Lecture 1

INTRODUCTION:

Discussion on Course Plan, Marking Criteria and other staffs.

IMPORTANCE OF PROGRAMMING/WHY PROGRAMMING?

- A computer is a Dumb Tool, has no analyzing power.
- Human Beings are good at analyzing but get easily bored with repetitive works.
- Efficiency.
- Avoiding error.
- Utilizing memory to save huge data.

Etcetera.

PROBLEM SOLVING STRATEGY:

1. Understanding the Problem“Clearly”.
2. Finding the Requirements and given Resources.
3. Finding the Solution of the problem.
4. Developing an Algorithm/Steps to follow to solve the problem.
5. Converting Algorithm into Code.

LECTURE 2

ALGORITHM: An Algorithm is an unambiguous specification of how to solve a class of problems. In other words, it is a procedure or formula for solving a problem, based on conducting a sequence of specified actions.

PROGRAMMING OR INSTRUCTING A COMPUTER TO SOLVE A PROBLEM: AN EXAMPLE

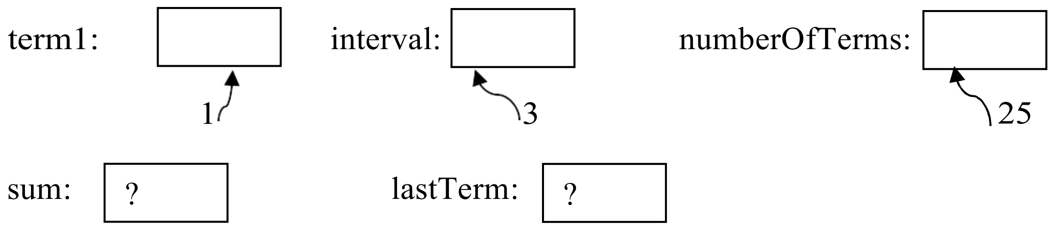
- Programs and Machines for Problem Solving
- Problem: Find the sum of a finite & equal interval series of integers (whole numbers), given the 1st term, the interval and the number of terms.
- **Major Steps of Development of a Program**

I. Analysis of the Problem and required Resources

** Say, 1st term = 1, interval = 3, number of terms = 25;
That is, the series is 1, 4, 7, 10, …, x.

** The sum may be computed as '((1st term + last term) x number of terms) / 2', where the last term will be '1st term + (number of terms – 1) x interval'.
That is, x = 1 + (25 – 1) x 3.

** Memory required:



II. Steps to Solve the Problem or the Algorithm:

1. Ask for the 1st term.
2. Read and remember the 1st term.
3. Ask for the interval.
4. Read and remember the interval.
5. Ask for the number of terms.

6. Read and remember the number of terms.
7. Calculate the required values as follows:

   $\text{lastTerm} = \text{term1} + (\text{numberOfTerms} - 1) \times \text{interval}$, and

   $\text{sum} = ((\text{term1} + \text{lastTerm}) \times \text{numberOfTerms}) / 2.$

8. Display the sum.

III. Converting the Algorithm into Code:

a. The Algorithm presented using Statements of the Language C:

```
int term1, interval, numberOfTerms;
int sum, lastTerm;

printf("Enter the first term:");
scanf("%d", &term1);

printf("Enter the interval:");
scanf("%d", &interval);

printf("Enter the number of terms:");
scanf("%d", &numberOfTerms);

lastTerm = term1 + (numberOfTerms-1) * interval;
sum = ((term1 + lastTerm) * numberOfTerms) / 2;

printf("The sum is: %d", sum);
```

Variable declaration
(Memory reservation, int for integer)

Output statement
Input statement

Calculation and assignment of values to variables (remembering)
Output statement

LECTURE 3

III. CONVERTING THE ALGORITHM INTO CODE:

a. The Algorithm presented using Statements of C Language:

```
int term1, interval, numberOfTerms;
int sum, lastTerm;

printf("Enter the first term:");
scanf("%d", &term1);

printf("Enter the interval:");
scanf("%d", &interval);

printf("Enter the number of terms:");
scanf("%d", &numberOfTerms);

lastTerm = term1 + (numberOfTerms-1) * interval;
sum = ((term1 + lastTerm) * numberOfTerms) / 2;

printf("The sum is: %d", sum);
```

Variable declaration (Memory reservation, int for integer)

Output statement
Input statement

Calculation and assignment of values to variables (remembering)
Output statement

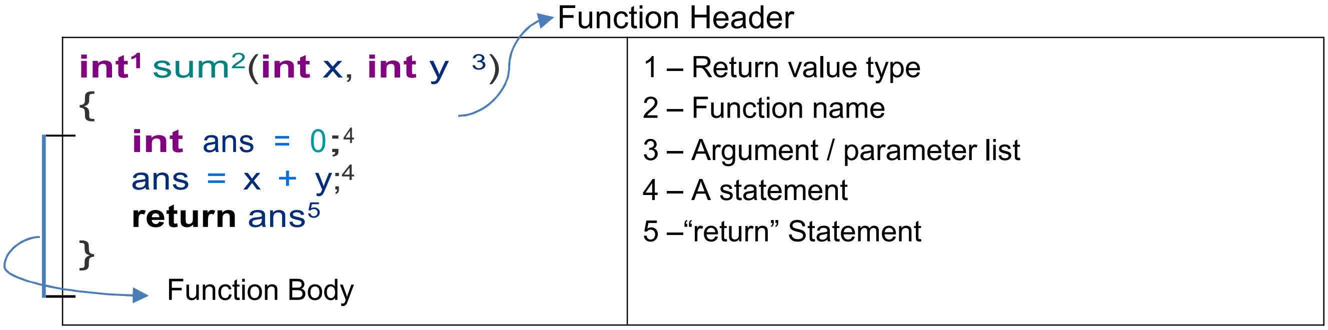
FUNCTION:

A function is a named block of code that can be called by other parts of a program. It performs a task and then returns control to the caller. “Function” is also known as “subroutine", "subprogram", "procedure", or "method" in other programming languages. Functions are made for code reusability and for saving time and space.
Functions are the building blocks of C. Remember that, every C program must contain at least one function, which is **main()**.

STRUCTURE OF FUNCTION:

There are two main parts of a function.

i. Function Header
ii. Function Body



Function Header

int^{1} sum^{2}(int x, int y ^{3})
{
    int ans = 0;^{4}
    ans = x + y;^{4}
    return ans^{5}
}

Function Body

1 – Return value type
2 – Function name
3 – Argument / parameter list
4 – A statement
5 –“return” Statement

Function Header: It consists of three main parts: *return value type*, *function name* and *parameters*/arguments of the function enclosed in parenthesis.

Function Body: It starts with the opening curly braces ({) and ends with the closing curly braces (}) after the function header. It contains a collection of statements that define what the function does.

III. b. C Program:

```
#include<stdio.h>

int main(void)
{
      int term1, interval, numberOfTerms;
      int lastTerm, sum;

      printf("Enter first term: ");
      scanf("%d", & term1);

      printf("Enter series interval: ");
      scanf("%d", &interval);

      printf("Enter no of terms: ");
      scanf("%d", & numberOfTerms );

      lastTerm= term1+( numberOfTerms -1)*interval;
      sum=(( term1+lastTerm)* numberOfTerms )/2;

      printf("\nSum of the series: %d.\n", sum);

      return 0;
}
```

LECTURE 4

VARIABLE:

- A variable represents a memory location that stores a specific type of 'data' or changeable 'value' accessed through a name.
- A variable requires to be declared before it is used in a statement.
- Naming convention:
  a-z, A-Z, 0-9 or '_', may be repeated
  No digit at first place
  Case sensitive

** This variable naming convention is valid for names of Functions and Constants as well.

CONSTANT:

- 'A', 'a', 25, 3.14,…
- There are other constants like octal or hexadecimal constants.
- May have names and may be used in expressions through names.

EXPRESSION:

- Meaningful sequence of operators and operands
- Arithmetic operators and names of variables of appropriate data type, or constants may constitute an expression
- A variable name, or a function name with its list of parameters, which may be empty, may be an expression

term1, sum_xy(5, 6.2)+100, term1 + (numberOfTerms – 1) * interval, char_f1(), …

STATEMENT:

- A sentence in C ending with ';'
- Complete executable instruction
- Contains expressions
- Various types
  - Declaration of variables

    int term1, x;   float y, sum1 = 2.5, z; ….

  - Assignment

    lastTerm = term1 + (numberOfTerms - 1) x interval;
    variableName = expression;   [= assignment operator]

- Return

  return 0; return funcVal;

  return expression; return 'value';

- Function prototype declaration

  float sum_xy(float x, float y);

- Function call

  odd_even(n);

BASIC DATA TYPES:

Keyword	Type	Output format specifier	Examples
int	signed whole numbers	%d	2569, -567, 1005, 0
char	character data	%c	A, b, +, /, …
float	floating-point numbers	%f	2.56, 1e-2, 0.003
double	double precision floating-point numbers	%f	2.56, 5.0006, 25E5
void	without any value		

*For double, %lf is also allowed as output format specifier but the 'l' is specified as having no effect if followed by 'f'.

*Input format specifiers are the same, except for double, where it is %lf.

KEYWORDS:

- Reserved predefined words; Very limited in number, 32 to be exact
- Have special meaning to the compiler
- Part of the syntax and must not be used as identifiers (names) of variables, constants or functions
- Must be written in lowercase
- Encountered so far:
  int, float, double, char, void, return.

**'include' is not a keyword; 'Int' or 'INT' is not a keyword as well.

LECTURE 5

CHARACTER SET:

- English characters and digits: A-Z, a-z, 0-9
- Special symbols: #, %, +, -, *, /, (,), {, }, „ ;, =, _, &, :, ., etc.

COMMENTS IN A PROGRAM:

- To make easy to interpret later:
  - Single line comment: //variable declaration
  - Multiple line comment: /* It is a function that returns the product of two integers. */

ARITHMETIC OPERATORS:

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+	Add two operands.	A + B = 30
-	Subtracts second operand from the first	B – A = 10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

- % can take only integers as operands.
- Be careful of integer division: **5/2** results in **2**!!
- - (subtraction) may also be used as unary minus
- Precedence (High-Low): [*, /, %], [+, -] (*left-to-right*); Suggested to put in parentheses.
  *a+b%c*d* is equivalent to *a+((b%c)*d)*

INCREMENT & DECREMENT OPERATOR:

Increment Operator

The statement

Count = Count +1;

can be replaced with

Count++; or *++Count;*

but after

Count=5; *Num = 4 * Count++;*

we get *Count: 6* *Num: 20*

whereas after *Count=5;* *Num = 4 * ++Count;*

we get *Count: 6* *Num: 24*

***Decrement Operator** has similar forms and functions, and we can use Count-- and --Count in a similar fashion.

EXERCISE: Write a C program to enter base and height of a right-angled triangle and find its hypotenuse and area. Note that, area = (1/2) * base * height

hypotenuse^2 = base^2 + height^2

Sample Input	Sample Output

Enter base: 3
Enter height: 4
	Hypotenuse = 5.00
	Area = 6.00

*Develop the algorithm first. Then write the C program to solve this problem.

LECTURE 6

TYPES OF FUNCTION:

i. Library Function
ii. User-defined Function

Library Function: Standard Library Functions or simply Library Functions are inbuilt function in C programming. The prototype and definitions of the functions are present in their respective *Header Files* and must be included in the program to access them.

For example, if we want to use *printf()* library function, the header file *stdio.h* should be included in the program.

User-defined Function: User-defined Functions (UDF) are those functions which are defined by the user at the time of writing a program. A program can be made of one or more user-defined functions beside *main()* function.

HEADER FILE: A header file is a file with extension *.h* which contains C function declarations and macro definitions to be shared between several source files. We request to use a header file in our program by including it by C preprocessing directive *#include*.

**Macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.

COMMON MATHEMETICAL FUNCTIONS CONTAINED IN THE HEADER FILE <math.h>:

- cos(0) = 1.000000
- ceil(0.001) = 1.000000 [*ceil(x) = n, n - nearest integer ≥ x*]
- floor(2.999) = 2.000000 [*floor(x) = n, n - nearest integer ≤ x*]
- log(6) = ……. [*Natural logarithm*]
- log10(1000) = 3.000000 [*Decimallogarithm*]
- pow(0.1, 3) = 0.001
- fabs(-5.3) = 5.300000
- sqrt(4) = 2.000000

……………

……………

GENERAL FORM OF A MULTIPLE FUNCTION C PROGRAM:

```
/* Inclusion of Header files */
-----------------------------


/* Declaration of Prototypes for user–defined functions f1, f2, … */
----------------------------


/* Definition of the ‘main’ function */
 int main(void)
 {

           /* Statements including
         return 0; */

 }


/* Definition of the function f1*/
return_value_type  f1(Parameter_List)
{
       /* Statements */
}


/* Definition of the function f2 */
return_value_type  f2(Parameter_List)
{
       /* Statements */
}

-----------------------------------
-----------------------------------
-----------------------------------
```

AN EXAMPLE OF MULTIPLE FUNCTION C PRPGRAM:

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

float sum_xy(float x, float y);
double prdct_xy(float x, float y);

int main(void)
{
  float n1, n2, n3;
  double n4, n5;

  printf("\nEnter two decimal fractions:");
  scanf("%f %f", &n1, &n2);

  n3 = sum_xy(n1, n2);
  n4 = prdct_xy(n1, n2);
  n5 = sqrt(n4);

  printf("\nThe 1st function returns %f.", n3);
  printf("\nThe 2nd function returns %f.", n4);
  printf("\nThe square root of %f is %f.\n", n4, n5);

  return 0;
}

float sum_xy(float x, float y)
{
    float z;
    z = x + y;
    return z;
}

double prdct_xy(float x, float y)
{
    double z;
    z = x * y;
    return z;
}
```

Inclusion of Header files

Function prototype declarations

main function

sum_xy user-defined function

prdct_xy user-defined function

LECTURE 7

LOGICAL OPERATORS:

- These operators are used to perform logical operations on the given expressions.
- There are 3 logical operators in C, those are logical *AND (&&)*, logical *OR (||)* and logical *NOT (!)*.
- Each of them returns 0 (false) or 1 (true).

Operators	Example/Description
&& (logical AND)	(x>5) && (y<5) It returns true when both conditions are true.
|| (logical OR)	(x>=10) || (y>=10) It returns true when at-least one of the condition is true.
! (logical NOT)	! ((x>5) && (y<5)) It reverses the state of the operand "((x>5) && (y<5))". If "((x>5) && (y<5))" is true, logical NOT operator makes it false.

Truth Table:

x	**y**	**x && y**	**x || y**	**!x**	**!y**	**!(x && y)**	**(x || y) && !(x && y)**
0	0	0	0	1	1	1	0
0	1	0	1	1	0	1	1
1	0	0	1	0	1	1	1
1	1	1	1	0	0	0	0

RELATIONAL (COMPARISON) OPERATORS:

- Relational operators are used to find the relation between two variables i.e. to compare the values of two variables in a C program.
- All are binary.
- Each return either 0 (false) or 1 (true).

Operators	Example/Description
>	x > y (x is greater than y)
<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)

==	x == y (x is equal to y)
!=	x != y (x is not equal to y)

ORDER OF PRECEDENCE OF OPERATORS:

Arithmetic Operators:

Highest	++	--	[Prefixed]
	- [unary]		
	*	/	%
Lowest	+	-	

Relational and Logical Operators:

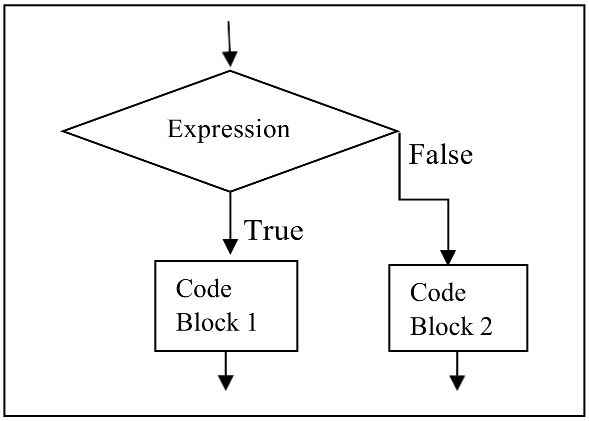
Highest	!	[unary]		
	>	>=	<	<=
	==	!=		
	&&			
Lowest	||			

**Arithmetic operators have higher precedence than relational or logical operators. The best way is to use parentheses.

SIMPLE SELECTION STATEMENT OR CONDITIONAL STATEMENT:

A. Most General Form

```
if (Expression)
        Code Block 1
else
        Code Block 2
```



if, else – keywords
Expression – valid C expression called the test
Code Block 1 – target of 'if'
Code Block 2 – target of 'else'

[A code block at any place can be replaced with a single statement.]

- Commonly, the expression contains a comparison or relational operator like >, <, ==, etc., that makes it to evaluate to true (1) or false (0).
- Logical operators like && (AND), || (OR) or ! (NOT) may also be used in combination with relational operators.
- In general, an expression that evaluates to **0** is **'false'** and that evaluates to any **nonzero value** is **'true'**.

B. Some Simple Forms of Selection Statements

1. if (Expression) Statement
2. if (Expression) Statement 1
   else Statement 2
3. if (Expression) Code Block
   else Statement
4. if (Expression) Statement
   else Code Block
5. if (Expression) Code Block

C. An Example

Problem: Write a program to input an integer and print "Even Number" or "Odd Number" if that given number is even or odd respectively.

Solution:

1. Algorithm (Develop by yourself)
2. C program

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

   printf("Enter an integer number: ");
   scanf("%d",&intNum);

   if(intNum % 2 == 0)
       printf("\nEven Number.\n");
   else
       printf("\nOdd Number.\n");

   return 0;
}
```

LECTURE 8

PROBLEM: Write a program to input mark of a course and print “Acceptable Result” if the mark falls between 70 and 100, otherwise print “Result not Acceptable”.

SOLUTION:

*Develop the algorithm and draw the flow chart before writing the code.

```
#include<stdio.h>
int main(void)
{
    int mark;
    printf("Enter mark: ");
    scanf("%d",&mark);

    if(mark>=70)
    {
        if(mark<=100)
            printf("\nResult Acceptable.\n");
        else
            printf("\nResult not Acceptable.\n");
    }
    else
        printf("\nResult not Acceptable.\n");

    return 0;
}
```

*The following code is equivalent to the one written above:

```
#include<stdio.h>
int main(void)
{
    int mark;
    printf("Enter mark: ");
    scanf("%d",&mark);

    if(mark>=70 && mark<=100)
        printf("\nAcceptable Result.\n");
    else
        printf("\n nResult not Acceptable.\n");

    return 0;
}
```

ESCAPE SEQUENCES OR BACKSLASH CONSTANTS:

\n – new line	\t - horizontal tab
\a - alert	\v - vertical tab
\b - backspace	\" - double quote
\' - single quote	\\ - backslash etc.

ASCII:

ASCII, abbreviated from 'AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE', is a character encoding standard for electronic communication. It is a set of digital codes representing letters, numerals, and other symbols, widely used as a standard format in the transfer of text between computers.

*Search on the internet for the full ASCII table.

NESTED SELECTION STATEMENTS OR CONDITIONAL STATEMENTS:

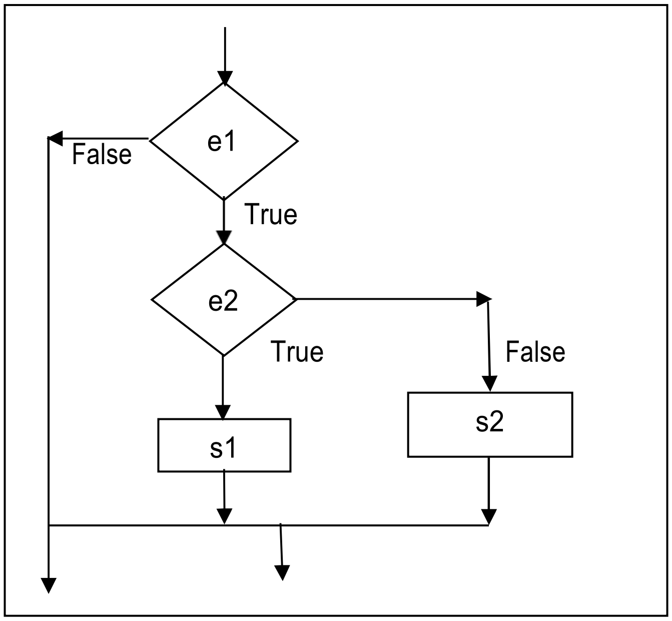
When an 'if' statement stands as the target of another 'if' statement in the 'if' part or in its 'else' part then it is called a nested selection statement.

A. Simple forms

i)

```
        if(e1)
           if(e2)
              s1
           else
              s2
```

Flowchart:



- e1, e2, … - Expression 1, Expression 2, …
- s1, s2, … - Statement 1, Statement 2, …
- Only one statement is executed.
- A statement at any place can be replaced with a code block.

ii) Another form

```
if(e1)
   s1
else
   if(e2)
      s2
```

iii) Another form

```
if(e1)
   if(e2)
      s1
   else
      s2
else
   s3
```

Problem: Write a program to classify, using nested if-else, a character entry into the following classes:

a. A lowercase letter;
b. An uppercase letter;
c. A digit;
d. I don't know.

3. EXERCISES:

1. Write a program that takes as input total marks obtained by a student in a course and displays the corresponding letter grade according to the following table.

Marks	Letter Grade
80-100	A
70-79	B
60-69	C
50-59	D
0-49	F

2. Write a program to classify, using nested if-else, a character entry into the following classes:
   A lowercase letter;
   An uppercase letter;
   A digit;
   I don't know.
3. Three numbers are input through keyboard. Write a program to find out the maximum and minimum of these three numbers.
4. Take a year as input and determine whether it is a leap year or not. [**Hint**: Check the input year is divisibility by 4 but not by 100 or by 400]

4. ASSIGNMENT 1:

1. Write a program to find whether a given number is even or odd. If found even, check for divisibility by 4, otherwise check for divisibility by 3.
2. Write a program to input two angles of a triangle and check whether the triangle is a right-angled triangle or not.
3. Write a program to input a letter and display it in opposite case, i.e., if the given letter is in upper case, display it in lower case and vice-versa.
4. According to Gregorian calendar, it was Monday on the date 01/01/1900. If any year is input through the keyboard write a program to find out what is the day on 1st January of this year.

LECTURE 10

LOOP:

In Computer Science, a loop is a programming structure that repeats a sequence of instructions until a specific condition is met.

Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. If it hasn't, the next instruction in the sequence is an instruction to return to the first instruction in the sequence and repeat the sequence. If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

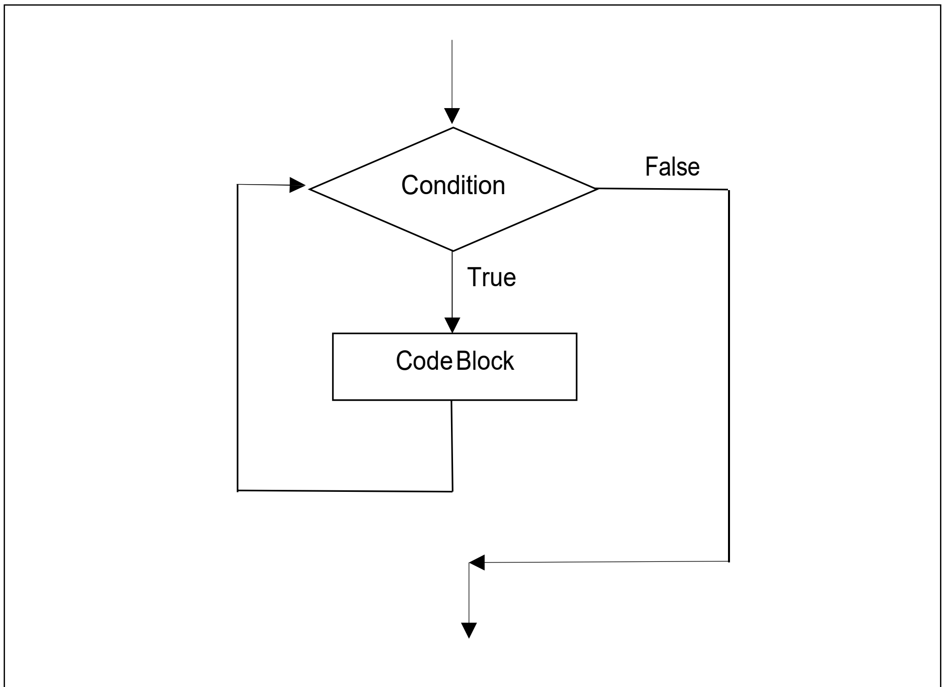


Fig: Flow Diagram

TYPES OF LOOP:

In C, there are three (3) types of loop which are:

(i) while
(ii) do-while loop
(iii) for loop.

“while” LOOP:

while is one of the three types of loop in C. The syntax of while loop in C programming language is:

```
while(condition)
{
      Statement(s);
}
```

Here, **statement(s)** may be a single statement (in that case the curly braces can be skipped) or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, the program control passes to the line immediately following the loop.

The key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

The **while loop <u>cannot be written</u> without a **condition** (leaving the place for condition empty). It will cause a compilation error.

EXAMPLE:

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

int main(void)
{
    char ch;

    printf("Enter a character: ");
    ch=getche();

    while(ch!='q')
    {
        printf(", Not q.\n\nEnter a character again: ");
        ch=getche();
    }
    printf(", Found the q.\n");

    return 0;
}
```

INFINITE LOOP:

An infinite loop (sometimes called an *endless loop*) is a piece of coding that lacks a functional exit so that it repeats indefinitely.

EXAMPLE OF AN INFINITE LOOP:

```
#include<stdio.h>

int main(void)
{
    int cnt=1;

    while(cnt >= 0)
    {
        printf("\nCount : %d", cnt);
        cnt++;
    }

    printf(“\n\n Out of the loop.");
                                              /*this statement will
    return 0;                                 *never get executed.
}                                             */
```

Practice on loop(while)
Structure of do-while loop

LECTURE 11

PROBLEM: Write a C program to display the multiplication table of 5 in the following format.

```
5 * 1 = 5
5 * 2 =10
5 * 3 =15
………………
……………..
5 * 10 = 50
```

PROBLEM ANALYSIS:

To display the first row:	`printf(“5 * 1 = 5”);`
To display the second row:	`printf(“5 * 2 = 10”);`
To display the third row:	`printf(“5 * 3 = 15”);`

Here we see that we need to write the same statement (*printf*) for every line of output with some data changes in a pattern. So, we can easily put this statement within a loop replacing the inconstant data with variables and change the value of the variables in every iteration.

GENERAL STATEMENTS:

```
printf("5 * %d = %d\n",cnt, 5*cnt);
cnt=cnt+1;
```

C PROGRAM:

```
#include<stdio.h>

int main(void)
{
    int cnt=1;

    while(cnt <= 10)
    {
        printf("5 * %d = %d\n",cnt, 5*cnt);
        cnt=cnt+1; // this statement can be replaced by cnt++;
    }

    return 0;
}
```

Practice on loop(while)
Structure of do-while loop

PROBLEM: Write a C program to display the multiplication table of any integer given by user in the following format.

X * 1 = Y1
X * 2 = Y2
X * 3 = Y3
………………
……………..
X * 10 = Y10

C PROGRAM:

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

DO-WHILE LOOP:

```
    do
    {
            Statement(s);
    }while(condition);
```

**Note that semicolon after the closing parenthesis at the end of the loop.
**Rewrite the previous program using do-while loop.

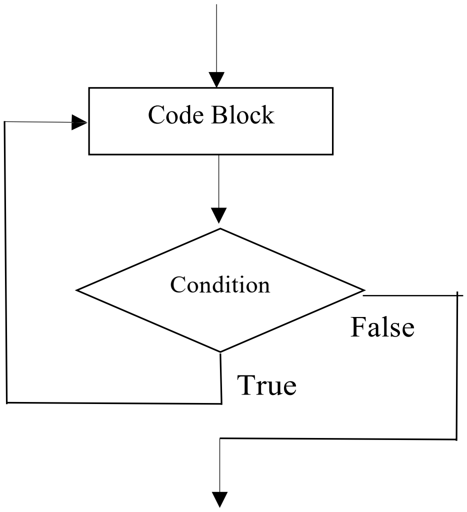


Fig: Flow Diagram

Practice on loop(do-while)
Structure of for loop

LECTURE 12

PROBLEM: Write a C program to display first ‘n’ odd natural numbers along with their sum and average using do-while loop.

SOLUTION:

```
#include<stdio.h>
int main(void)
{
    int n, i, sum=0, oddNum=1;
    printf("Enter n: ");
    scanf("%d", &n);
    i=n;
    printf("\nFirst %d odd natural numbers: ", n);
    do
    {
        printf("\t%d", oddNum);
        sum+=oddNum;
        oddNum+=2;
    }while(--i>0);
    printf("\n\nSum = %d \t Average = %.2f\n", sum, sum/n*1.0);
    return 0;
}
```

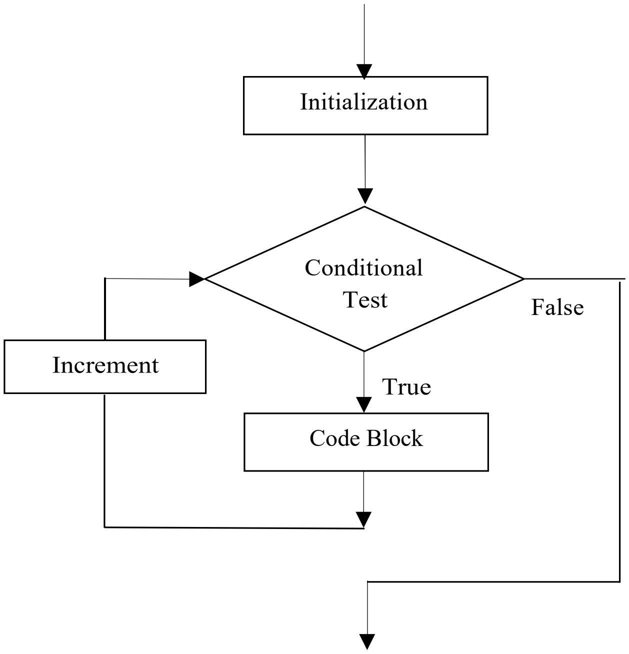
‘FOR’ LOOP:

```
for(initialization; conditional-test; increment)
{
    Statement(s);
}
```

Initialization section is used to give an initial value to the variable that controls the loop. This variable is usually referred to as the *loop-control variable*. The initialization section is executed only once before the loop begins.

Conditional-Test portion of the loop tests the loop-control variable against a target value. If the conditional test evaluates true, the loop repeats. If it is false, the loop stops, and program execution picks up with the next line of code that follows the loop. The conditional test is performed at the start or the top of the loop each time the loop is repeated.

Increment portion of the for is executed at the bottom of the loop. That is, the increment portion is executed after the statement or block that forms its body has been executed. The purpose of the increment portion is to increase (or decrease) the loop-control value by a certain amount.



Three types of loop with
a common example

LECTURE 13

PROBLEM: Develop a program to find the n^{th} term ($n \geq 2$) of the recurrence relation,
$Term_n = Term_{n-1} + Term_{n-2}$,
given that $\underline{Term_0 = 0}$ and $\underline{Term_1 = 1.}$

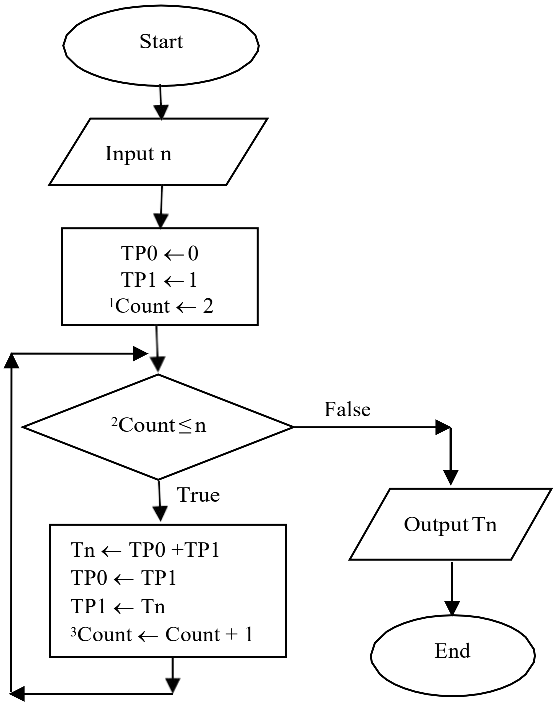
RESOURCES AND ALGORITHM:

Series									
	Index: 0	1	2	3	4	5	6	7	...
	Term: 0	1	1	2	3	5	8	13	...

Pseudo code

```
Input n
TP0 ← 0
TP1 ← 1
[1]Count ← 2
While ([2]Count ≤ n) repeat
        Tn ← TP0 + TP1
        TP0 ← TP1
        TP1 ← Tn
        [3]Count ← Count + 1
Output Tn
```

Flowchart Diagram



Verification:

n: 5 TP0: 0 TP1: 1

Count	Tn	TP0	TP1
2	1	1	1
3	2	1	2
4	3	2	3
5	5	3	5
6			

Three types of loop with
a common example

PROFRAM FRAGMENT WITH ‘WHILE’ LOOP:

```
scanf(“%d”, &n);

TP0=0;
TP1=1;
Count=2;

while(Count<=n)
{
        Tn = TP0 + TP1;
        TP0 = TP1;
        TP1 = Tn;
        Count = Count + 1;
}
printf(“%d”, Tn);
```

PROFRAM FRAGMENT WITH ‘FOR’ LOOP:

```
scanf(“%d”, &n);

for(TP0=0, TP1=1, Count=2; Count<=n; Count=Count+1)
{
        Tn = TP0 + TP1;
        TP0 = TP1;
        TP1 = Tn;
}
printf(“%d”, Tn);
```

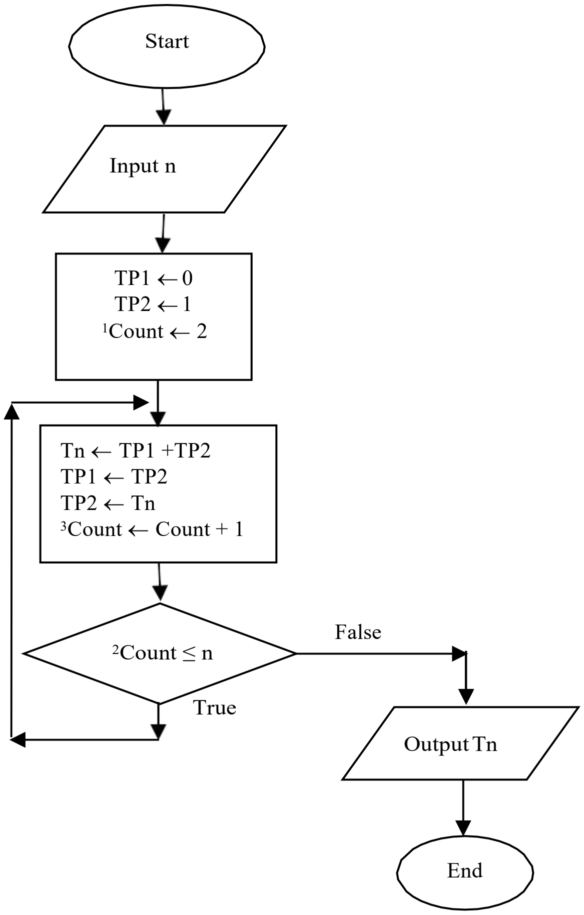
ANOTHER VERSION OF ‘FOR’ LOOP:

```
scanf(“%d”, &n);

for (TP0=0, TP1=1, Count=2; Count<=n; Tn=TP0+TP1, TP0=TP1, TP1=Tn,
Count=Count+1);
printf(“%d”, Tn);
```

PROGRAM FRAGMENT WITH ‘DO-WHILE’ LOOP:

Let’s redraw the flowchart diagram in the following way.



Code segment using ‘do’-‘while’ loop:

```
scanf(“%d”, &n);

TP0=0;
TP1=1;
Count=2;
do
{
        Tn = TP0 + TP1;
        TP0 = TP1;
        TP1 = Tn;
        Count = Count + 1;
}
while (Count <= n);
printf(“%d”, Tn);
```

Flexibility of ‘for’ loop,
Structure of ‘switch’ with example

LECTURE 14

FLEXIBILITY OF ‘FOR’ LOOP:

1.
```
for(i=1; i <= 5; printf(“%d ”, i++));
```
Output: 1 2 3 4 5

2.
```
for(i=1, j= 5; i <= 5; printf(“%d ”, j*i++));
```
Output: 5 10 15 20 25

3.
```
for(i=1, j= 5; i <= 5; printf(“%d ”, i*j), i++, j--);
```
Output: 5 8 9 8 5

4.
```
for(i=1, j= 10; i <= 5, j>7; printf(“%d ”, i), i++, j--);
```
Output: 1 2 3

```
for(i=1, j= 10; i <= 5, j<7; printf(“%d ”, i), i++, j--);
```
Output:

***‘<u>Test</u>’ is supposed to be a single expression, but if contains more, only the last one is expected to be considered.*

5.
```
for(; ; printf(“\nInfinite loop”));
```
Output:

```
Infinite loop
Infinite loop
Infinite loop
.......
```

6.
```
for(printf(“To repeat, enter any character except q: ”), c=getche();
c!=’q’; printf(“\nAgain! ”), c=getche());
```
or
```
for(printf(“To repeat, enter any character except q: ”), c=getche();
c!=’q’; )
{
   printf(“\nAgain! ”); c=getche();
}
```

Output: To repeat, enter any character except q:x

```
Again! y
Again! a
Again! 5
....
Again! q                    [ q for ‘quit’]
```

EXERCISE: Rewrite the above code snippets using ‘while’ loop.

SELECTION OR CHOOSING FROM ALTERNATIVES USING 'SWITCH' STATEMENT:

General form of 'switch' statement

```
switch(Expression)
{
        case Constant 1:
                Statement 1
                Statement 2
                …..
                break;
        case Constant 2:
                Statement 1
                Statement 2
                …..
                break;
        …….
        default:
                Statement 1
                Statement 2
                …..
 }
```

1. Keywords: *switch*, *case*, *default*, *break*
2. '**:**' is required as shown
3. '*Expression*' evaluates to an ***integer*** or a ***character*** only;
4. *Constant1*, *Constant2*, … an **integer** or a **character** constant
5. The value of '*Expression*' is matched (equaled) to a *Constant* to choose the sequence of statements that follows;

   The 'break' then takes to the end of 'switch'

- It is used to simplify lengthy and cumbersome if-else chains.
- It's like a 'call-center' that links a user to some service or person according to the 'call'.
- 'break' and 'default' parts may be absent.
- Statements under a 'case' do not form a block.
- A 'case' may not have any statement.
- Constants are supposed to be distinct.
- A Statement may also be of 'switch' type, allowing nesting.

EXAMPLES:

i)

```
int  i; float j, k, l;

printf("\nEnter two numbers:”);
scanf("%f%f", &j, &k);

printf("\nEnter 1(add), 2(subtract) and anything else for no action:”);
scanf("%d", &i);

switch (i)
{
        case 1:         l= j+k;
                        printf("\nThe sum: %f”, l);
                        break;

        case 2:         l= j-k;
                        printf("\nThe difference: %f”, l);
                        break;

        default:
                        printf("\nNo action!!”);
}
```

ii)

```
char c1;
printf("\nEnter a letter:”);

switch(tolower(c1=getche()))
{
        case ‘a’:
        case ‘e’:
        case ‘i’:
        case ‘o’:
        case ‘u’:
                        printf(" is a vowel.”);
                        break;
        default:
                        printf(" is a consonant.”);
}
```

** ‘**tolower()**’ may need **ctype.h** header file.

LECTURE 15

A. ARRAY

- An array is a list of variables to store a list of data of same type, which are accessed through indexing for collective use.
- There is various usage of arrays like storing of group of data for repeated use, searching in a data space, getting minimum or maximum of a data set, getting an ordered sequence of data, etc.
- Arrays may be one-dimensional as well as of more dimensions.

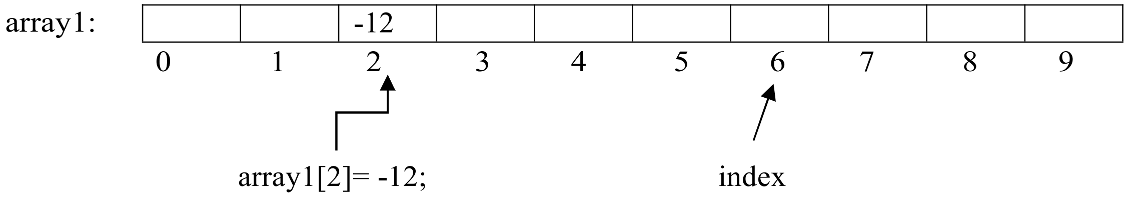
B. DECLARATION AND INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

```
DataType ArrayName[ArraySize];
```

Here, DataType – type of the ‘element’s (data) stored
ArrayName – name of the array
ArraySize – number of ‘element’s stored in the array

Example:

```
int array1[10];
```



array1:

		-12							
0	1	2	3	4	5	6	7	8	9

This is just a list of 10 variables.

`array1[0], array1[1], …, array1[9],` to store ten integers.

As it is a small array, it could be initialized during declaration as follows:

```
int array1[10] = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};
```

or using the following code:

```
int i, array1[10];
for(i=0; i<10; i++)
   array1[i] = (i+1) * 3;
```

array1:

3	6	9	12	15	18	21	24	27	30
0	1	2	3	4	5	6	7	8	9

C. USING ONE-DIMENSIONAL ARRAYS

Inserting data into an array, displaying all the data and searching for a given data in it can be arranged with the code that follows:

```
int e, i, j, array1[10];
for(i=0; i<10; i++)
{
     printf(“\n Enter an element:”);
     scanf( “%d”, &array1[i]);
}

printf(“\n The elements of the array are:”);
for(i=0; i<10; i++)
     printf( “%d,”, array1[i]);

printf("\nEnter element to search for:");
scanf("%d", &e);

j=0;
for(i=0; i<10 && !j; i++)
   if (array1[i]==e)
   {
      j=i+1;
   }
if (j==0)
   printf("\nNot found!");
else
   printf("Found at position %d.", j);
```

- Mark that the memory is accessed individually, say,

```
     scanf(“%d”, &array1[6]);
```

or

```
     printf(“%d,”, array1[8]);
```

- Like any variable the value of an element can be reassigned, for example.

```
     array1[6] = 156;
```

- Mind that, as no bound checking is done automatically, one may not be prohibited from reading or writing beyond array size, say, array1[10], but it will be a mistake.

LECTURE 16

PROBLEM 1: Write a C program to find the maximum element of an unordered array of floating-point numbers.

SOLUTION:

```
#include<stdio.h>

int main(void)
{
    float arf[5], max;     //assuming array size is 5
    int i;

    for(i=0; i<5; i++)
    {
        printf("Enter value for location %d: ", i+1);
        scanf("%f", &arf[i]);
        if(i==0)
            max=arf[i];
        else
            if(arf[i]>max)
                max=arf[i];
    }
    printf("\nThe largest value: %f\n",max);

    return 0;
}
```

PROBLEM 2: Write a C program to *find the total vowel count* in a one-dimensional array of characters.

SOLUTION:

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

int main(void)
{
    char a[5];              //assuming array size is 5
    int i,count=0;

    for(i=0; i<5; i++)
    {
        printf("\nEnter character for location %d: ", i+1);
        a[i]=getche();
        if(a[i]=='a'||a[i]=='e'||a[i]=='i'||a[i]=='o'||a[i]=='u')
            count++;
        else if(a[i]=='A'||a[i]=='E'||a[i]=='I'||a[i]=='O'||a[i]=='U')
            count++;
    }
    printf("\n\nTotal vowel count: %d\n",count);
    return 0;
}
```

PROBLEM 3: Write a C program to delete *a given element* from a one-dimensional array of integers.

SOLUTION:

```c
#include<stdio.h>

int main(void)
{
    int ari[5], element, i, index;  //assuming array size is 5

    for(i=0; i<5; i++)
    {
        printf("Enter integer for location %d: ", i+1);
        scanf("%d",&ari[i]);
    }
    printf("\nGiven List of integers: ");
    for(i=0; i<5; i++)
        printf("%d ",ari[i]);

    printf("\n\nEnter element to be deleted: ");
    scanf("%d",&element);

    for(i=0; i<5; i++)
        if(element==ari[i])
            index=i;
    for(i=index; i<4; i++)
        ari[i]=ari[i+1];

    printf("\nList after deletion: ");
    for(i=0; i<4; i++)
        printf("%d ",ari[i]);

    return 0;
}
```

LECTURE 17

A. In C, *strings* are defined and processed as arrays of characters. May be declared, for example, as follows:

char student_name[40];

i) In *student_name* maximum length of string is 39, the last place in this case is filled up with a null. Null is a zero written as '\0'. We can store a student's name of shorter length, say, 15, when the 16th place, that is, *student_name[15]* must be filled in with a null.

ii) Standard input function **gets()** inserts a null automatically at the end when input ends with pressing 'enter'. [*gets(student_name);*]

iii) Standard output function **puts()** displays up to null. [*puts(student_name)*; Other possible ways of output: *printf(student_name); printf("Student Name:%s", student_name);*]

iv) A string is thus a null terminated character array.

v) Mind that gets() does not perform a bounds check!! Specify a size big enough during declaration.

** Reading and storing characters in a character array and making that array a string. And then printing that string without using any library functions from **string.h**

```
#include<stdio.h>

int main(void)
{
    char ar[20];
    int i, flag=1;

    printf("Enter characters (<20): ");
    for(i=0; i<19 && flag; i++)
    {
        ar[i]=getchar();
        if(ar[i]=='\n')   //ASCII of ‘\n’: 10
            flag=0;
    }
    ar[i-1]='\0';

    printf("Printing String: ");
    for(i=0; ar[i]!='\0'; i++)
        putchar(ar[i]);

    return 0;
}
```

** Now re-write the code for abovementioned scenario using library functions from **string.h**

Comparison between some similar library functions:

getc() and getchar(): Both functions read a character from user but the difference between getc() and getchar() is getc() can read from any input stream, but getchar() reads from standard input. So, gtechar() is equivalent to getc(stdin).

getch() and getche(): Both functions read a character from console but the difference is getch() does not echo to the screen where getche() does. Both these functions are non-standard library functions.

LECTURE 18

Problem 1: Write a program to input user's name and current time, and then greet the user with appropriate message. [Good Morning (00-11)/Good Afternoon (12-17)/Good Evening (18-23)]

Problem 2: Write a C program to print a sentence taken from user in reverse.

A. String related standard library functions included in **string.h:**

a) **strlen(str1)** – returns the number of characters in str1, without null.

b) **strupr(str1) -** converts the content of str1 to uppercase and returns the result.

c) **strlwr(str1) -** converts the content of str1 to lowercase and returns the result.

d) strcpy(to, from)
Copies and returns the content of 'from' to 'to'; both are string arguments.
[to: Hi from: You] will be [to: You from: You]

e) strcat(to, from)
Concatenates the content of 'from' to the end of 'to' and returns the result.

f) strcmp(str1, str2)
Returns 0 if str1 and str2 matches absolutely, -1 if in str1 a character with lesser ASCII code is found first in character by character comparison (lexicographically, that is, in dictionary order), and otherwise, 1. The comparison is case sensitive! Remember that lowercase letters have greater ASCIIcodes.

g) **strchr**(str,ch) – 'true' if ch is in str.

h) **strstr**(str2, str1) –'true' if str1 is in str2.

B. Conversions:

String(array) to int:

```
int i1 = atoi(str1);
```

String(array) to float:

```
float f1 = atof(str1);
```

- For conversion **stdlib.h** header file is required.

LECTURE 19,20

NESTED LOOPS:

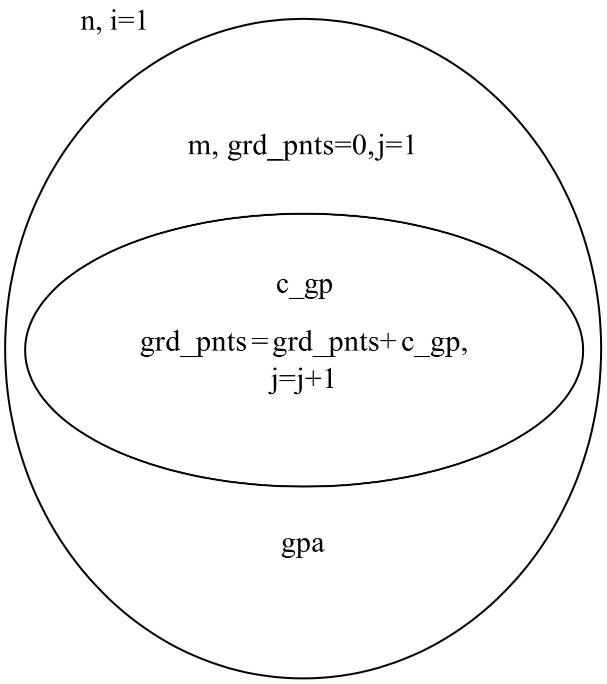
- If a loop L_1 contains in its body another loop, L_2, then L_2 is called nested in L_1.
  L_1 - outer loop, L_2 –inner loop

- **Example:**

  Problem: Develop a program to display the GPA obtained in an exam session by a batch of students.

  n- number of students
  m- number of courses taken by each student
  grd_pnts – total grade points obtained by a student
  c_gp – grade point obtained in a course
  gpa- grd_pnts /m

  Outer loop for students, Inner loop for courses of a student
  i, j – loop control parameters



** Flowchart Diagram

[--]

Possible major code segment:

```
int n, m, i, j;
float grd_pnts, c_gp, gpa;
printf("How many students?");
scanf("%d", &n);
i=1;
while(i<=n)
{
        printf("How many courses taken by std. no. %d?", i);
        scanf("%d", &m);
        grd_pnts=0;  j=1;
        while(j <= m)
        {
                printf("Enter the grade points obtained in course %d:", j);
                scanf("%f", &c_gp);
                grd_pnts = grd_pnts +c_gp;
                ++j;
        }
        gpa= grd_pnts/m;
        printf("GPA obtained by std. no. %d is %f.", i, gpa);
        ++i;
}
```

Practice at class:

	*	*	*	*		
	*	*	*	*		
	*	*	*	*		
	*	*	*	*		

	*					
	*	*				
	*	*	*			
	*	*	*	*		

				*		
			*	*		
		*	*	*		
	*	*	*	*		

Exercise:

			*			
		*	*	*		
	*	*	*	*	*	
*	*	*	*	*	*	*

			*			
		*		*		
	*				*	
*	*	*	*	*	*	*

*	*	*	*	*	*	*
	*	*	*	*	*	
		*	*	*		
			*			

TWO-DIMENSIONAL ARRAY:

- **An example**

  A 2-D chart for recording noontime temperature may look like the following table:

	0	**1**	**2**	**…**	**30**
0					
1			16.8		
2					
. .					
11					

12 rows for months of a given year

31 columns for days of a given year

16.8 – noontime temperature of 3rd February of a given year

- **Declaration and accessing:**

```
float    temp_chart2D[12][31];
temp_chart2D[2][3] = 26.9;
/* Noontime temperature of 4th March is assigned 26.9 degrees. */
 scanf(“%f”, &temp_chart2D[2][4]);
```

- **Sample code for processing:**

```
int i, j;
float sum=0, ar2D[2][3]={23.51, 12.5, 41.3, 16.35,26.75, 9.25};
for(i=0;i<3;i++)
    printf("\t%2d",i);     //+ right align, - left align
for(i=0;i<2;i++)
{
    printf("\n\n");
    printf("%-2d",i);
    for(j=0;j<3;j++)
    {
        printf("\t%-5.2f", ar2D[i][j]);
        sum=sum+ar2D[i][j];
    }
}
printf("\n\nThe sum of the elements: %5.2f", sum);
```

*How the output will look like?

Let's solve a problem:

Problem: Write a program to find the sum of the main/leading diagonal elements of a two-dimensional array of floating point numbers.

** \ Main, Major, Principal, Leading, Primary: Diagonal \
** / Minor, Counter, Secondary, Anti-, Counter: Diagonal /

Exercise:

$$\begin{array}{|c|c|c|c|}\hline 2 & 1 & 6 & 8 \\ \hline 4 & 6 & 0 & 3 \\ \hline 5 & 8 & 9 & 2 \\ \hline 3 & 1 & 1 & 5 \\ \hline 4 & 4 & 2 & 8 \\ \hline \end{array} \quad \text{X} \quad \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline 4 \\ \hline \end{array} \quad = \quad \begin{array}{|c|}\hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \end{array}$$

**Write a program to arrange data entry and to find the output matrix.

LECTURE 21

DISCONTINUATION OF LOOPS USING 'break' STATEMENT:

- To go out of a loop after execution of any of the statements of its body, the '*break*' statement may be used.
- It is simply written as

  break;

  where 'break' is a keyword.
- It helps when a loop needs to be discontinued under certain circumstances.
- **Example**: Showing an unspecified number of multiples of a given number.

```
int  i, n;
char c;

printf("\nEnter a positive integer:");
scanf("%d", &n);

i=2;
while (i)
{
    printf("\n%d\tMore? (Y for yes, anything else for no):", i * n);
    c = getche();
    if(c!='Y')
        break;
    ++i;
}
```

- In case of nested loops 'break' discontinues only that loop, in which it appears as a statement, not the outer loops; the inner loops are automatically skipped.

- **Another example**

```
int i,j,k;
for(i=1;i<=2;++i)
   for(j=1;j<=3;++j)
   {
      if(j%2 == 0)
         break;
      for(k=1;k<=2;++k)
         printf("\ni: %d\tj: %d\tk:%d",i,j,k);
   }
```

DISCONTINUATION OF LOOPS USING ‘continue’ STATEMENT:

- The ‘continue’ statement may be used after any of the statements in the body of a loop to go for next iteration without execution of the statements that follow it in that loop.
- It is simply written as

  continue;

  where ‘continue’ is a keyword.
- It helps when a loop needs to skip execution of some statements in the current pass under certain circumstances.
- In ‘while’ and ‘do-while’, *continue* takes to the ‘Test’, but in ‘for’ the ‘Renewal’ part is performed before going to the ‘Test’.
- **Example:** Ensuring entry from a given range of numbers.

```
int  i, n;
i=1;
while(i)
{
      printf("\nEnter a positive integer from 100 to 200: ");
      scanf("%d", &n);

      if(!(n>=100 && n<=200))
            continue;
      printf("\nThe valid number is %d. ", n);
      i=0;
}
```

LECTURE 22

ARRAY OF STRINGS:

A. List of Strings

- “List of Strings” is a two-dimensional “Array of Characters”.
- Declaration and accessing

```
char frndList[10][15];
```

10 strings, each of length 15, including the null.

0	1	2	3	…	9
modhu	shadhu	jadu	rhishi		hashi

```
gets(frndList[4]);          /* Reading the 5th string */
printf(frndList[4]);        /* Writing the 5th string */
```

- **Simple code for processing**

```
int i;
char strList[3][4];
for(i=0; i<3; ++i)
{
   printf("\nEnter string:");
   gets(strList[i]);
}
for(i=0; i<3; ++i)
   printf("\n%d\t%s", i+1, strList[i]);
```

** Re-write the code to input the strings reading one character at a time.

Solve the Problem: Write a C program to let the user enter 10 names and then display the names, one at a time, in any order according to user’s choice. To stop the program, user must enter a negative integer.

LECTURE 23

ARRAY OF STRINGS

B. Table of Strings

- Declaration and accessing

```
char frndList[3][50][15];
```

3 lists, each containing 50 strings of maximum 15 characters including null.

2 leftmost indices are used to access:

```
gets(frndList[2][1]);        /*Reading the 2nd string of the 3rd list*/
```

- Simple code for processing

```
int i, j;
char strTable[3][2][6];

for(i=0; i<3; ++i)
    for(j=0; j<2;j++)
    {
        printf("\nEnter string:");
        gets(strTable[i][j]);
    }

for(i=0; i<3; ++i)
{
    printf("\n%d", i+1);
    for(j=0; j<2;j++)
        printf("\t%s", strTable[i][j]);
}
```

MULTIPLE FUNCTION PROGRAM

A program, which has been written using a number of functions (at least 2, *main* function and another user defined function), is called a multiple function program.

General Structure:

- Inclusion of Header Files
- Function prototype declarations
- *main* function
- Other additional user defined functions

Example:

Write a multi-function program repeat the process of calculating the sum of a series of integers. First term, interval and number of terms of the series are user inputs.

Solution:

```
#include <stdio.h>
int CompSum(void);
int main(void)
{
    int Sum, Times, Count;
    printf(“\nHow many times repeated?”);
    scanf(“%d”, &Times);

    Count = 1;
    while(Count <= Times)
    {
        printf(“\nCalculation serial no.:  %d”,  Count);
        Sum = CompSum();
        printf(\nThe Sum is %d”, Sum);
        Count = Count + 1;
    }
    return 0;
 }

 int CompSum(void)
 {
   int Term1,Interval,NumberOfTerms;
   int FuncVal, LastTerm;

   printf("Enter the first term:");
   scanf("%d", &Term1);

   printf("Enter the interval:");
   scanf("%d", &Interval);

   printf("Enter the number of terms:");
   scanf("%d", &NumberOfTerms);

   LastTerm = Term1 + (NumberOfTerms - 1) * Interval;
   FuncVal = ((Term1 + LastTerm) * NumberOfTerms)/2;

   return FuncVal;
  }
```

LECTURE 24, 25

LOCAL VARIABLE AND GLOBAL VARIABLE

i) **Declaration:** Local variables are declared at the beginning of a code block, usually, at the beginning of the body of a function. *Formal parameters* of a function declared in the parameter list are also local variables with respect to the function.

   Global variables are declared outside any function definition, usually, at the beginning of a program, just before the 'main' function.

ii) **Scope:** The scope of a local variable is the code block, at the beginning of which it is declared. But the scope of global variables is all the functions of the program, that is, the whole program.

iii) **Lifetime:** Local variables are created upon entry into the function /block, and destroyed upon its exit, and so, cannot retain values between function calls.
   But global variables are maintained all through the runtime of the program.

iv) **Naming:** No restriction is imposed on naming local and global variables with the same name.
   But if a local variable takes the name of a global variable, the function just goes out of the scope of that global variable.
   On the other hand, local variables in different functions with the same name have no conflict at all.

v) **Initialization:** Local variables may be initialized with constants and values through variables or from function calls.
   But global variables may be initialized only with constants, usually, by default, with 0.

Example:

```
#include <stdio.h>

int f1(int x);
void f2(int x);

int v1;

int main(void)
{
   int i=5,j,k=10,n=30;
   v1=100;
   {
      int n=n+5;
   }
   f2(k);
   j=f1(i);
   printf("%d\t%d\t%d",j,v1,n);

   return 0;
}

int f1(int x)
{
    int j=200;
    v1=v1+x;
    return v1;
}

void f2(int x)
{
    int v1,l=500;
    v1=l+x;
}
```

Memory activities:

v1:

<u>main</u>

i: 5   j: k: 10   n: 30

...

Problem: Calculate the salary of an employee in a separate user-defined function taking basic salary and percentage of house rent and medical allowances as input in main function. The result must be displayed from the main function.

SIMPLE USE OF STATIC LOCAL VARIABLES

```
int add_amount(int new_amount)
{
        static int total_amount =100;
        total_amount = total_amount+ new_amount;
        return total_amount;
}
```

- A call

  `add_amount(25)`

  will return 125, and next call

  `add_amount(50)`

  will return 175.

*A static local variable is a local variable that retains its value between function calls. It functions like a global variable, but is known only locally. It is useful for storage without conflict by multiple users.

Type Conversion (automatic):

- Mixing in an expression
  General principle: Operand by operand conversion is done to the maximum sized operand, if not otherwise restricted.
  [int x;   float y;   x+y – float]
- During assignment right hand side is converted to the type of variable on the left hand side. Data loss may occur. [y =x*5; (ok) x=y*5; (not ok)]
- It is suggested that mixing data types in expressions be avoided.

Type Cast (forced conversion):

Temporary change of type may be arranged if required.
Example:

```
float f = 100.25;

printf(“Integer part:%d”, (int) f);

printf(“Fractional part:%f”, f - (int) f);
```

LECTURE 26

Problem 1: Write a function called **im_pow** that has two integer arguments and returns first number raised to the power second number.

Problem 2: Write a function that returns the factorial of a number where number is passed to the function as argument.

TYPE CONVERSION (automatic)

- Mixing in an expression
  General principle: Operand by operand conversion is done to the maximum sized operand, if not otherwise restricted.

  [int x; float y; x+y – float]

- During assignment right-hand side is converted to the type of variable on the left-hand side. Data loss may occur.

  [y =x*5; (ok) x=y*5; (not ok)]

- It is suggested that mixing data types in expressions be avoided.

TYPE CAST (forced conversion)

Temporary change of type may be arranged if required.
Example:

```
float f = 100.25;

printf("Integer part:%d", (int) f);

printf("Fractional part:%f", f - (int) f);
```

DATA-TYPE MODIFIERS

The five (5) basic data types of C (void, char, int, float, double) can be modified using C's type modifiers. The type modifiers are:

(1) long (2) short (3) signed (4) unsigned

The type modifier precedes the type name. For example, this declares a long integer:

```
long int i;
```

SHORT & LONG:

-May be applied to **int**.
-As a general rule, **short int**s are smaller than **int**s and **long int**s are often larger than **int**s.
-However, the precise meaning of long and short is implementation dependent. In ANSI C standard, minimum ranges for integers, short integers and long integers are specified, not the fixed sizes for these items.

Example:
- In most 16-bit environments, an int is 16 bits, long int is 32 bits, no difference between int and short int.
- In many 32-bit environments, integers and long integers are of same size.
- According to ANSI C standard, the smallest acceptable size for an int is 16 bits. For a short int, the acceptable size is also 16 bits.

*The long modifier may also be applied to double. Doing so roughly doubles the precision of a floating point variable.

SIGNED & UNSIGNED:

- **signed** modifier is used to specify a signed integer value and a signed character.
- By default, integer declaration automatically creates a signed value.
- Whether char is signed or unsigned by itself is implementation dependent.
- **unsigned** modifier can be applied to **char** and **int**.
- It may also be used in combination with **short** and **long**.

The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C compiler will generate code that assumes the high-order bit is used as a sign flag. *If the sign flag is 0, the number is positive; if it is 1, the number is negative.*

Example:

	signed char	**unsigned char**
Typical size in Bits	8	8
Minimal Range	-128 to 127	0 to 255

Recursion
Recursive Function

LECTURE 27

RECURSION

Recursion is the process by which something is defined in terms of itself. When applied to computer languages, recursion means that a function can call itself. Not all computer languages support recursive functions, but C does.

Example 1:

```
#include<stdio.h>
void recurse(int i);
int main(void)
{
      recurse(0);
      return 0;
}
void recurse(int i)
{
      if(i<10)
      {
          recurse(i+1);
          printf(“%d”,i)
      }
}
```

This program prints: 9 8 7 6 5 4 3 2 1 0

Example 2: Finding factorials of positive integers

Factorial(n) = n x Factorial(n-1)

5! = 5x4! = 5x4x3! = 5x4x3x2! = 5x4x3x2x1! = 5x4x3x2x1

- Definition of a function for finding factorials

```
int fctrl(int n)
{
   if (n<2)
      return 1;                    /* Terminating rule or Base case */
   else
      return n*fctrl(n-1);       /* Recursive rule or Recursive case or Main rule */
}
```

- Demonstration of execution

fctrl(5) → 5 x fctrl(4) → 5 x 4 x fctrl(3) → 5 x 4 x 3 x fctrl(2)
→ 5 x 4 x 3 x 2 x fctrl(1)

→ 5 x 4 x 3 x 2 x 1 [**Winding / Coiling**]

5 x 4 x 3 x 2 x 1 → 5 x 4 x 3 x 2 → 5 x 4 x 6 → 5 x 24 → 120 [**Rewinding / releasing the coil**]

LECTURE 28

Problem 1: Finding the sum of a finite numeric series using recursion

2+4+6+8+10+ … + 2n

Sum of n terms = Sum of n-1 terms + nth term

2+4+6+8+10 = (2+4+6+8) +10

- Definition of a function for finding the sum of 1st n terms

```
int sum1(int n)
{
   if (n==0)
      return 0;                /* Terminating rule or Base case */
   else
      return sum1(n-1) + 2*n;  /* Recursive rule or Recursive case or Main rule */
}
```

- Demonstration of execution

Winding:

sum1(5) → sum1(4) + 10 → sum1(3) + 8 + 10 → sum1(2) + 6 + 8 + 10
→ sum1(1) + 4 + 6 + 8 + 10 → sum1(0) + 2 + 4 + 6 + 8 + 10
→ 0 + 2 + 4 + 6 + 8 + 10

Rewinding:

0 + 2 + 4 + 6 + 8 + 10 → 2 + 4 + 6 + 8 + 10 → 6 + 6 + 8 + 10 → 12 + 8 + 10

→ 20 + 10 → 30

Problem 2: Write a program to find the sum of the first ‘n’ terms of the Fibonacci series using loop and recursion separately in two different user-defined functions.

Solution:

```
#include<stdio.h>
void fibo(int pv1,int pv2);
int sum=0, n;
int main(void)
{
    printf("Enter number of terms: ");
    scanf("%d",&n);
    fibo(0,1);
    printf("sum: %d\n", sum);
    return 0;
}

void fibo(int pv1,int pv2)
{
    static int i=1;
    int next;
    if(i<n)
    {
        sum=sum+pv2;
        next=pv1+pv2;
        pv1=pv2;
        pv2=next; i++;
        fibo(pv1,pv2);
    }
}
```

**Now write another function which will be called from main function to do the same task using loop only

```
returnValue_type functionName(parameterList)
{
     //Statemenets
}
```

LECTURE 29

BASICS OF STRUCTURE

A. An example of declaration of Structure

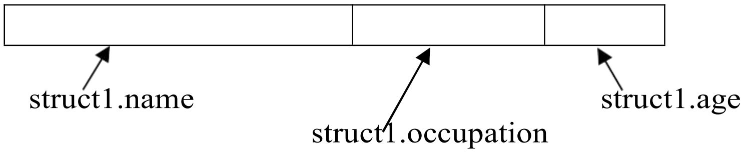
```
..................
struct ourStruct
{
    char name[40], occupation[15];
    int age;
} struct1, sa1[5];

int main(void)
{
    int i, j, e;
    struct ourstruct struct2, sa2[5];
     .....................
```

- A Structure is declared as a compound data type, generally, at the beginning of a program, outside all functions, like global variable declaration, and also used as a global declaration.
- Global variables and arrays of the declared compound type may be declared at the same time (`struct1, sa1[5]`).
- Local variables and arrays of the declared compound type may be used in a function (`struct2, sa2[5]`).

B. Memory allocation and accessing

- struct1:



[3 ‘fields’]

- Assigning values to the fields and accessing the data:

```
i = 36; srtuct1.age = i;
[or just, struct1.age = 36;]
gets(csn); strcpy(struct1.name, csn);

[or just, gets(struct1.age);]


printf(“Age: %d”, struct1.age);
printf(“Name: %s“, struct1.name);
```

Problem: Write a program to read name, age and marks of 3 courses for a student using structure and then print total marks along with the student's name.

LECTURE 30

PROBLEM 1: Write a program to read name, age and marks of 3 courses for a student using structure and then print total marks along with the student's name.

```
#include<stdio.h>

struct student
{
   char name[30];
   int age;
   float marks[3];
}std;

int main(void)
{
   int i;
   float sum=0;

   printf("Enter name: ");
   gets(std.name);
   printf("Enter age: ");
   scanf("%d",&std.age);
   printf("Enter 3 marks: ");
   for(i=0; i<3; i++)
   {
      scanf("%f",&std.marks[i]);
      sum+=std.marks[i];
   }
   printf("\n\n::: Output :::");
   printf("\n\nName: ");
   puts(std.name);
   printf("Total Marks: %f\n", sum);

   return 0;
}
```

PROBLEM 2: Write a program to read name, age and cgpa of a class of 10 students using structure and print average name length, average age and average cgpa.

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

struct student
{
    char name[30];
    int age;
    float cgpa;
}std[10];

int main(void)
{
    int i, nameLen=0, totalAge=0;
    float sumCG=0;

    for(i=0; i<10; i++)
    {
        printf("\n:: Student %d ::\n\n", i+1);
        printf("Enter name: ");
        gets(std[i].name);
        nameLen+=strlen(std[i].name);

        printf("Enter age: ");
        scanf("%d",&std[i].age);
        totalAge+=std[i].age;

        printf("Enter cgpa: ");
        scanf("%f",&std[i].cgpa);
        sumCG+=std[i].cgpa;

        getchar();
    }
    printf("\n\n::: Output :::");
    printf("\n\nAvg. Name Length: %f", nameLen/10.0);
    printf("\nAvg. age: %f", totalAge/10.0);
    printf("\nAvg. cgpa: %f\n", sumCG/10.0);

    return 0;
}
```

LECTURE 31, 32

BASICS OF POINTER:

Definition: A pointer is a variable that holds the *memory address* of another variable or object.

Declaration:

dataType *variableName;

- The variable name is preceded by an asterisk (*).
- A variable named ‘variableName’ is created to hold the address of a variable of type ‘dataType’, called *base type.*

Pointer Operators:

&: &p returns the address of p;
* : *p returns the value stored at the memory, address of which is stored in p.

Example:

```
int *p, *q;
p = &q;
```

Sample Data:

Variable	Memory	Content
p	100	102
q	102	?

p points to q. [&p → 100, &q → 102]

Now, ‘q = 1000;’ or ‘*p = 1000;’ will change the scenario as follows:

p	100	102
q	102	1000

Indirection:

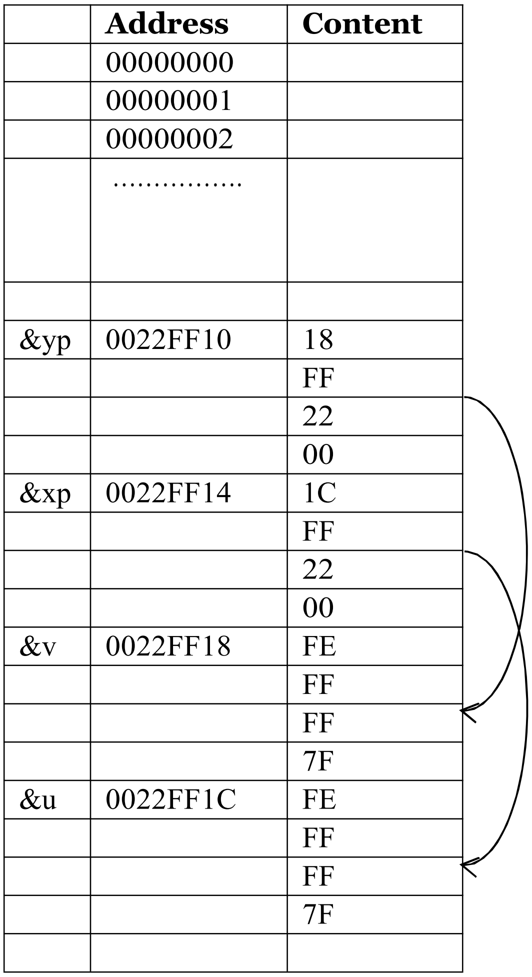
Value of a variable referenced through a pointer is known as indirection.

No type mixing allowed:

An integer pointer must not be assigned the address of a float variable.

Sample Memory Map:

```
long u = 2147483646, v;      /* (2147483646)10 = (7FFFFFFE)16 */
long *xp, *yp;
xp = &u;          v = *xp;            yp = &v;
```



	Address	Content
	00000000	
	00000001	
	00000002	
	…………….	
&yp	0022FF10	18
		FF
		22
		00
&xp	0022FF14	1C
		FF
		22
		00
&v	0022FF18	FE
		FF
		FF
		7F
&u	0022FF1C	FE
		FF
		FF
		7F

OPERATIONS WITH POINTERS:

- The only operations allowed:
  Binary: + -
  Unary: ++ --
- As addresses are <u>positive whole numbers</u>, <u>only integer</u> quantities may be added to or subtracted from a pointer, but <u>not</u> resulting to a <u>negative</u> number.
- Base type is important

```
int *p;
p = 200;
```

p++ will return 202, assuming 2 bytes for an integer.

```
float *q;
q = 220;
```

For ‘q= q-4;’ q will now contain 204, assuming 4 bytes for a floating-point number.

Use of Pointers for passing Arrays to Functions:

- To access elements of an array, the address of one of those elements is used.

  ```
  void abc(int *i, int j);
  ```

  /* Function ‘abc’ with a pointer parameter i is declared. This formal parameter is a reference to a memory location. */

  ```
  abc(&ar1[4], 3);
  ```

  /* Function ‘abc’ is called with the address of the 5th element of the array ar1. The 1st actual parameter is an address, while the 2nd actual parameter is a value. */

- Calling a function with values as actual parameters is known as **call by value**, while with addresses as actual parameters is known as **call by reference**.

- **Sample code**

```
void gp_sum(int i, float *j);

int main(void)
{
    float gp_list[] = {12.5, 10, 14.25, 8.75, 15, 7.5, 12, 10.25, 11};
    gp_sum(5, &gp_list[8]);

    /*    ..................*/
    return 0;
}

void gp_sum(int j, float *k)
{
    int m; float s=0;
    for(m=0; m<j; m++, k--)
    {
        s=s+*k;
    }
    printf("\nThe sum is %f.", s); /* sum of last 5 elements */
}
```