conversion Character	Memory occupied	Value Range
Signed Integer	signed int or int	%d	2 byte	-2^{15} to $2^{15}-1$
Unsigned Integer	unsigned int	%u	2 byte	0 to $2^{16}-1$
Signed Short Integer	signed short	%d or %i	2 byte	-2^{15} to $2^{15}-1$
Unsigned Short Integer	unsigned short	%u	2 byte	0 to $2^{16}-1$
Signed Long Integer	signed long int or long int	%ld	4 byte	-2^{31} to $2^{31}-1$
Unsigned Long Integer	unsigned long int	%lu	4 byte	0 to $2^{32}-1$

Floating Point Types

- ✓Floating types are fractional numbers (i.e. real numbers).
- ✓They are defined in c by keyword float.
- ✓Generally floating numbers reserve 32 bits of storage.

Type	Represented by	Conversion Character	Memory Occupied	Value Range
Floating Point	Float	%f	4 byte	-2^{31} to $2^{31}-1$
Double	double	%lf	8 byte	-2^{63} to $2^{63}-1$
Long Double	long double	%Lf	10 byte	-2^{79} to $2^{79}-1$

Void Type

- ✓The void type has no value.
- ✓It is usually used to specify type of function when it does not return any value to calling function

User Defined Data types

- The user defined data types enable a program to invent his own data types and define what values it can take on.
- C supports 2 types of user defined data types.
  - typedef (type definition)
  - enum (enumerated data type)

Example of typedef

Program without using typedef	Program using typedef

#include<stdio.h>
void main()
{
int a;
a=3;
printf("Value of a=%d",a);
}
 	
	#include<stdio.h>
	void main()
	{
	typedef int num;
	num a;
	a=3;
	printf("Value of a=%d",a);
	}

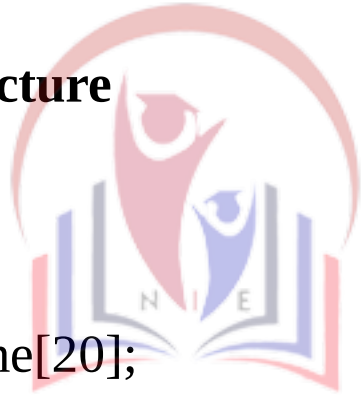
Enumeration

- An **enumeration** is used in any programming language to define a constant set of values. For **example**, the days of the week can be defined as an **enumeration** and used anywhere in the program
- Example:

```
#include <stdio.h>
enum week {Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Sunday;
    printf("Day %d",today);
    return 0;
}
```

Derived Datatype

- ✓ Data types that are derived from the built-in data types are known as derived data types.
- ✓ The various derived data types provided by C are *arrays, pointers and structures.*



Example of Structure

```
struct student
{
    char Name[20];
    int Roll;
    float Marks;
}s;
```

Variables

- variable is an identifier that is used to represent a single data item, i.e., a numerical quantity or a character constant

Example: *int num*; Here num is an integer type variable.

float num; Here num is an floating type variable.

The declaration of variables mainly has two significances:

- It gives name of variables and its type.
- It allocates appropriate memory space according to its data types.

Rules for naming variables:

- First character must be an alphabet (or Underscore).
- Must consist of only letters, digits or underscore.
- Only first 31 characters are significant.
- Cannot use a keyword.
- Must not contain white space.

Constants\Literal

- constant is a value or an identifier whose value cannot be altered in a program.

  Example: 1, 2.5, "C programming is easy", etc.

Constants can be

- Integer constant — **const** int a= 5;
- Floating point constant — **const** float b= 3.14;
- Character constant — **const** char a='A';
- String Constant — **const** str[15]="My name is C Programming";

Integer Constant

- Integer constant is a numeric constant (associated with number) without any fractional or exponential part

  Example: Decimal constant :- 2, 0, 9, -20 etc..

  Octal Constant:- 021, 077, 033 etc..

  Hexadecimal constant: - 0x7F, 0x2A etc

Floating-point constants

- A floating point constant is a numeric constant that has either a fractional form or an exponent form.

  Example:- 2.0, 0.0000234, -0.22E-5 etc..

Character constants

- A character constant is a constant which uses single quotation around characters.
- They have equivalent ASCII values.

  Example: 'a', 'l', 'm', 'F' etc..

String Constant

- A string constant is a sequence of characters enclosed in double quotes.
- It may contain letters, numbers, special characters or blank spaces. However it does not have equivalent ASCII values.

  Examples:- "Hello", "2008", "5+3" etc..

Preprocessor Directive

- C preprocessor is a collection of special statements, called directives, that are executed at the beginning of compilation process
- begin with the symbol # in column one and do not require a semicolon at the end
- Example
  - #define
  - #include

Symbolic Constant

- A symbolic constant is name that substitute for a sequence of character that cannot be changed.
- usually defined at the beginning of the program.

Example:
#define PI 3.141593 ⟶ We can use symbol PI instead of constant term 3.141539 program.
#define TRUE 1
#define FALSE 0

Escape Sequence

- Certain nonprinting characters used in c are called escape sequences.
- An escape sequence always begins with a backslash and is followed by one or more special characters.

Example:	**Character**	**Escape Sequence**
	bell (alert)	\a
	backspace	\b
	horizontal tab	\t
	vertical tab	\v
	newline (line feed)	\n etc…..

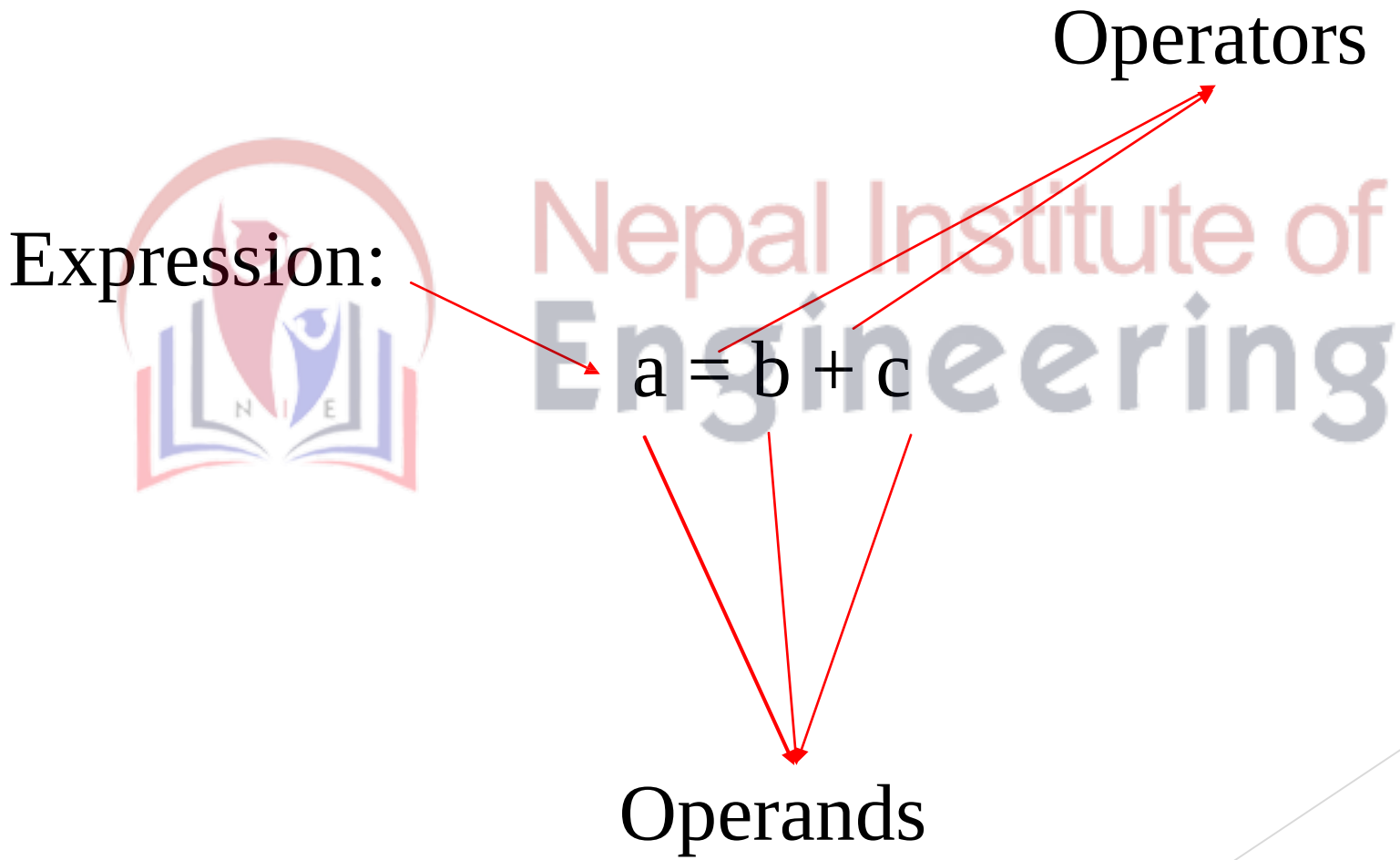
Operator

Nepal Institute of Engineering

Topics to be covered

- Operators and Expressions
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment/Decrement Operators
- Conditional Operators
- Bitwise Operators
- Comma Operators
- Sizeof() Operators
- Precedence and Associativity
- Type casting in C

Operators and Expression



- C supports a rich set of built in operators.
- An operator is a symbol that tells the computer to perform mathematical or logical manipulations
- Operators are broadly categorized into:
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and Decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators

Arithmetic operator

- performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).
- Arithmetic operators are:- +, -, *, /, %.

Example: value= a+b;

Relational Operators

- checks the relationship between two operands.
- If the relation is true, it returns 1; if the relation is false, it returns value 0.
- Relational operators are:- >, <, <=, >=, ==, !=

Example: a=10;

b=20;

b>a ⟶ returns 1

a>b ⟶ return 0

Logical Operator

- used in decision making in C programming.
- logical operator returns either 0 or 1 depending upon whether expression results true or false.
- Logical operators are: &&, ||, !.

Example:

```
a=5;
b=10;
if(a==5 && b==10)  ——→  returns 1.
if(a==5 && b==20)  ——→  returns 0.
```


Assignment Operators

- used for assigning a value to a variable
- Assignment operators are: =, +=, -=, *=, /=, %=

Example:

```
a+=b  ——→  a=a+b.
a*=b  ——→  a=a*b
```

Increment/Decrement operator

- Two operators: increment operator (++) and decrement operator (--).
- ++ operator increases the value of operand by 1.
- -- operator decreases the value of operand by 1.
- Increment operator are of two type: - pre increment operator (++a).
  -post increment operator (a++).
- Decrement operator are of two type: - pre decrement operator (--a).
  -post decrement operator (a--).

Example: (1) int a=10, b;

b = a++; ⟶ Assign value to b and increases value of a to 11.

i..e value of b will be 10, but value of a will be 11.

(2) int a=10, b;

b = ++a; ⟶ Increases value of a to 11 and assign value to b.

i..e value of a will be 11 and value of b will be 11.

Note: Decrement operators are same as increment operator but decrement operators decreases the value by 1.

Conditional Operator

- The operator (?:) is known as conditional operator.
- A conditional expression is written in the form:

  expression1? expression2: expression3;

  - expression1 is evaluated first.
  - If expression1 is true, the value of expression2 is the value of conditional expression.
  - If expression1 is false, the value of expression3 is the value of conditional expression.

- **Example:** (1) a = 10;

  b = 15;

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

  Here (a>b) is false so value of b will be assign to x, i..e value of x will be 15.

  (2) a = 10;

  b = 15;

  x = (a< b)? a: b;

  Here (a<b) is false so value of a will be assign to x, i..e value of x will be 10.

Bitwise Operators

- Performs operation in bit level i..e in binary form.
- Bitwise operators are:
  - Bitwise AND operator (&).
  - Bitwise OR operator (|).
  - Bitwise XOR operator (^).
  - Bitwise one's complement operator (~).
  - Bitwise Right shift operator (>>).
  - Bitwise Left shift operator (<<).

Bitwise AND operator (&)

- The output of bitwise AND is 1 if corresponding bits of two operand is 1, else 0.

Example:

```
int a =12, b= 25, c;
c= a & b;
```

Here value of c will be 8.

a = 12 ↓(Binary) 0000 1100	b = 25 ↓(Binary) 0001 1001	c= a & b i..e a= 0000 1100 & b= 0001 1001 c= 0000 1000 So value of c is 8.

Bitwise OR Operator (|)

- The output of bitwise OR is 0 if corresponding bits of two operand is 0, else 1.

Example: int a =12, b= 25, c;

c= a | b;

Here value of c will be 29.

a = 12 ↓ (Binary) 0000 1100	b = 25 ↓ (Binary) 0001 1001	c= a | b i..e a= 0000 1100 | b= 0001 1001 c= 0001 1101 So value of c is 29.

Bitwise XOR Operator (^)

- The results of XOR operator is 1, if corresponding two bits are opposite, else 0.

Example: int a =12, b= 25, c;

c= a ^ b;

Here value of c will be 21.

a = 12 ↓ (Binary) 0000 1100	b = 25 ↓ (Binary) 0001 1001	c= a ^ b i..e a= 0000 1100 ^ b= 0001 1001 c= 0001 0101 So value of c is 21.

Bitwise 1's Complement Operator (~)

- It is unary operator and has effects of flipping bits i..e the results of ~ operator is 1, if given bit is 0 and 0 if given bits is 1.

Example: int a = 12, b;

b = ~a;

Here value of b will be 115.

a = 12 ↓ (Binary) 0000 1100	b= ~a i..e a= 0000 1100 so b= 1111 0011 So value of b is 115

Bitwise Right shift Operator (>>)

- Right shift operator shifts all bits towards right by certain number of specified bits.

Example: (1) int a= 212, b;

b = a >> 1

So value of b will be 106.

a= 212 ↓ (Binary) 1101 0100	Now a= 1 1 0 1 0 1 0 0 Eleminated 0 1 1 0 1 0 1 0 (Right shif by 1 i..e >>1 ↓ Added (Always added bit at MSB will be 0) So value of b will be 106

(2) int a= 212, b;

b = a >> 2

Here Right shift is performed 2 times.

i..e 1 1 0 1 0 1 0 0 → Given Question

0 1 1 0 1 0 1 0 (a >> 1) → Shifted Right by 1 bit.

b = 0 0 1 1 0 1 0 1 (a >> 2) → Again shifted Right by 1 bit.

i..e Value of b becomes 53.

Bitwise Left shift operator (<<)

- ✓ Left shift operator shifts all bits towards left by certain number of specified bits.

Example: (1) int a= 212, b;

b = a << 1

a= 212 ↓ (Binary) 1101 0100	Now a= 1 1 0 1 0 1 0 0 1 1 0 1 0 1 0 0 0 Always added bit at LSB will be zero

Here shift is performed 2 times.

(2) int a= 212, b;

b = a << 2

i..e 1 1 0 1 0 1 0 0 → Given Question

1 1 0 1 0 1 0 0 0 (a <<1) → Shifted Left by 1 bit.

b = 1 1 0 1 0 1 0 0 0 0(a <<2) → Again shifted Left by 1 bit.

i..e Value of b becomes 848.

Special Operators

- C supports some special operators such as comma operator, size of operator.
  - Comma Operator (,).
  - Sizeof Operator.

Comma Operator (,)

- The comma operator can be used to link the related expression together.
- Comma linked lists of expression are evaluated left to right and the value of right-most expression is the value of combined expression.

Example:

Value = (x = 10, y = 5, x + y),

Here, 10 is assigned to x and 5 is assigned to y and so expression x+y is evaluated as (10+5) i.e. 15.

Sizeof() Operator

- The sizeof() operator is used with an operand to return the number of bytes it occupies.

Example:

(1)
```
int a, value;
value = sizeof(a);
printf("value = %d", value);
```
display value = 2

(2)
```
float a;
int value;
value = sizeof(a);
printf("value = %d", value);
```
display value = 4

Operator precedence and Associativity

- A higher precedence operator is evaluated before a lower precedence operator.
- precedence and associativity of arithmetic operators:

Operators	Decription	Associativity	Precedence Rank
() []	Function call Array element Reference	Left -> Right	1
++ -- !	Increment Decrement Logical negation	Right -> Left	2

~ * & sizeof	Ones complement Pointer reference Address Sizeof operator	Right -> Left	2
* / %	Multiplication Division Modulus	Left -> Right	3
+ -	Addition Subtraction	Left -> Right	4
<< <<	Left shift Right Shift	Left -> Right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left -> Right	6

== !=	Equality Inequality	Left -> Right	7
&	Bitwise AND		8
^	Bitwise XOR		9
|	Bitwise OR		10
&&	Logical AND		11
||	Logical OR		12
?:	Conditional operator		13
= += -= *= /= %=	Assignment operator	Right -> Left	14
,	Comma operator	Left -> Right	15

Types of Control statements

- Selective (Branching)
  - a) Two way branching : if...else
  - b) Multiple branching : if...else if...elseif and switch
- Repetitive (Looping)
  - a) Entry controlled : while and for
  - b) Exit controlled : do...while
- Jumping:
  - a) goto
  - b) break
  - c) continue

Selective (Branching) Statements

- Selective structure are used when we have a number of situations where we may need to change the order of execution of statements based on certain condition.
- The decision making statements test a condition and allow to execute some statements on the basis of result of the test (i.e. either true or false).
- C provides the following statements for selective structures.

  a) Two way branching : if...else

  b) Multiple branching : if...else if...elseif and switch

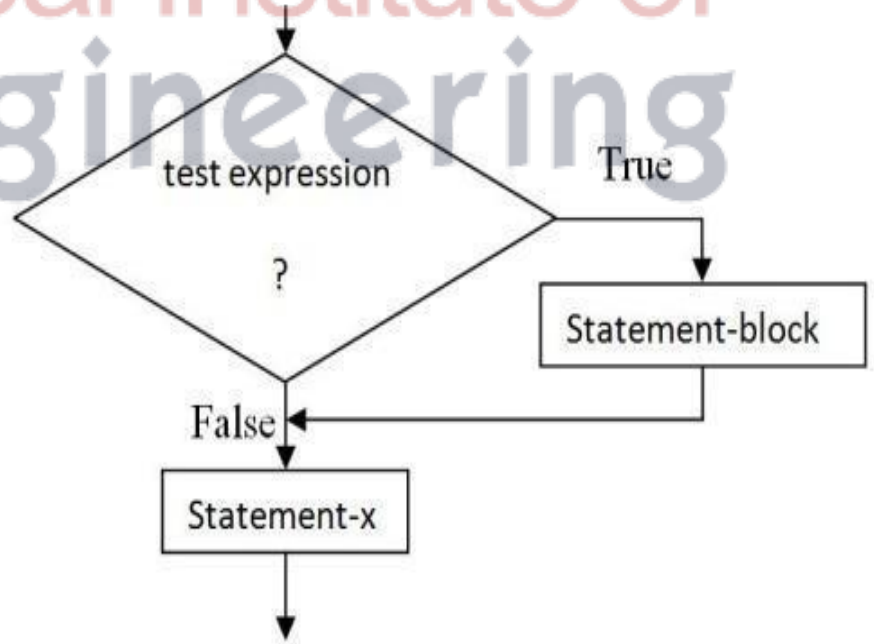
The if statement

- The **if** statement is used to execute a block of code conditionally based on whether the given condition is true or false.
- If the condition is true , the block of code will be executed, otherwise it will be skipped.

Syntax:

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

Example



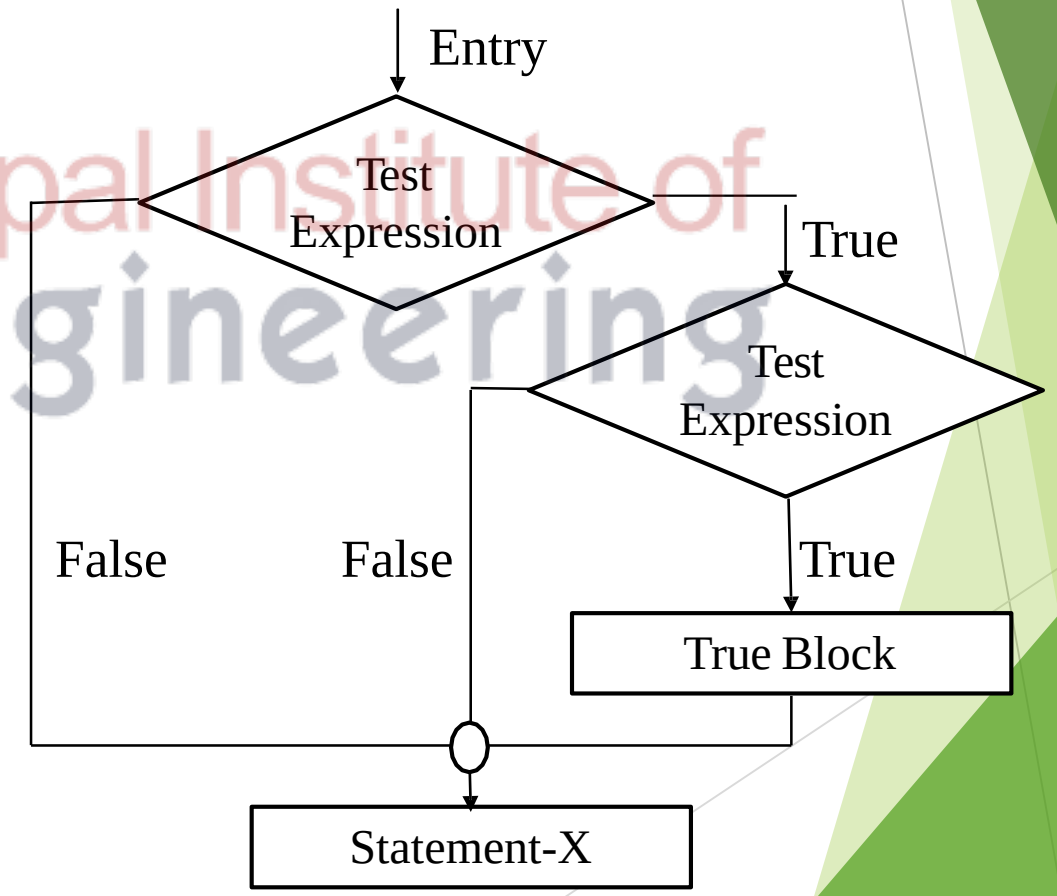
Nested if statement

- If a ***if*** statement is written within the body of another ***if*** statement then it is called Nested ***if*** statement

Syntax

```
if(test expression)
{
    if(test expression)
    {
        true block statement(s)
    }
}
```

Flowchart



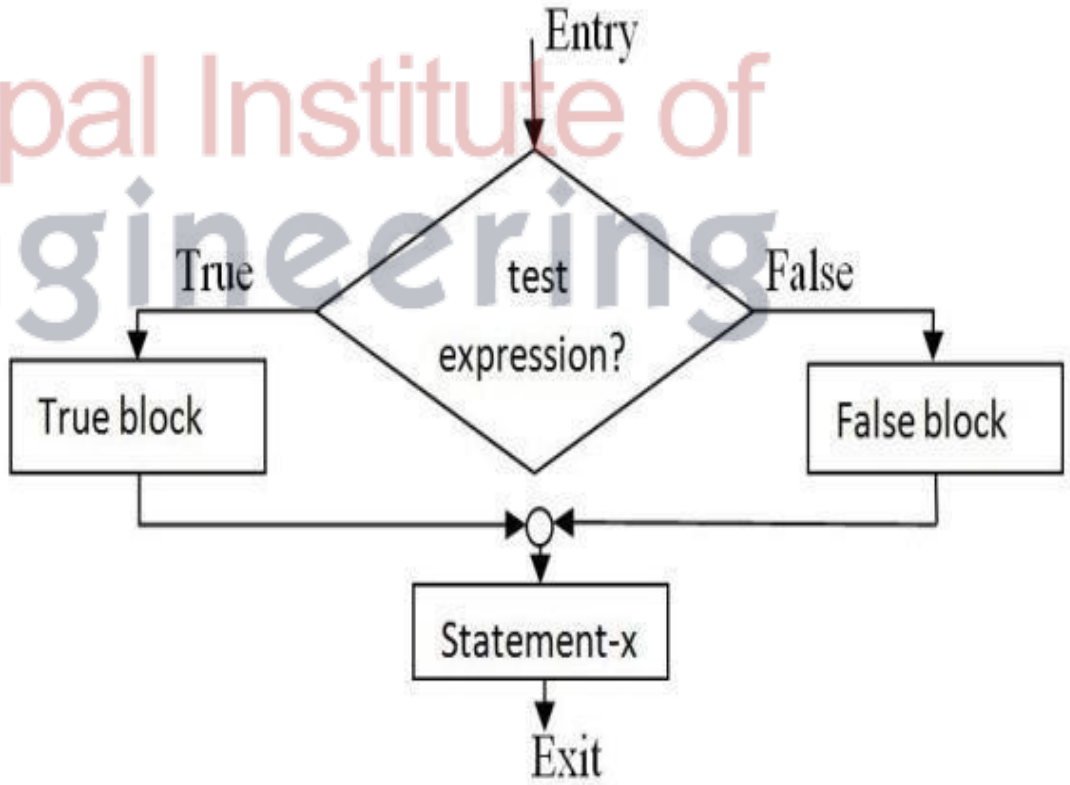
The if … else statement

- ✓ The if...else statement extends the idea of the if statement by specifying another section of code that should be executed only if the condition is false

Syntax:

```
if(test expression)
 {
      true block statement(s)
 }
 else
{
     false block statement(s)
}
```

Flowchart



Nested if … else Statement

- When a series of decision are involved, we may have to use more than one if...else statement in nested form.

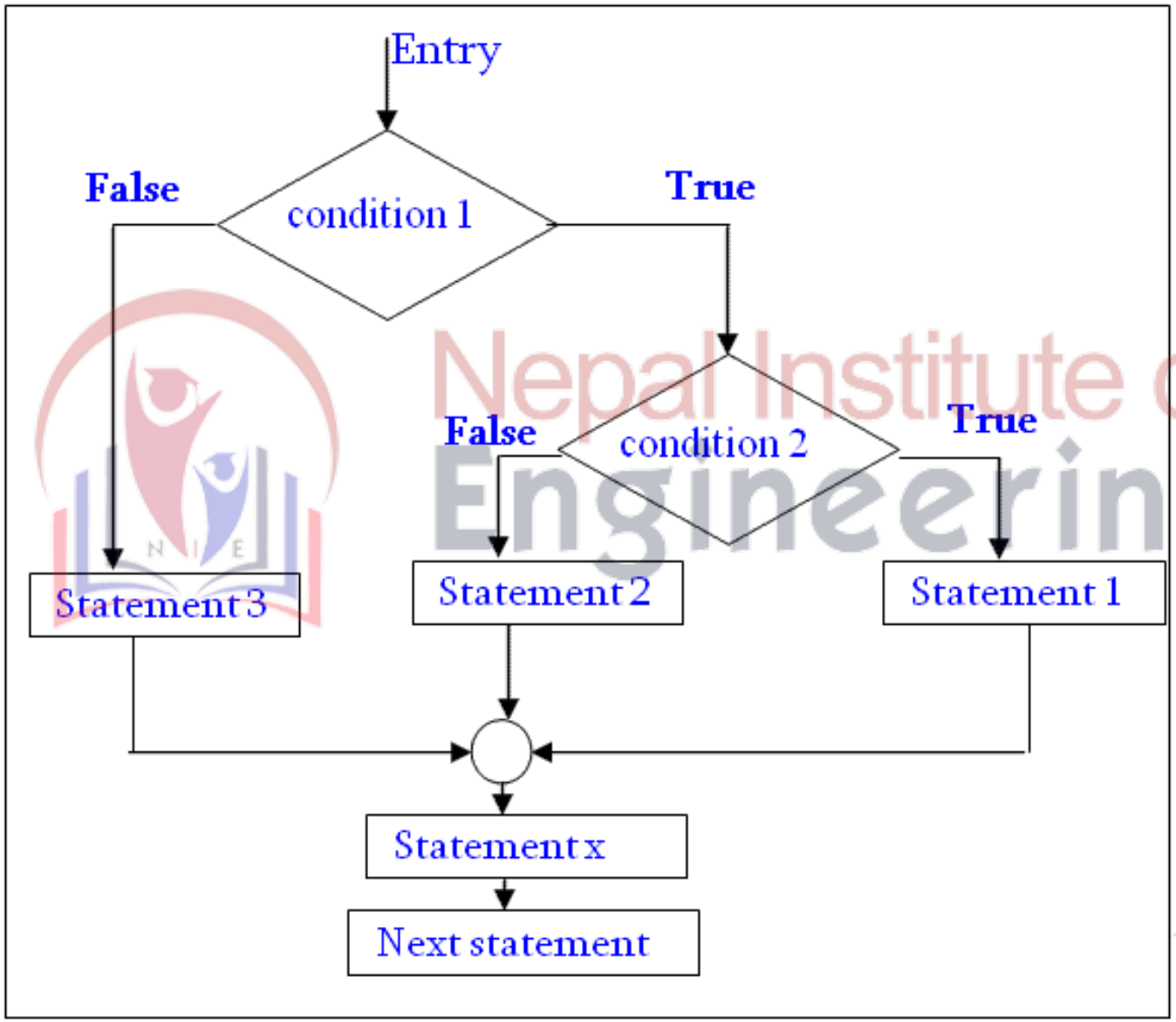
Syntax1:

```
if(test-expression1)
{
    if(test-expression2)
    {
        statementblock-1;
    }
    else
    {
        statementblock-2;
    }
}
else
{
statementblock-3;
}
```

Syntax2:

```
if(test-expression1)
{
    statementblock-1;
}
else
{
    if(test-expression2)
    {
        statementblock-2;
    }
    else
    {
        statementblock-3;
    }
}
```

Flow chart



The if ..elseif statements (elseif Ladder)

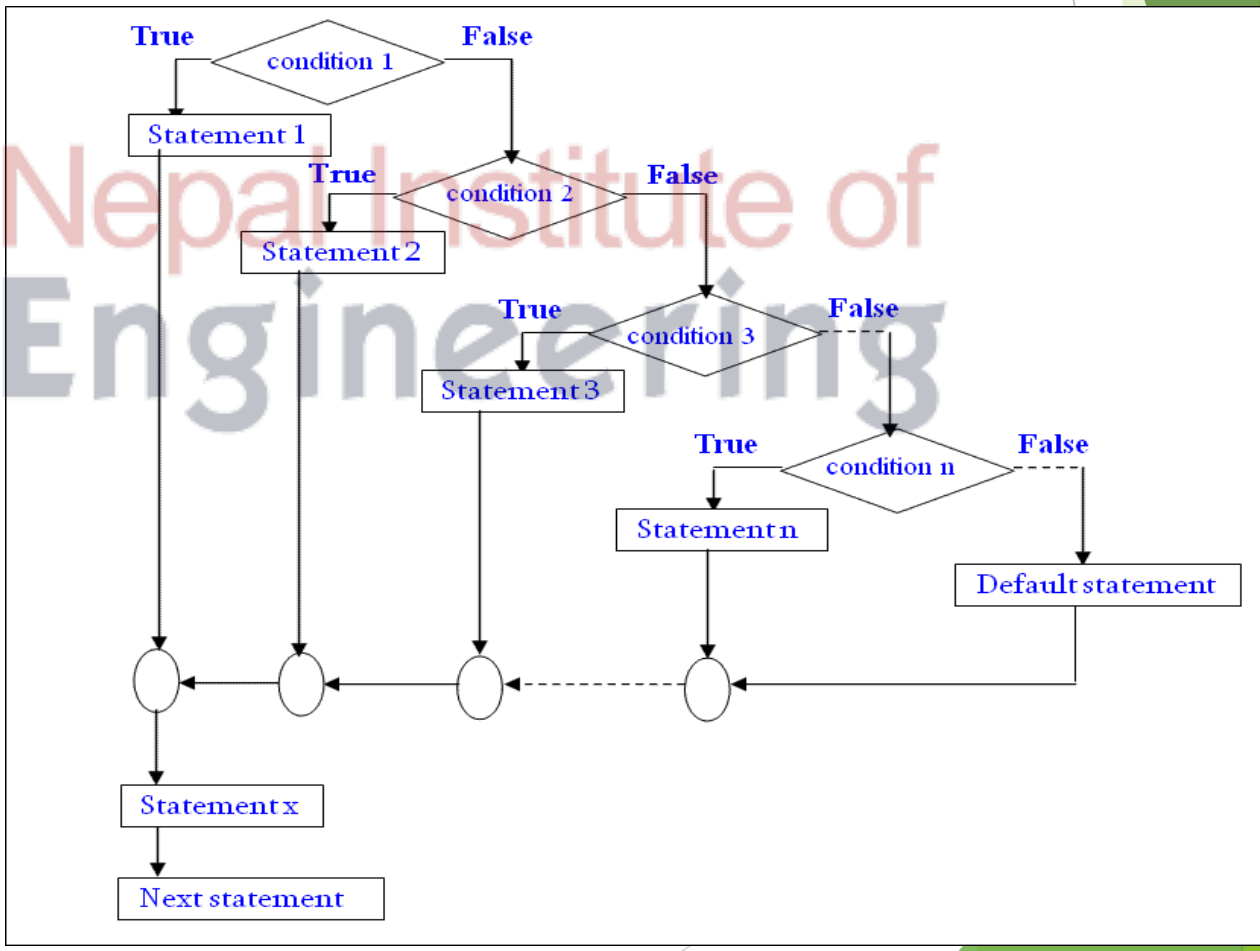
- It is another way of putting **ifs** together when multipath decisions are involved. A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**

Syntax

```
if (condition-1)
     statement-1;
else if (condition-2)
     statement-2;
else if (condition-3)
     statement-3;
.....
.....
else if (condition-n)
     statement-n;
else
     default-statement;

statement-x;
```

Flowchart:



The switch Statement

- C has built a multi way decision statements known as switched, that tests the value of an expression against a list of case values (integer or character constants).
- When a match is found, the statements associated with that case is executed.

Syntax

```
switch (expression)
{
        case constant1:
            block of case constant1;
            break;
        case constant2:
            block of case constant2;
            break;
        case constant3:
            block of case constant3;
            break;
        .....
        .....
        default:
 }
```

Flowchart:

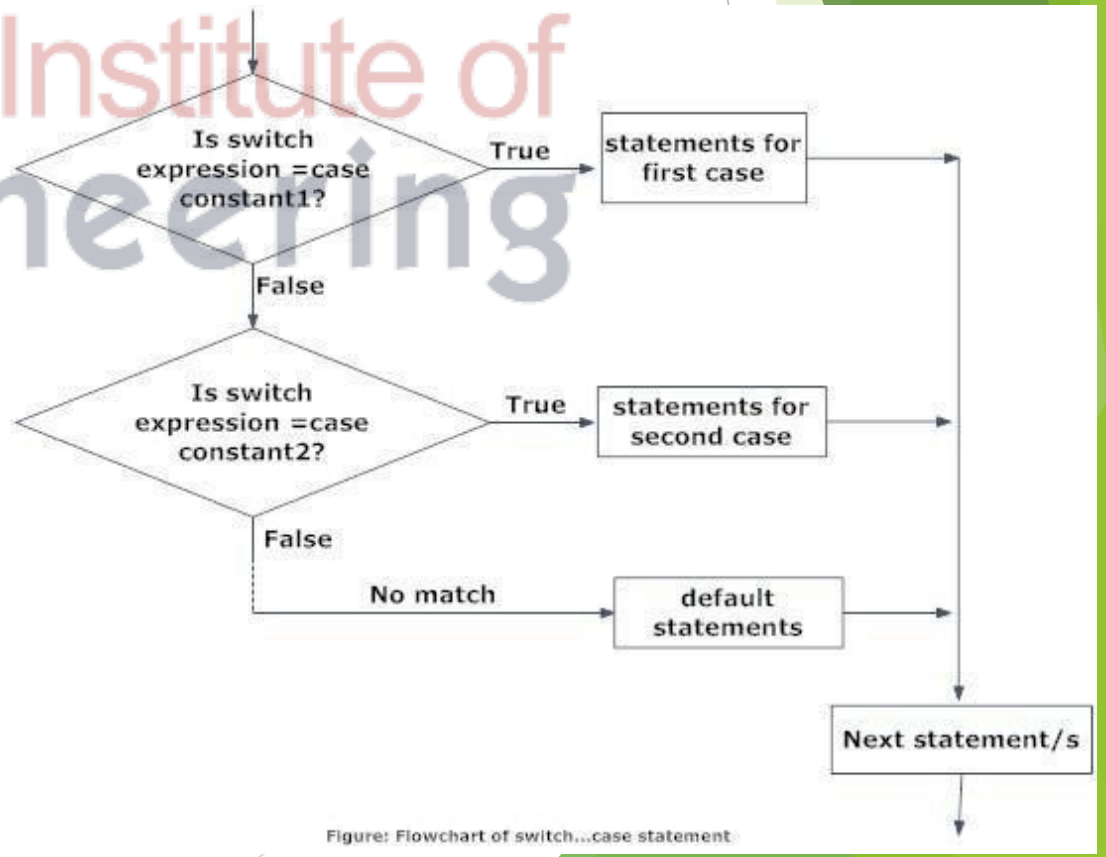


Figure: Flowchart of switch...case statement

Repetitive Structure (Looping)

- Repetition means executing the same section of code more than once.
- A looping process, in general, would include the following 4 steps.

  1) Setting and initialization of a counter.

  2) Execution of statements in the loop.

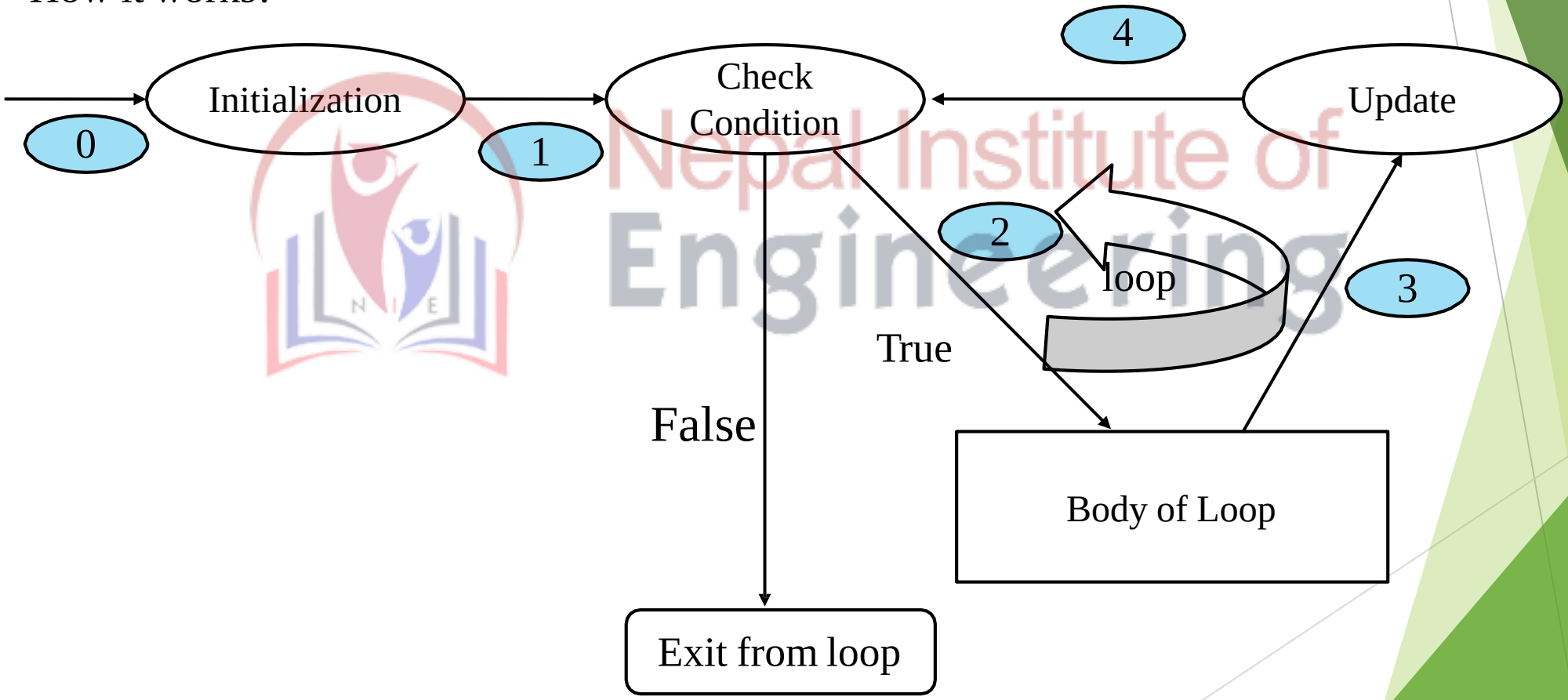
  3) Test for a specified condition for execution for the loop.

  4) Updating the counter.

- C provides 3 loop constructs for performing loop generations.
  - For loop
  - while loop
  - do while loop

The for loop

- **for** statement is used to execute a block of code for a fixed number of repetition.
- It is entry controlled loop
- How it works?



Syntax

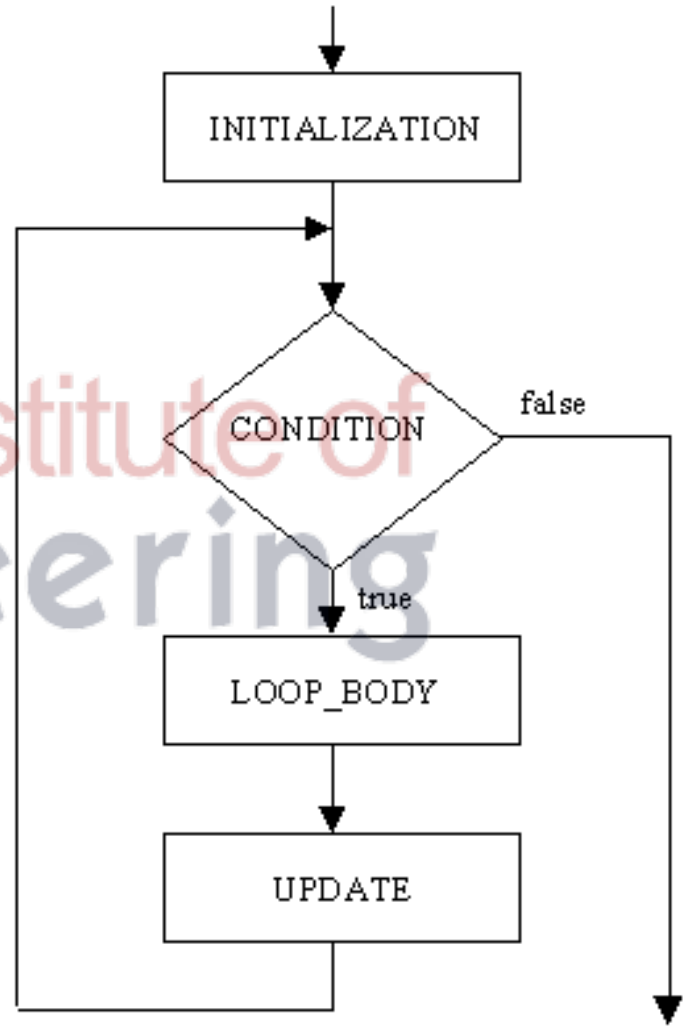
```
for (initialization ; test_expression; update_expression)
{
        body of loop.
}
```

Example:

```
void main()
{
    int i, n = 10, count = 0;
    for (i = 0; i < n ; i++)
     {
            count++;
      }

 }
```

Flow



The while loop

- ✓ It specifies that a section of code should executed while a certain condition holds true.
- ✓ It is entry controlled loop.

Syntax:

```
while(test_expression)
{
    //body of loop
}
```

Example:

```
void main()
{
    int n=1, count=0;
    while ( n <= 10)
    {
        count++;
    n++;
    }
```

Flowchart:

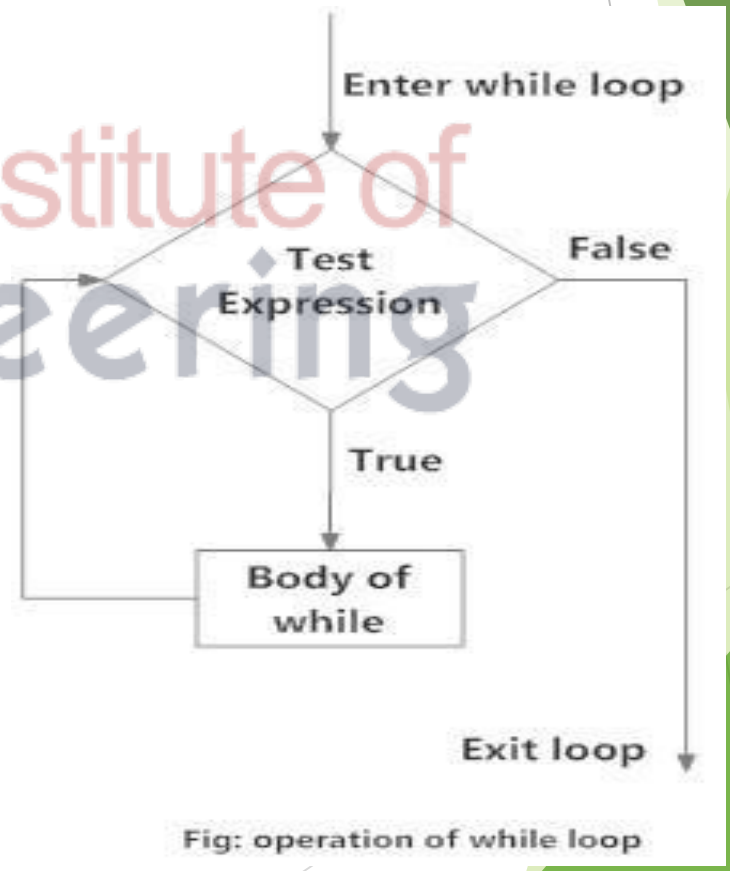


Fig: operation of while loop

The do while loop

- ✓ It also specifies that a section of code should be executed while a certain condition holds true.
- ✓ It is exit controlled loop.

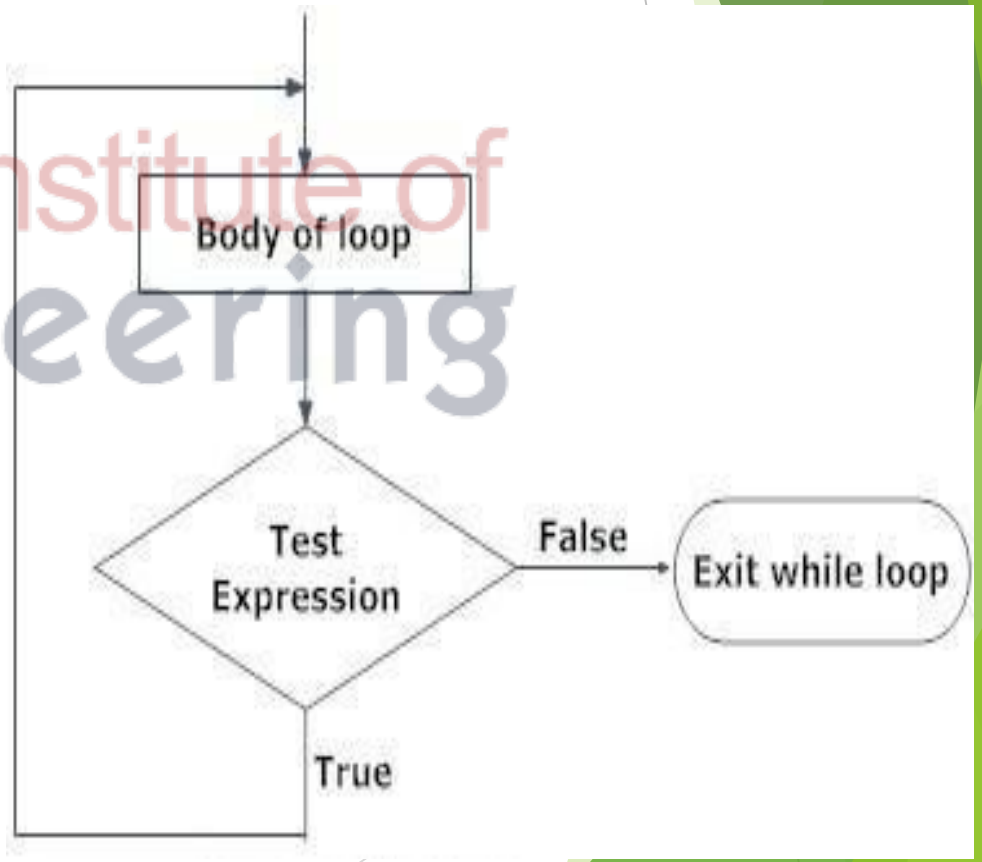
Syntax

```
do
{
        //body of loop.
}while(test_expression);
```

Example:

```
void main()
{
        int n = 1, count = 0;
        do{
                count++;
                n++;
            } while ( n <= 10);
}
```

Flowchart



Differentiate between entry controlled and exit controlled loops

Entry Controlled Loop	Exit Controlled Loop
Test condition appears at the beginning of loop.	Test condition appears at the end of loop.
Control variable is counter variable.	Control variable is counter and sentinel variable.
Each execution occurs by testing condition.	Each execution except first the first one occurs by testing condition.
for and while loop belongs to entry controlled loop.	do...while loop belongs to exit controlled loop.

Example: ----------
Sum=0;
N=1;
while(n<=10)
{
sum= sum + pow(n,2);
n=n+1;
}
---------- 	
	Example: ----------
	do
	{
	
	printf("Input a number: ");
	scanf("%d",&num);
	}while(num>0);
	---------- 	

Nested loops

- Putting one loop statement within another loop statement is called nesting of loop.
- Nested loops can be :
  - Nested for loops
  - Nested while loops
  - Nested do while loops

Nested for loops

- If one *for* statement is within the another *for* statement is called nesting of *for* loops.

Syntax:

```
for ( initialization ; test_expression ; update_expression )
 {
      for ( initialization ; test_expression ; update_expression )
           {
                // body;
           }
 }
```

Example:

```
void main()
{
        int i, j, n;
        for( i = 0 ; i < n ; i++)
        {
          for( j = 0 ; j < n ; j++)
              {
                  // Body of for Loop
              }
        }
}
```

Concept Associated with function

- Function Declaration or Prototype
- Function Definition
- Passing Arguments
- Return Statement
- Function Call
- Elimination of Function Declaration

Example

```
// Program to add two numbers
#include<stdio.h>
#include<conio.h>
int sum(int,int);
void main()
{
      int a,b;
      a=2;
      b=3;
      s=sum(a,b);     //Function Call
      printf("Sum is %d",s);
      getch();
}
int sum(int x, int y)       //Function Definition
{
      return(x+y);
}
```

Function Declaration

Main Function Section

User Defined Function

Function Declaration or Prototype

- Provides information to the compiler
  - Name of the function
  - Type of the value to be returned(optional, default return type is integer)
  - Number of arguments that must be supplied in a call to the function
  - Type of Arguments
- When a function call is encountered, the compiler checks the function call with its declaration so that correct arguments types are used.
- **Syntax:**
  - Return_type function_name(type 1, type 2, type 3, …, type n);
- **Examples :**

float add(int,int);

Or,

float add(int a, float b);

Function Definition

- **A function definition has two principal components:**
  - **Function Declarator**
  - **Function Body/Body of the Function**
- **Function Declarator**: We must declare as same as the function declaration[function name, no of arguments and their types, return type] must be same.
- **Syntax:**
  - Return_type function_name(type1 arg1, type2 arg2, type3 arg3, …, type n argn);

- **Here,**
  arg1 agr2 arg3 … argn n are called **Formal Arguments** or **formal parameters.**
- **Function Body**: Function declarator is followed by function body started and ended with curly brackets and contains the tasks to be done by that function

Passing Arguments

- Providing values to the **formal arguments** of called function through **the actual arguments** of calling function is called passing arguments

Example:

```
void add( int, int);
void main(){
    int a=5,b=6;
    add(a ,b);
}
void add( int x, int y)  {
    printf("Sum =%d", x+y );
}
```

Note: **main()** is Calling function because it calls add function and **add()** function is Called function because it is being called by main()

Return Statement

- A Function may or may not send back any value to the calling function. If it does, it is through **return** statement
- The called function can **only return one value** per call, at the most.
- **Syntax** :

return ; or **return(**expression**);**

- Plain return does not return any value and acts as a closing bracket of the function.
- Example: void main(){ return; }

Function call

- Function call is an inactive part of the program which comes in to life when a call is made to the function
- A function call is specified by function name followed by the values of parameters enclosed within parenthesis and terminated by a semicolon.
- **Example :**

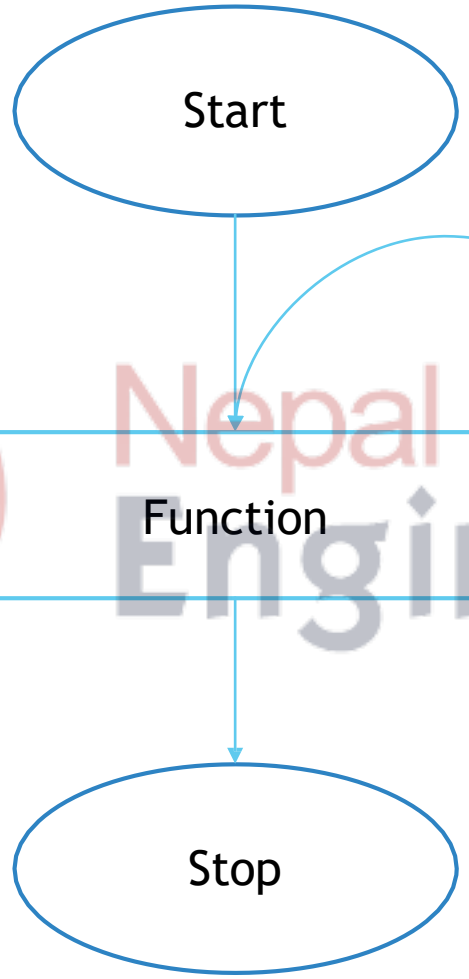
add(a , b);

- **Note:** the **number**, **type** and **order of arguments** along with **function name** should be same in ***function declaration*** , ***function call*** and ***function declarator***

Introduction

- Recursion is the process of expressing a function in terms of itself.
- Function call itself unless some specific condition is satisfied i.e, base condition.
- The problem is solved by repeatedly breaking into smaller problems, which is similar in nature to the original problem.
- Each time a function calls itself and it must be closer to the solution.

Recursion



Example 2: fibonacci Series [Main Function]

```
#include<stdio.h>
int fibo(int);
int main()
{
   int n, i = 0, c;
    scanf("%d",&n);
   printf("Fibonacci series\n");
   for ( c = 1 ; c <= n ; c++ )
   {
      printf("%d\n", fibo(i));
      i++;
   }
return 0;
}
```

Example: [Sub Function]

```
int fibo(int n)
{
   if ( n == 0 || n==1) return n;
else
      return (fibo(n-1) + fibo(n-2) );
}
```

Tracing the Evaluation

fibo(5) = fibo(4) + fibo (3)

= fibo(3) + fibo (2) + fibo(2) + fibo (1)
= fibo(2) + fibo (1) + fibo(1) + fibo (0) + fibo(1) + fibo (0) + fibo(1)
= fibo(1) + fibo (0) + fibo (1) + fibo(1) + fibo (0) + fibo(1) + fibo (0) + fibo(1)
= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1
= 5

ARRAY

- An array is a group of related data items that share a common name.

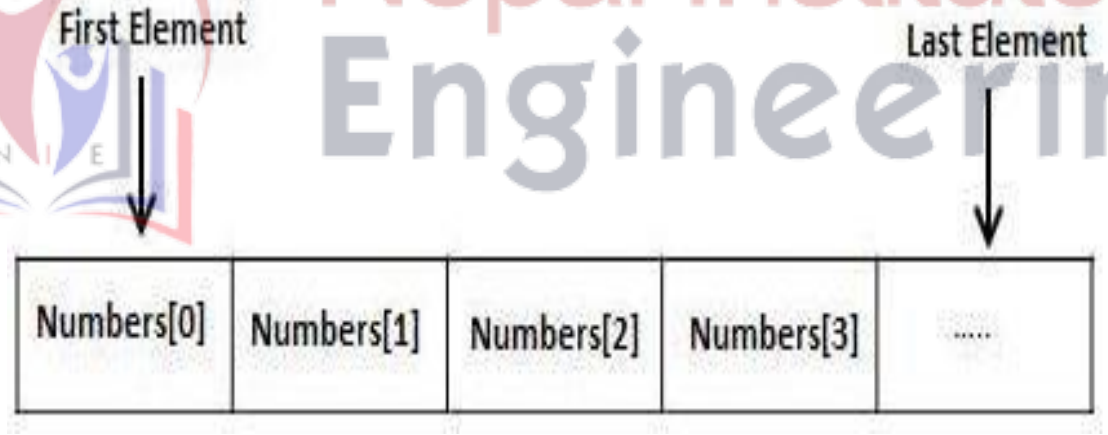
  i.e. a data structure as a single entity

- Simple variable is a single memory location with unique name and a type whereas an array is a collection of adjacent memory locations that have one collection name and type.
- The individual values are called **“elements”** in an array and

  length of an array = number of elements.
- Each element is identified by its position number or **“index” or “Subscript”** in an array. [An index always begins with value **0**].

ARRAY

- All arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.



Advantages of Arrays

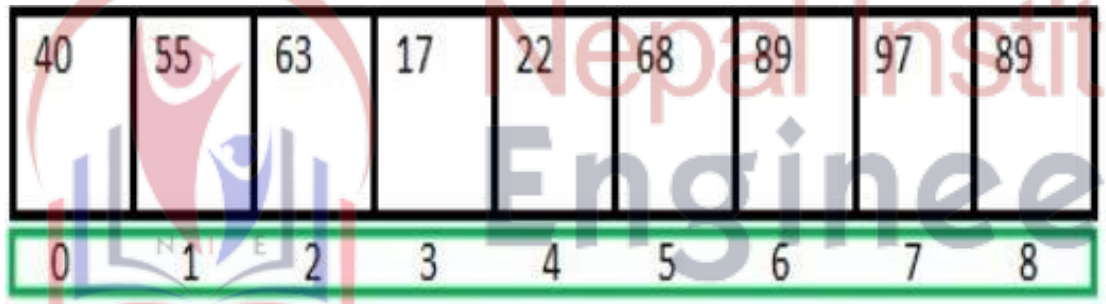
- Arrays can store a large number of values with single name.
- Value stored in arrays can be accessed/processed easily and quickly and can be sorted and conduct searching process easily.
- Array can be used for matrix computations as well.

Types of Array

- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

One-Dimensional Array

- One-Dimensional Array is also called as linear array i.e 1-D array.
- It stores data in a single row or column.



Array Length = 9
First Index = 0
Last Index = 8

- **Declaration of 1-D Array**

  Syntax:

  storage_class data_type array_name[size]

  int num[5];

- **Initialization of 1-D Array**
  - The process of assigning values to array elements at the time of array declaration is called array initialization.

  Syntax:

  storage_class data_type array_name[size] = [value1,value2,……]

- **Declaration of 1-D Array**

  Syntax:

  storage_class data_type array_name[size]

  int num[5];

- **Initialization of 1-D Array**
  - The process of assigning values to array elements at the time of array declaration is called array initialization.

  Syntax:

  storage_class data_type array_name[size] = [value1,value2,……]

- **The array initialization can be of following types:**

  i. int a[5] = {1,2,3,4,5};

  here, array ‘a’ has 5 elements and values are assigned as (a[0]=1, a[1]=2, a[2]=3, a[3]=4, a[4]=5)

  ii. int b[ ] ={ 2,5,7};

  here, the size of array is automatically set by compiler ,according to number of values given or number of elements.

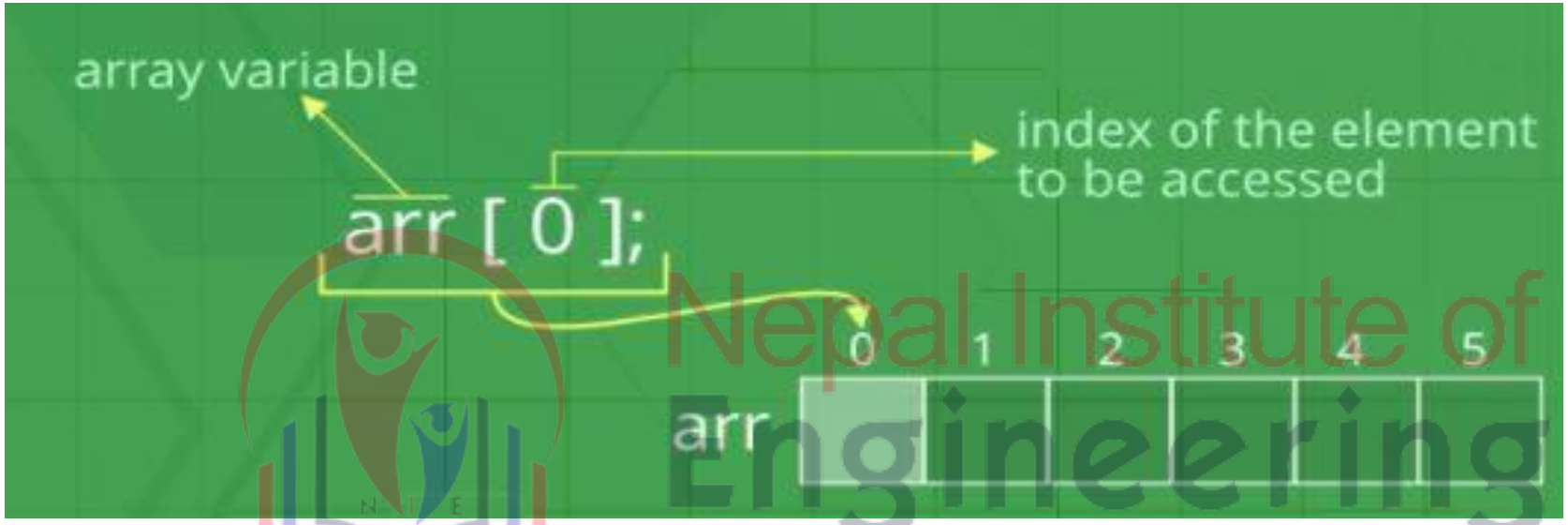
  Also, identical to: **int b[3] ={2,5,7};**

  iii. int c[8] = {45,56,78,44,23};

  here,array size is set to 8 but only 5 elements, so c[5], c[6], c[7] is **‘0’ zero** in this example.

• Accessing Array Elements

- Array elements are accessed by using an integer index.



- All elements of an array can neither be set at once nor one array may be assigned to others.
- For Example: int a[5], b[5];

```
a=0;   // Wrong  //
b=a; // Wrong //
if(a<b>
{…. // Wrong  //….}
```

Introduction

- Multi-Dimensional arrays are those which have more than one dimension.
- Are defined in same manner as one dimensional array, except a separate square bracket is required for each dimension or index
- The Two Dimensional array is also called matrix.

Declaration & Initialization of two-dimensional array

- Just like 1-D arrays, it can be declared as:

  data_type array_name [row_size] [column_size];

  - **Example:**

    int matrix [2] [3];

    float m [10] [20];

- Initialization:
  - **Example:**

    int marks [2] [3]={ {2,4,6},{8,10,12} };

- is equivalent to: marks[0] [0]=2; marks [0][1]=4; marks[0][2]=6;
  marks[1][0]=8; marks[1][1]=4; marks[1][2]=12;

Accessing 2-D array elements

- A two – dimensional array can be seen as a table with ‘x’ rows and ‘y’ columns
- Where, the row number ranges from 0 to (x-1) and,
- column number ranges from 0 to (y-1).
- A two – dimensional array ‘x’ with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

String Handling functions

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strncpy()	strncpy(string1, string2, 5)	Copies first 5 characters string2 into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strncat()	strncpy(string1, string2, 4)	Appends first 4 characters of string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strncmp()	strncmp(string1, string2, 4)	Compares first 4 characters of both string1 and string2
strcmpi()	strcmpi(string1,string2)	Compares two strings, string1 and string2 by ignoring case (upper or lower)
stricmp()	stricmp(string1, string2)	Compares two strings, string1 and string2 by ignoring case (similar to strcmpi())
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strdup()	string1 = strdup(string2)	Duplicated value of string2 is assigned to string1
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strnset()	strnset(string1, 'B', 5)	Sets first 5 characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

1. strlen( ):

•The length of string is the number of characters present in it, excluding the terminating null character.

•Its syntax is

```
integer_variable = strlen(string);
```

2. strcpy( ):

•The syntax is

```
strcpy(destination_string, source_string)
```
;

i.e. strcpy(s1, s2) ; means the content of s2 is copied to s1.

3.strcat( ):

•Its syntax is:

```
strcat(string1, string2);
```

i.e. string1 = string1 + string2 ;

4. strcmp( ):

•This function accepts two string as parameters and returns an integer whose value is

•1) less than 0 if the first string is less than the second

•2) equal to 0 if both are same

•3) greater than 0 if the first string is greater than the second

•Its syntax:

```
integer_variable = strcmp(string1, string2);
```

Thank You!!!