

CSE-100

Level-1 Term-1

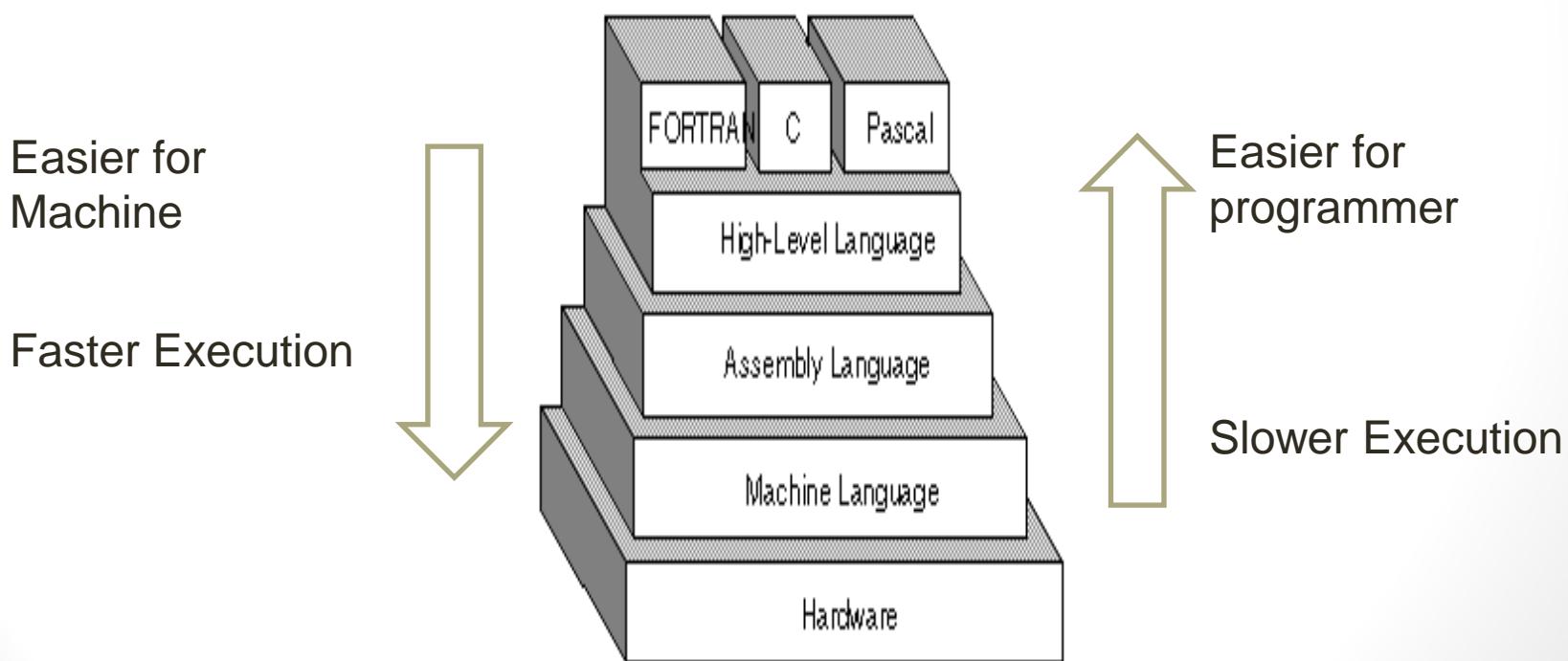
C Programming

Computer Programming

- Creating a sequence of instructions to enable the computer to do something.
- It is the process of designing, writing, testing, debugging and maintaining the source code of computer programs.
- **Ada Lovelace** is the first computer programmer.

Programming Language

- A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.



High Level Language

- A **high-level programming language** is a programming language with strong abstraction from the details of the computer.
- In comparison to low-level programming languages it uses natural language elements, easier to use and more portable across platforms.
- The main advantage of high-level languages over low-level languages is that they are easier to read, write, and maintain.
- programs written in a high-level language must be translated into machine language by a **compiler** or **interpreter**.
- Example: C, C++, Java, FORTRAN, Pascal, ADA etc.

Compiler



High Level Language	Compiler	Machine Language
#include<stdio.h> void main(void) { printf("Hello World"); }		10111011 10101011 11101110 11010110 11011101

History of C Programming Language

- Basic Combined Programming Language(**BCPL**) called **B** developed in 1960.
- **Dennis Ritchie** modified **B** in 1972 and the new language was named **C**.
- In 1983, the American National Standards Institute (**ANSI**) formed a committee to establish a standard specification of C. In 1989, the standard was ratified.
- This version of the language is referred to as ANSI C, Standard C, or sometimes C89.

Features of C Programming

. Language

- C is a structured programming language.
- C is a general purpose language.
- C is a mid level language.
- C is a case sensitive language.
- C is easily extendable.
- C is machine independent language.

ASCII Value

- ASCII – American Standard Code for Information Interchange.
- Every character in C programming is given an integer value to represent it. That integer value is known as ASCII value of that character.

ASCII Value

Decimal	Binary	Symbol
0	000 0000	Null character
.....
32	010 0000	Space
.....
48	011 0000	0
49	011 0001	1
.....
65	100 0001	A
.....
97	110 0001	a

Structure of a C Program

header file	//optional
global declaration	//optional
function 1	//optional
function 2	//optional
.....	
function n	//optional
main function	//Must – Entry point for a program
{	
local declaration	
statements	
}	

A Sample of a C Program

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

header file

main function

It will print “Hello World” in the screen

```
graph TD; HF[header file] --> H; MF[main function] --> M; IWS[It will print “Hello World” in the screen] --> P[printf("Hello World")];
```

Explanation of previous program

- Every program starts from **main() function**.
- **printf()** is a **library function** to display output which only works if `#include<stdio.h>` is included at the beginning.
- Here, stdio.h is a header file (**standard input output header file**) and `#include` is command to paste the code from the header file when necessary. When compiler encounters **printf()** function and doesn't find stdio.h header file, compiler shows error.
- Code `return 0;` indicates the end of program. You can ignore this statement but, it is good programming practice to use `return 0;`.

Main Function

- The main() is a special function used by the C system to tell the computer where the program starts.
- Every program must have exactly one main function.
- If you use more than one main function, the compiler cannot understand which one marks the beginning of the program.

Main Function (cont..)

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

Necessary Software

Codeblocks :

<http://www.codeblocks.org> ->

Downloads -> Binaries ->

“codeblocks-16.01mingw-
setup.exe”

Printf()

```
#include<stdio.h>
void main()
{
    printf("Hello World.");
    printf("My id is 07.");
    printf("I am a student of MIST.");
    printf("My Email id is - mist@mist.ac.bd");
}
```

Output

Hello World.My id is 07.I am a student of MIST.My Email id is - mist@mist.ac.bd

[16]

Printf()

```
#include<stdio.h>
void main()
{
    printf("Hello World.\n");
    printf("My id is 07.\nI am a student of MIST.");
    printf("\nMy Email id is - mist@mist.ac.bd");
}
```

Output

```
Hello World.
My id is 07.
I am a student of MIST.
My Email id is - mist@mist.ac.bd
```

Identifier

User defined name

- Must start with letter or ‘_’.
- In an identifier, middle characters are letter, digits, or underscore(‘_’).
- Must not match with keywords.
- Maximum 31 characters.
- In an identifier, upper & lowercase are treated as distinct. Ex. count, Count & COUNT are three separate identifiers.

Examples of Identifier

Correct

Count

Test23

High_balance

Incorrect

1count

Hi!there

high...balance

C Keywords

C has 32 keywords

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

Data Type

Type	Characteristics	Typical size in Bytes	Minimal Range	Examples
int	Numeric / integer	4 Byte	-2147483648 to 2147483647	10, -101, 15
float	Fractional numbers	4 Byte	3.4e-38 to 3.4e+38	10.568960
double	Fractional numbers	8 Byte	1.7e-308 to 1.7e+308	11.5789658912
char	Individual letters , symbols, digits.	1 Byte	-128 to 127	A, Y
void	No data type			

Variable

- In computer programming, a **variable** is a facility for storing data. They can be modified by the program.
- All variables **must be declared** before they can be used.
- Declaration of Variables:
 - It tells the compiler what the variable name is.
 - It specifies what type of data the variable will hold.
 - Format :
`data_type var_name;`
`data_type var_name1, var_name2, var_name3, ;`

Example

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

```
void main()
{
```

```
    int buying_price, selling_price;
```

```
    int profit;
```

```
    buying_price = 70;
```

```
    selling_price = 90;
```

```
    profit = ((selling_price - buying_price)/buying_price) * 100;
```

```
}
```

$$\text{profit} = \frac{\text{sell} - \text{buy}}{\text{buy}} \times 100$$

Format Specifier

Data Type	Format Specifier	Use
int	%d	To input/output Integer value
float	%f	To input/output Fractional (float) value
double	%lf	To input/output Fractional (double) value
char	%c	To input/output character

Example

```
#include<stdio.h>
void main()
{
    int var;
    var = 7 + 8;
    printf("%d", var);
    printf("\nResult is %d.", var);
}
```

Output

15

Result is 15.

(25)

Example

```
#include<stdio.h>
void main()
{
    float var, var1, var2;
    var1 = 7.45;
    var2 = 8.36;
    var = var1 + var2;
    printf("Addition of 7.45 and 8.36 is %f", var);
```

```
}
```

Output

Addition of 7.45 and 8.36 is 15.810000

Example

```
#include<stdio.h>
void main()
{
    int num;
    float amount;
    num=100;
    amount = 30.75 + 15.35;
    printf("%d\n",num);
    printf("%5.2f",amount);
}
```

Five places in all and two places to the right of the decimal point.

Output

```
100
106.10
```

Example

```
#include<stdio.h>
void main()
{ int var1, var2, var3;
  var1 = 7;
  var2 = 9;
  var3 = 13;
  printf("%d %d %d", var1, var2, var3); }
```

The diagram illustrates the mapping of printf format specifiers to variable values. It shows three "%d" specifiers in the printf call, each with an arrow pointing to its corresponding variable value below it. The first "%d" points to "var1" (7), the second to "var2" (9), and the third to "var3" (13). A large blue bracket groups all three variables under their respective "%d" specifiers. An arrow points from this group to a box containing the expanded printf statements.

```
printf("%d", var1);
printf(" %d", var2);
printf(" %d", var3);
```

Output

7 9 13

scanf() (cont...)

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

& = Ampersand sign



Scanf() (cont...)

```
#include<stdio.h>
void main()
{
    int var1, var2;
    scanf(" %d %d", &var1, &var2);
    printf("%d %d", var1, var2);
}
```

scanf(" %d", &var1);
scanf(" %d", &var2);

Input

2010
2014

Output

2010 2014

(30)

scanf() (cont...)

```
#include<stdio.h>
void main()
{
    int var, var1, var2;
    scanf(" %d %d", &var1, &var2);
    var = var1 + var2;
    printf("%d", var);
}
```



scanf() (cont...)

```
#include<stdio.h>
void main()
{
    int var1;
    float var2, result;
    scanf(" %d %f", &var1, &var2);
    result = var1 + var2;
    printf("Result is %f", var);
}
```

Input

20 10.5

Output

Result is 30.500000

[32]

scanf() (cont...)

```
#include<stdio.h>
void main()
{
    double buying_price, selling_price, profit;
    scanf(" %lf %lf", &buying_price, &selling_price);
    profit = ((selling_price - buying_price)/buying_price) *
    100;
    printf("profit = %lf", profit);
}
```

$$\text{profit} = \frac{\text{sell} - \text{buy}}{\text{buy}} \times 100$$

70
90

Input

Output
profit = 28.571429

(33)

scanf() (cont...)

```
#include<stdio.h>
void main()
{
    double buying_price, selling_price, profit;
    scanf(" %lf %lf", &buying_price, &selling_price);
    profit = ((selling_price - buying_price)/buying_price) *
    100;
    printf("profit = %.2lf", profit);
}
```

$$\text{profit} = \frac{\text{sell} - \text{buy}}{\text{buy}} \times 100$$

70
90

Input

profit = 28.57

Output

[34]

scanf() (cont...)

```
#include<stdio.h>
void main()
{
    char c1;
    scanf(" %c", &c1);
    printf("Output: %c", c1);
}
```



scanf() (cont...)

```
#include<stdio.h>
void main()
{
    char c1;
    scanf(" %c", &c1);
    printf("%d", c1);
}
```



scanf() (cont...)

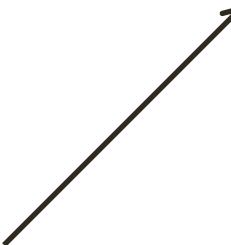
```
#include<stdio.h>
void main()
{
    char c1, c2;
    scanf(" %c", &c1);
    c2 = c1 + 1;
    printf("%c", c2);
}
```



Variation in Output for Integer and floats

```
#include<stdio.h>
int main()
{
    printf("Case 1:%6d\n",9876);
    printf("Case 2:%e\n",987.6543);
```

```
}
```

- 
1. Number right justified within 6 columns
 2. Number in Scientific notation

Case 1: 9876
Case 2: 9.876543e+002

output

What is return printf() & scanf()

```
#include<stdio.h>
int main()
{
    int var,var1,var2,var3;
    int p,q;
    p=scanf("%d %d",&var1,&var2);
    var=var1+var2;
    q=printf("%d\n",var);
    printf("%d",q);
    return 0;
}
```

Backslash Character Constants / Escape Sequences

Code	Meaning
\n	New Line
\b	Back Space
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\0	NULL
\"	Double quote
\'	Single quote
\\\	Backslash
\a	Alert
\?	Question mark
\f	Form feed

Add Comments to Program

```
#include<stdio.h>
void main()
{
    int x, y;          // x = selling price, y = buying price
    // this is a comment
    /*this is a comment*/
    /*
        this is a
        longer comment
    */
}
```

Arithmatic Operators

Operator	Name	Example
+	Addition	$7 + 8, x + y$
-	Subtraction	$17 - 11, x - 6$
*	Multiplication	$12 * 13, x * 3$
/	Division	$18 / 3, x / y$
%	Modulus	$17 \% 3, x \% 2$

Relational Operators

Operator	Name	Example
>	Greater than	$x > y$
<	Less than	$11 < 17, x < 6$
\geq	Greater than or equal	$x \geq y$
\leq	Less than or equal	$x \leq y$
\neq	Not equal to	$x \neq y$
\equiv	Equal to	$x \equiv y$

Sample Program

```
#include<stdio.h>
void main()
{
    int var, var1=9, var2=7;
    var = var1 > var2;           //var ← 1
    var = var1 == var2;          //var ← 0
    var = var1 != var2;          //var ← 1
    var = var1 % 2 == 0          //var ← 0
}
```

Logical Operators

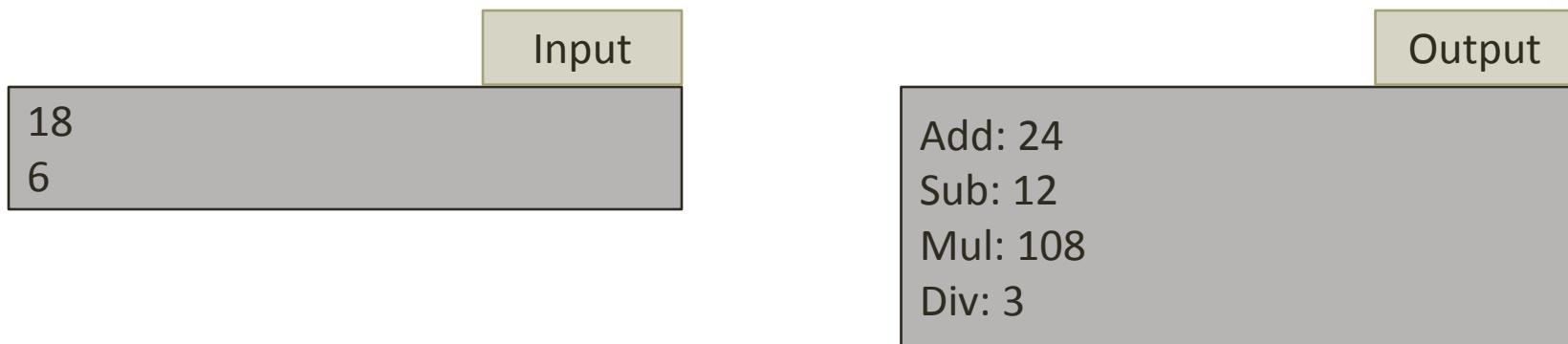
Operator	Name	Example
!	NOT	$!(x > y)$
&&	AND	$x < 6 \&\& x > 2$
	OR	$x \geq y \mid\mid a \leq b$

Sample Program

```
#include<stdio.h>
void main()
{
    int var, var1=9, var2=7;
    var = 9 > 7 && var1 != var2;           // var ← 1
    var = (var1 == 7) && (var1 > var2);   // var ← 0
    var = var1==7 || var2==7;              // var ← 1
    var = !(var1 == var2);                // var ← 1
}
```

Excercise

1. Write a program that takes two integer number as input and find out their addition, subtraction, multiplication and division result.



Excercise (Cont.)

2. Write a program that takes a temperature as input given in celcious scale. convert it to fahrenheit scale.

$$F=(9*C)/5+32$$



Excercise (Cont.)

3. Write a program that takes necessary inputs from user for solving the following equation and find out the value of X .

$$X = \frac{a}{b-c}$$

Excercise (Cont.)

4. Write a program that takes radius of a circle as input and find out the area of the circle.

$$\text{Area} = \text{Pi} \times r^2$$

Excercise (Cont.)

1. Print Hello World using printf () function.
2. Print an integer number using printf () function.
3. Print an integer number using printf () function which is taken from the user.
4. Take input of 3 integer number using scanf () function and calculate sum of them.
5. Take a character input and print its corresponding ASCII value.
6. Take an integer ($0 < N < 128$) input and print its corresponding Character.
7. Print a floating point number using printf () function.
8. Print slash (/) and backslash (\) using printf () function.
9. Take 10 input of any type of number and find out the average of them.
10. Take two integer numbers. Calculate the sum, subtraction, multiplication and division of them. Print the results in each new line.

Operator precedence in C determines the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated before operators with lower precedence. Here's a list of operator precedence in C, from highest to lowest:

1. Parentheses: ()
2. Array Subscript: []
3. Function Call: ()
4. Postfix Increment/Decrement: ++ --
5. Unary Increment/Decrement: ++ --
6. Unary Plus/Minus: + -
7. Logical NOT: !
8. Bitwise NOT: ~
9. Dereference (Pointer): *
10. Address-of (Pointer): &
11. Multiplication: *
12. Division: /
13. Modulus: %
14. Addition: +
15. Subtraction: -
16. Left Shift: <<
17. Right Shift: >>
18. Less Than: <
19. Less Than or Equal: <=
20. Greater Than: >
21. Greater Than or Equal: >=
22. Equality: ==
23. Inequality: !=
24. Bitwise AND: &
25. Bitwise XOR: ^
26. Bitwise OR: |
27. Logical AND: &&
28. Logical OR: ||
29. Conditional Operator (Ternary): ?:
30. Assignment: =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=
31. Comma: ,

It's important to note that operator precedence affects how expressions are evaluated. When in doubt, use parentheses to make the evaluation order explicit and avoid any ambiguity.

Some Examples :

1. Arithmetic Operators:

```
int a = 10, b = 5, c = 3;
```

```
int result1 = a + b * c; // 10 + (5 * 3) = 25
int result2 = (a + b) * c; // (10 + 5) * 3 = 45
```

2. Relational and Logical Operators:

```
int x = 8, y = 5;
```

```
int result3 = x > y && y != 0; // 1 (True)
int result4 = x <= y || y == 0; // 0 (False)
```

3. Bitwise Operators:

```
unsigned int num = 7;
```

```
unsigned int result5 = num << 2; // 28 (Binary: 111 << 2 = 11100)
unsigned int result6 = num | 8; // 15 (Binary: 111 | 1000 = 1111)
```

4. Ternary (Conditional) Operator:

```
int m = 5, n = 10;
```

```
int result7 = (m > n) ? m : n; // 10
```

5. Assignment Operators:

```
int p = 20;
```

```
p += 5; // p = p + 5 => 25
p *= 2; // p = p * 2 => 50
```

6. Unary Operators:

```
int q = 8;
```

```
int result8 = ++q; // Increment q and assign to result8 => 9
int result9 = q--; // Assign q to result9 and then decrement q => 9 (q is now 8)
```

7. Pointer Operators:

```
int value = 42;  
int *ptr = &value;  
  
int result10 = *ptr; // Dereference ptr to get value => 42
```

CSE – 105

Structured Programming Language

Operators

(Week – 02, Part - 01)

Operators

- Symbol that tells the computer to perform certain mathematical or logical manipulation
 - Arithmetic Operators
 - Relational Operators
 - Logical Operators
 - Bitwise Operators
 - Assignment Operators
 - Increment and Decrement Operators
 - Conditional Operators
 - Special Operators

Arithmatic Operators

- Integer Arithmatic – both operands are integers
 - $3 + 2 = 5$
 - $3 - 2 = 1$
 - $3 * 2 = 6$
 - $3 / 2 = 1 \rightarrow$ Decimal part trancated
 - $3 \% 2 = 1 \rightarrow$ Remainder
- Real Arithmatic
 - $3 / 2 = ?$
 - $3 \% 2 = ?$

Arithmatic Operators

```
#include<stdio.h>
int main()
{
    int var1, var2, sum, multiplication, division, mod;
    scanf("%d %d", &var1, &var2);
    sum = var1 + var2;
    multiplication = var1 * var2;
    division = var1 / var2;
    mod = var1 % var2;
    printf("Addition: %d\n", sum);
    printf("Multiplication: %d\n", multiplication);
    printf("Division: %d\n", division);
    printf("Modulus: %d\n", mod);
    return 0;
}
```

Relational Operators

- $a > b$: a is greater than b
- $a < b$: a is less than b
- $a \geq b$: a is greater than or equal to b
- $a \leq b$: a is less than or equal to b
- $a == b$: a is equal to b
- $a != b$: a is not equal to b

Relational Operators

```
#include<stdio.h>
int main()
{
    int result, var1=9, var2=7;
    result = var1 > var2;
    result = var1 == var2;
    result = var1 != var2;
    result = var1 % 2 == 0;
    return 0;
}
```

Logical Operators

- Condition 1 **&&** Condition 2 – both have to be true to make this statement true - **AND**
- Condition 1 **||** Condition 2 – atleast one of them has to be true to make this statement true – **OR**
- **!** – to make a statement inverse - **NOT**

Logical Operators

```
#include<stdio.h>
int main()
{
    int result, var1=9, var2=7;
    result = 9 > 7 && var1 != var2;
    result = (var1 == 7) && (var1 > var2);
    result = var1==7 || var2==7;
    result = !(var1 == var2) ;
    return 0;
}
```

Bitwise Operators

Operator	Name	Example
<code>~</code>	bitwise Complement	<code>~var</code>
<code>&</code>	bitwise AND	<code>var1 & var2</code>
<code> </code>	bitwise OR	<code>var1 var2</code>
<code>^</code>	bitwise XOR	<code>var1 ^ var2</code>
<code><<</code>	Left shift	<code>var1 << 1</code>
<code>>></code>	Right Shift	<code>var2 >> 2</code>

Bitwise Operators

```
#include<stdio.h>
int main()
{
    int a,b;
    a=12; b=9;
    printf("%d\n", a&b);
    printf("%d\n", a|b);
    printf("%d\n", a^b);
    printf("%d\n", a<<1);
    printf("%d\n", a>>1);
    return 0;
}
```

Assignment Operators

Operator	Name	Example	Meaning
=	Assignment	Var = 7;	Var \leftarrow 7
++	Increment and Assignment	var++;	var = var + 1;
--	Decrement and Assignment	Var--;	var = var - 1;
+=	Add and assign	var += 7;	var = var + 7;
-=	Subtract and assign	var -= var1;	var = var - var1;
*=	Multiply and assign	var *= 7;	var = var * 7;
/=	Divide and assign	var /= var1;	var = var / var1;
%=	Take modulus and assign	var %= var1;	var = var % var1;
=	bitwise OR and assign	var = var1;	var = var var1;
&=	bitwise AND and assign	var &= var1;	var = var & var1;
^=	bitwise XOR and assign	var ^= var1;	var = var ^ var1;
<<=	Left shift and assign	var <<= 2;	var = var << 2;
>>=	Right shift and assign	var >>= var1;	var = var >> var1;

Postfix, Prefix

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

Increment and Decrement Operators!!

++, --

Unary Operators

++, --

Ternary Operator ?:

condition ? true result : false result;

1st Part

2nd Part

3rd Part

```
var1 > var2 ? printf("var1 Big") : printf("var2 Big");
```

Ternary Operator ?:

```
#include<stdio.h>

int main()
{
    int a = 6, b = 10;
    a > b ? printf("A is greater than B") : printf("B is
greater than A");
    x = (a%2==0) ? 1: 0;
    return 0;
}
```

Type Casting

- Implicit – Usually the '**lower**' type automatically converted to the '**higher**' type
 - float will convert into double implicitly
- Explicit
 - float to int
 - double to float – this is explicit because double is higher and float is lower.
 - long int to int

Type Casting

Example	Action
<code>x=(int)7.5</code>	7.5 is converted to integer
<code>a=(int)21.3/(int)4.5</code>	Evaluated as $21/4$ and the result would be 5
<code>b=(double)sum/n</code>	Division is done in floating point mode
<code>Y=(int)(a+b)</code>	The result of $a+b$ is converted to integer
<code>Z=(int)a+b</code>	a is converted to integer and then added to b

Expressions

- Any thing that performs an operation is an expression

$$X = A + B;$$

Here, the priority of addition operator is greater than assignment operator, that's why the operations will perform from right to left

Operator Precedence

Rank of Equality operator is 7

Rank of Bitwise And operator is 8

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int num = 4;
6     num & 1 == 0 ? printf( "Even") : printf( "Odd");
7
8     return 0;
9 }
```

Self study

- `sizeof()`
- Precedence of operators
- Math functions
- `scanf()`, `printf()`
- `getchar()`, `putchar()`

Self study

- `scanf("%2d % 5d",&num1,&num2);`
- `scanf("%d %*d %d",&a,&b,&c);`
- `scanf("%d %*d %d",&a,&b);`
- `printf("%d",9876);`
- `printf("%6d",9876);`
- `printf("%2d",9876);`
- `printf("%-6d",9876);`
- `printf("%06d",9876);`

References

- Programming in ANSI C
 - E Balagurusamy
 - Chapter 3 (Operators and Expressions)
- Programming in ANSI C
 - E Balagurusamy
 - Chapter 4 (Managing Input and Output Operations)
 - Topic: Input character string is excluded for the time being

Course Site

Google Classroom : 27tqtvk

Discussion on Last sessional class

Interesting Bitwise Operations

- To check/keep unchanged a bit of a number n – $n \& 1$
- To set a bit = use bitwise OR – $\text{num} | 1$
- To change the UPPER case letters to LOWER case – use letter $| ' '$

```
1 #include<stdio.h>
2
3 int main()
4 {
5     char upperCaseLetter = 'C';
6
7     printf("%c ", upperCaseLetter | ' ');
8     /* Bitwise OR with space(' ') */
9
10 }
11
```

Interesting Bitwise Operations

- **Multiplied by 2**

`n << 1; // n*2`

- **Divided by 2**

`n >> 1; // n/2`

- **Multiplied by the m-th power of 2**

`n << m;`

- **Divided by the m-th power of 2**

`n >> m;`

Interesting Bitwise Operations

- **Exchange two values (Swap)**

$a \wedge= b;$

$b \wedge= a;$

$a \wedge= b;$

- **Check odd number**

$(n \& 1) == 1;$

To change uppercase to lowercase

```
1  /*Upper case to lower case conversion */
2  #include<stdio.h>
3
4  int main()
5  {
6      char upperCaseLetter, lowerCaseLetter;
7      upperCaseLetter = getchar();
8
9      lowerCaseLetter = upperCaseLetter | ' ';
10     /* Bitwise OR with space(' ') */
11
12     putchar(lowerCaseLetter);
13
14     return 0;
15 }
16
```

To change uppercase to lowercase

```
1  /*Upper case to lower case conversion */
2  #include<stdio.h>
3
4  int main()
5  {
6      char upperCaseLetter, lowerCaseLetter;
7      upperCaseLetter = getchar();
8
9      lowerCaseLetter = upperCaseLetter | ' ';
10     /* Bitwise OR with space(' ') */
11
12     putchar(lowerCaseLetter);
13
14     return 0;
15 }
16
```

To change Lowercase to uppercase

```
1  /*Lower case to upper case conversion */
2  #include<stdio.h>
3
4  int main()
5  {
6      char lowerCaseLetter = 'a';
7
8      printf("Uppercase %c\n",lowerCaseLetter ^ ' ');
9      /* Bitwise XOR with space(' ') */
10
11     return 0;
12 }
13
```

Extract Digits from a Number

- Steps –
 - Mod with base number to get the last digit
 - Divide with base number to discard the last value

Extract Digits from a Hexadecimal Number

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int HexNumber = 110;
6     int unitPlaceDigit;
7     int DecimalPlaceDigit;
8     /* Binary value of 110 is 01101110 */
9     /* Hexadecimal value of 110 is 6E */
10
11    unitPlaceDigit = HexNumber%16; /* Mod with base 16 */
12    printf("Unit Place Digit %X\n", unitPlaceDigit);
13
14    HexNumber = HexNumber/16;
15
16    DecimalPlaceDigit = HexNumber%16;
17    printf("Decimal Place Digit %X\n", DecimalPlaceDigit);
18
19    return 0;
20
21 }
```

“ Any fool can write code that a computer can understand. Good programmers write code that humans can understand. ”

- Martin Fowler

CSE – 105

Structured Programming Language

Control Statement

(Week – 02, Part - 02)

Conditional Statement

- **Control structures** control the flow of execution in a program.
 - if statement
 - switch statement
 - Conditional operator statement
 - goto statement
 - Loop

if..else statement

- When we have only two options
- Any non-zero value will be considered as true.

```
if (condition)
{
    /*Control will come inside only when the above condition is true*/
    //C statement(s)
}
else
{
    /*Control will come inside only when condition is false */
    //C statement(s)
}
```

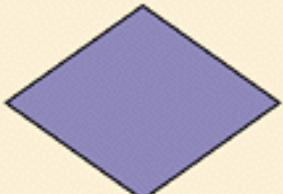
Example - 01

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int age;
6     scanf ("%d", &age);
7
8     if (age < 18)
9     {
10         printf ("Not eligible for vote");
11     }
12     else
13     {
14         printf ("Eligible for vote");
15     }
16
17     return 0;
18 }
19
```

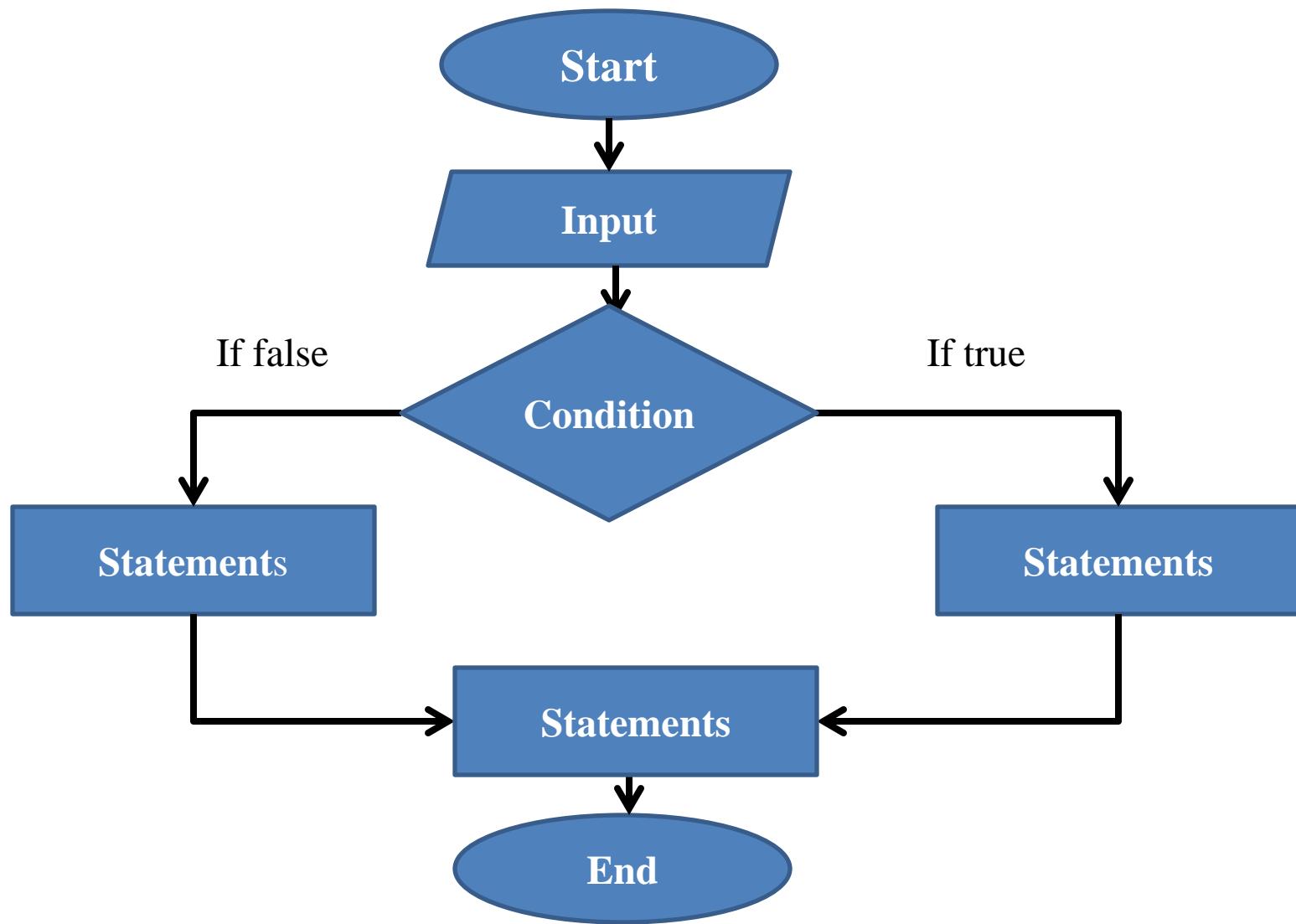
Flowchart

- Step by step procedure to complete a work is called **Algorithm**
- To represent the algorithm with symbol is called **Flowchart**

Flowchart Symbols

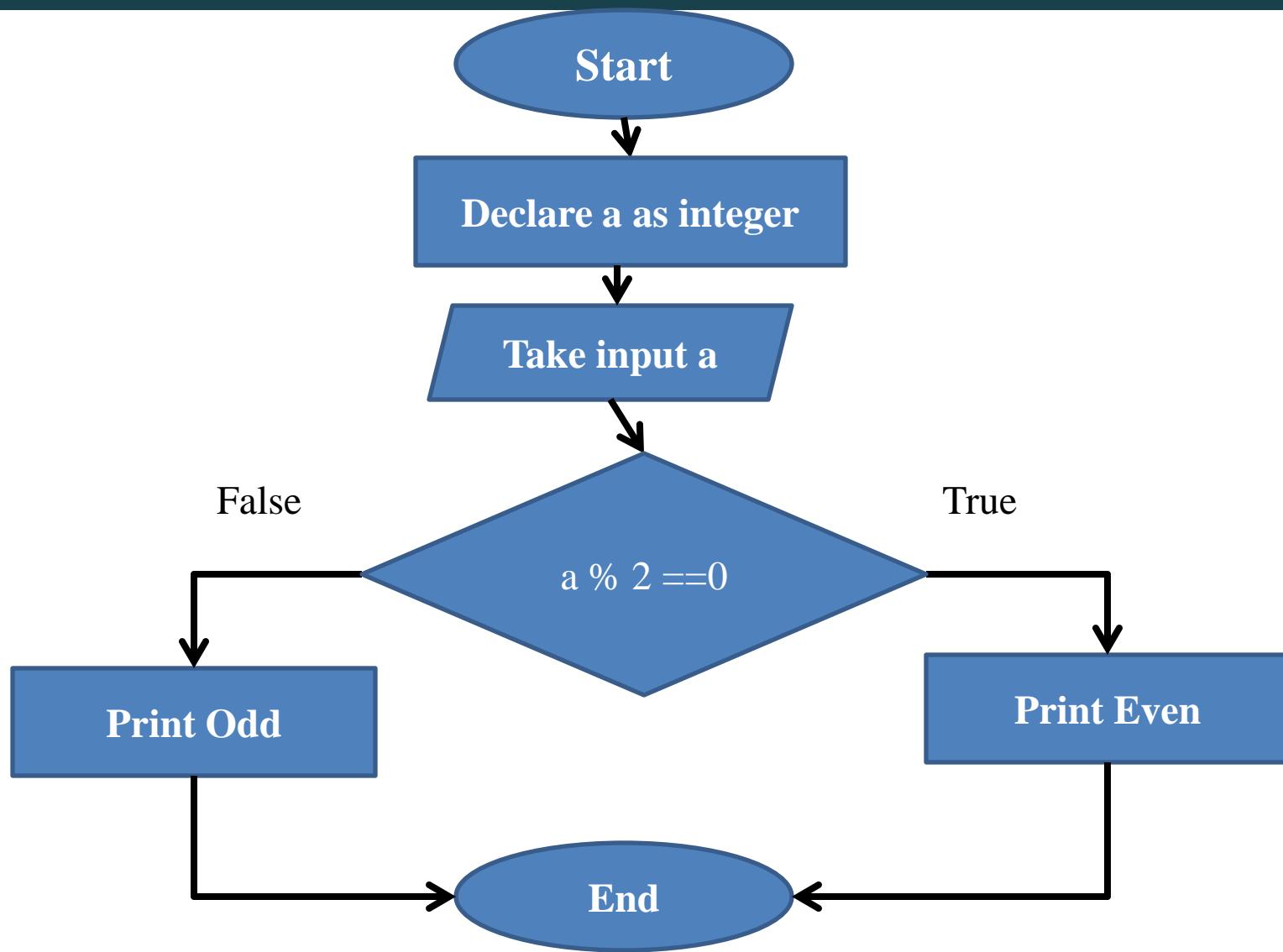
Name	Symbol	Use in flowchart
Oval		Denotes the beginning or end of a program.
Flow line		Denotes the direction of logic flow in a program.
Parallelogram		Denotes either an input operation (e.g., INPUT) or an output operation (e.g., PRINT).
Rectangle		Denotes a process to be carried out (e.g., an addition).
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes (e.g., IF/THEN/ELSE).

Flowchart Structure of Simple if..else



- Draw a flowchart to test a number is whether even or odd

Flowchart



Code Representation

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int a;
6     scanf ("%d", &a);
7
8
9     if(a % 2 == 0)
10    {
11        printf ("Even Number");
12    }
13    else
14    {
15        printf ("Odd Number");
16    }
17
18    return 0;
19 }
```

- $1 \% 3 = ?$
- $2 \% 3 = ?$
- $3 \% 3 = ?$
- $4 \% 3 = ?$
- $5 \% 3 = ?$
- $6 \% 3 = ?$
- $7 \% 3 = ?$
-

- $1 \% 3 = ?$
- $2 \% 3 = ?$
- $3 \% 3 = ?$
- $4 \% 3 = ?$
- $5 \% 3 = ?$
- $6 \% 3 = ?$
- $7 \% 3 = ?$

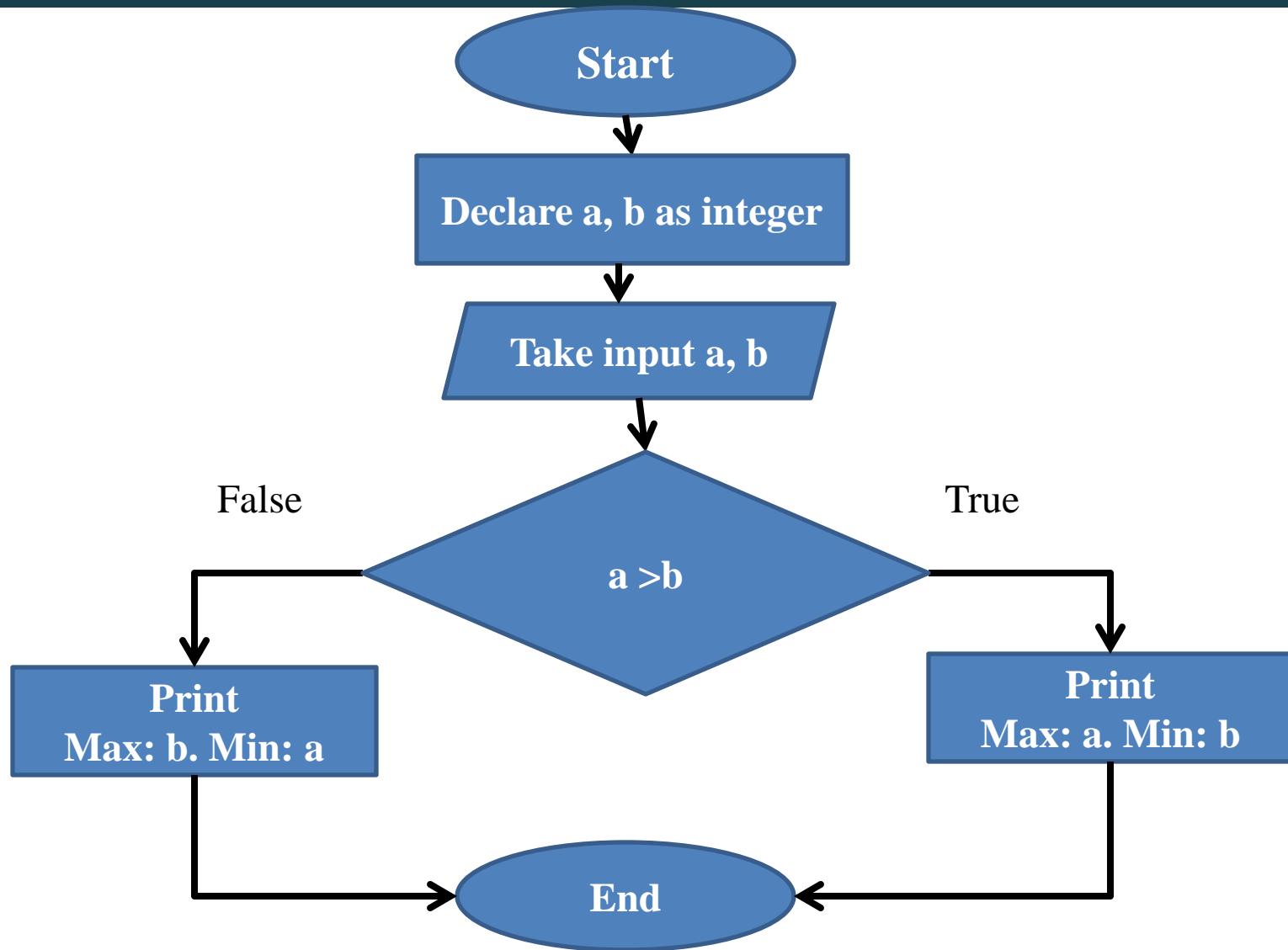
.....

If we divide any number m with n ($m \% n$), remainder will be from 0 to (n-1)

Max and Min Between Two Values

- Draw flowchart
- Write the program

Flowchart



Code Representation

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int a;
6     scanf ("%d %d", &a, &b);
7
8
9     if(a > b)
10    {
11        printf("Max: %d, Min: %d", a,b);
12    }
13    else
14    {
15        printf("Max: %d, Min: %d", b,a);
16    }
17
18    return 0;
19 }
20 }
```

- Find out the Maximum and Minimum among three numbers

Assignment

Draw flowchart and then write the program

1. Find out the maximum and minimum among four numbers

Submission deadline:

References

- Programming in ANSI C
 - E Balagurusamy
 - Chapter 5 (Decision Making and Branching)

Thank You !

Introduction to Control Statements

Increment & Decrement (section 2.5)

- **Postfix**

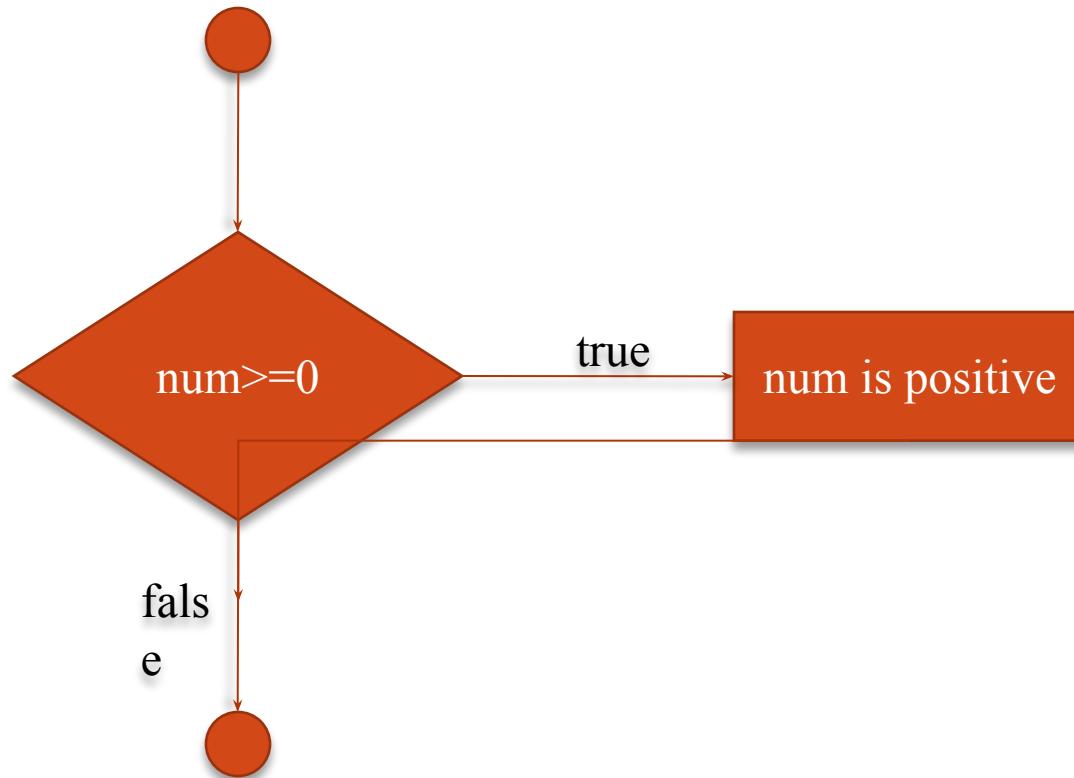
- **j=i++;**

- First current value of **i** is assigned to **j**
- Then **i** is incremented
- If the current value of **i** is 5
 - After the execution of the statement the value of
 - **i:** 6
 - **j:** 5

Increment & Decrement (section 2.5)

- **Prefix**
- **j=++i;**
 - First **i** is incremented
 - Then current value of **i** is assigned to **j**
 - If the current value of **i** is 5
 - After the execution of the statement the value of
 - **i:** 6
 - **j:** 6

if statement



if statement

- Selection statement/conditional statement
- Operation governed by outcome of a conditional test
- **if(*expression*) *statement*;**
 - *expression*:
 - any valid C expression
 - If *expression* is **true** *statement* will be executed
 - If *expression* is **false** *statement* will be bypassed
 - **true**: any nonzero value
 - **false**: zero
 - if(num+1) printf("nonzero");//num!=-1 statement will execute
 - Normally *expression* consists of relational & logical operator

true, false

- **true**: any **nonzero** value
- **false**: **zero**

if statement

```
#include<stdio.h>
int main(void)
{
    int num;
    scanf("%d", &num);
    if(num>=0) printf("num is positive");//if(num>-1)
    return 0;
}
```

if statement

```
#include<stdio.h>
int main(void)
{
    int num;
    scanf("%d", &num);
    if(num>=0) printf("num is positive");//if(num>-1)
    if(num<0) printf("num is negative");
    return 0;
}
```

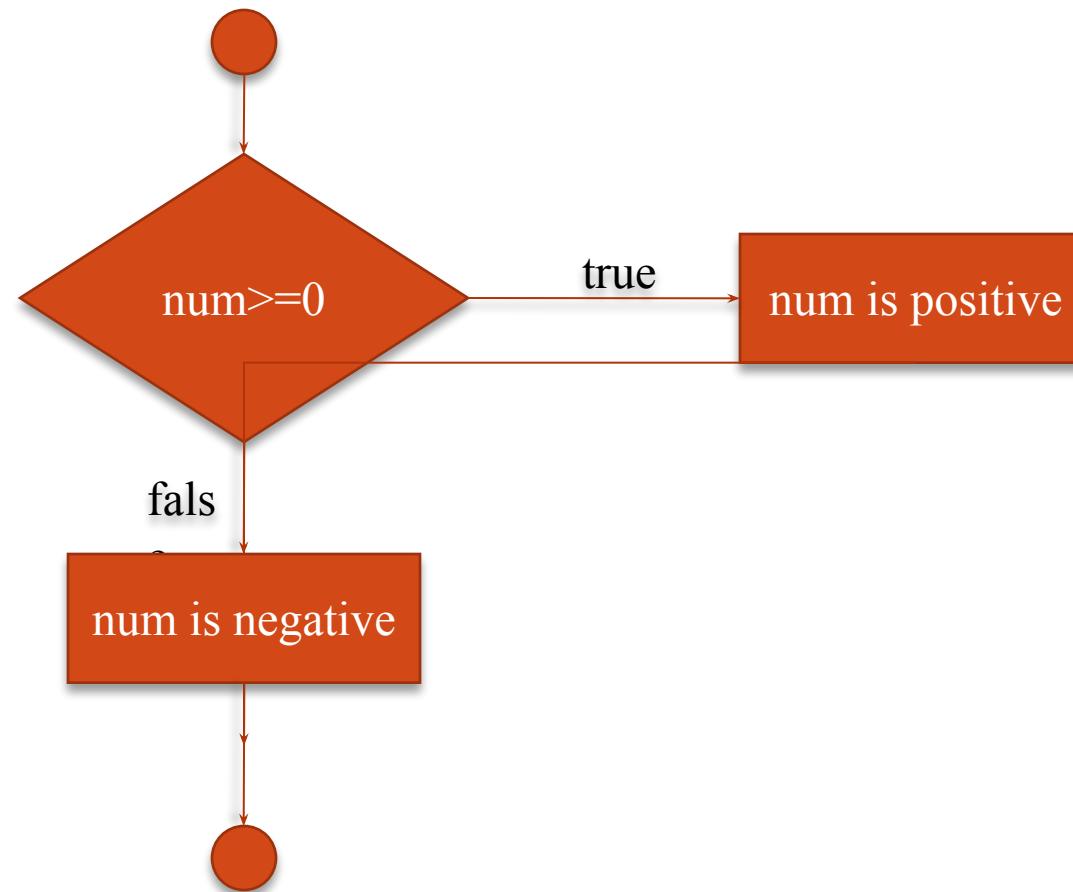
if statement

- Common programming error:
 - Placing ; (semicolon) immediately after condition in if
 - `if(expression); statement;`
 - Confusing equality operator (==) with assignment operator (=)
 - `if(a=b)`
 - `if(a=5)`
 - `if(9=5)`
 - left operand must be l-value
 - `if(9+5)`

if-else statement

- *if(expression) statement1;
else statement2;*
 - If *expression* is **true** *statement1* will be evaluated and *statement2* will be skipped
 - If *expression* is **false** *statement1* will be bypassed and *statement2* will be executed
 - Under no circumstances both the statements will execute
 - Two-way decision path

if-else statement



if-else statement

```
#include<stdio.h>
int main(void)
{
    int num;
    scanf("%d", &num);
    if(num>=0) printf("num is positive");//if(num>-1)
    else printf("num is negative");
    return 0;
}
```

if-else statement

- **else** part is optional
- The **else** is associated with closest **else-less if**
- ```
if (n>0)
 if (a>b) z=a;
 else z=b;
```
- Where **else** will be associated is shown by indentation
- Braces must be used to force association with the first
- ```
if (n>0)
{
    if (a>b)  z=a;
}
else  z=b;
```

Nested if (section 3.4)

```
#include<stdio.h>
int main(void)
{
    int id;
    printf("Please enter last
3 digits of your id:\n");
    scanf("%d", &id);
    printf("You are in ");
    if(id%2)
    {
        if(id<60)
            printf("A1\n");
        else
            printf("A2\n");
    }
    else
    {
        if(id<61)
            printf("B1\n");
        else
            printf("B2\n");
    }
    return 0;
}
```

blocks of code

- Surround the statements in a block with opening and ending curly braces.
- One indivisible logical unit
- Can be used anywhere a single statement may
- Multiple statements
- Common programming error:
 - Forgetting braces of compound statements/blocks

blocks of code

- ```
if(expression) {
 statement1;
 statement2;
 ...
 statementN;
}
else {
 statement1;
 statement2;
 ...
 statementN;
}
```

  - If *expression* is **true** all the statements with if will be executed
  - If *expression* is **false** all the statements with else will be executed

# Example

```
#include<stdio.h>
int main()
{
 int numOfArg, sum;
 scanf ("%d", &numOfArg);
 if (numOfArg==2)
 {
 int a, b;
 scanf ("%d %d", &a, &b);
 sum=a+b;
 }
}
```

```
else
{
 int a, b, c;
 scanf ("%d %d %d", &a, &b, &c);
 sum=a+b+c;
}
printf ("The sum is %d\n",
sum);
return 0;
}
```

# if-else if statement

- *if(expression)  
statement;  
else if (expression)  
statement;  
else if (expression)  
statement;  
else  
statement;*

# if-else if statement

- Multi-way decision
- *expressions* are evaluated in order
- If any *expression is true*
  - the *statement* associated with it is executed
    - Multiple *statements* can be associated using curly braces
    - the whole chain is terminated
- If none of the *expressions* are true
  - **else** part is executed
  - Handles none of the above/ default case
  - Optional

# if-else if statement

```
#include<stdio.h>
int main()
{
 int num;
 scanf ("%d", &num);
 if (num>=80)
 printf ("5.0\n");
 else if (num>=75)
 printf ("4.75\n");
 else if (num>=70)
 printf ("4.50\n");
 ...
 ...
 ...
 else
 printf ("0.0");
 return 0;
}
```

# if-else if statement

```
#include<stdio.h>
int main()
{
 int numOfArg, sum;
 scanf("%d", &numOfArg);
 if(numOfArg==2)
 {
 int a, b;
 scanf("%d %d", &a,
 &b);
 sum=a+b;
 }
 else if(numOfArg==3)
 {
 int a, b, c;
 scanf("%d %d %d", &a, &b,
 &c);
 sum=a+b+c;
 }
 printf("The sum is %d\n",
 sum);
 return 0;
}
```

# Short Circuit Evaluation

```
if (a !=0 && num/a)
{
}
}
```

# Conditional Expressions (sec 11.7)

- Uses ternary operator “?:”
- $expression1?expression2:expression3;$
- $z= (a>b)? a: b; /* z=max(a,b); */$
- Can be used anywhere an expression can be

# Example

- Find maximum of three numbers
- Find second maximum of three numbers
- Find minimum of four numbers

# **switch case**

```
switch (expression) {
 case constant: statements
 case constant: statements
 default: statements
}
```

- Use of break

# switch case

```
switch (month) {
 case 1: printf("January\n");
 case 2: printf("February\n");
 ...
 ...
 default: printf("Invalid\n");
}
```

# switch case

```
#include<stdio.h>
#include<conio.h>
int main()
{
 char i=getch(); //getche();
 switch (i) {
 case '0': case '1': case '2': case '3': case '4': case
 '5': case '6': case '7': case '8': case '9':
 printf(" : digit\n");
 break;
 default: printf(" : non digit\n");
 }
 return 0;
}
```

- Use of break

# switch case

```
//int x=a/b;
int x, a, b;
scanf("%d %d", &a, &b);
switch (b) {
 case 0:
 printf("divide by zero error\n");
 break;
 default: x=a/b;
}
```

# for loop

- Allows one or more statements to be repeated
- *for(initialization; conditional-test; increment) statement;*
- Most flexible loop

# for loop

- *for(initialization; conditional-test; increment) statement;*
- *initialization:*
  - Give an initial value to the variable that controls the loop
  - *loop-control variable*
  - Executed only once
  - Before the loop begins

# for loop

- *for(initialization; conditional-test; increment) statement;*
- *conditional-test:*
  - Tests the *loop-control variable* against a target value
  - If **true** the loop repeats
    - *statement* is executed
  - If **false** the loop stops
    - Next line of code following the loop will be executed

# for loop

- *for(initialization; conditional-test; increment) statement;*
- *increment:*
  - Executed at the bottom of the loop

# for loop

## Single Statement

```
for(i=1; i<100; i++)
 printf("%d\n", i);
```

- Prints 1 to 99

```
for(i=100; i<100; i++)
 printf("%d\n", i);
```

- This loop will not execute

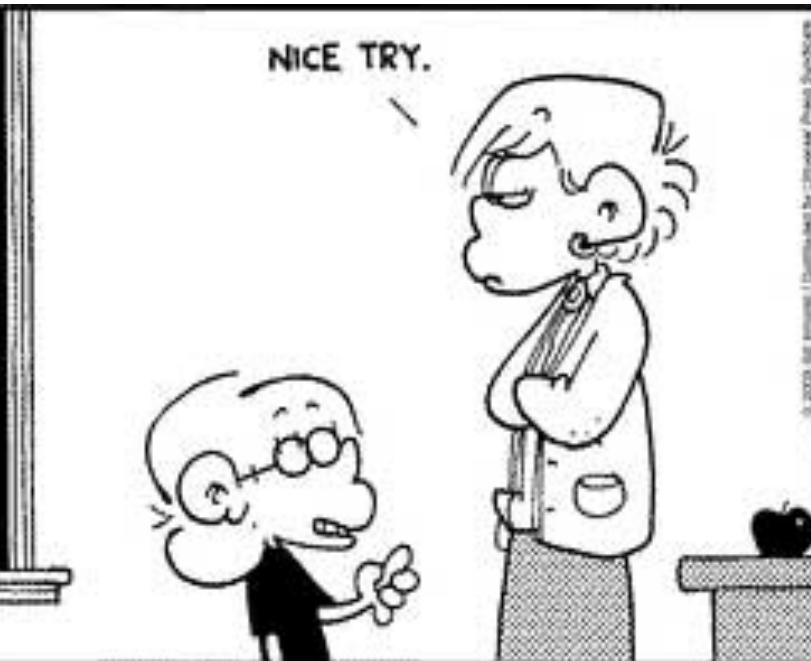
## Block of Statements

```
sum=0;
prod=1;
for(i=1; i<5; i++)
{
 sum+=i;
 prod*=i;
}
printf("sum, prod is
%d, %d\n", sum, prod);
```

# Programming Jokes!

```
#include <stdio.h>
int main(void)
{
 int count;
 for(count = 1; count <= 500; count++)
 printf("I will not throw paper airplanes in class.");
 return 0;
}
```

MEND 10-3



# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

1. i is initialized to 1

Initialization part is  
executed only once

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

2. Conditional test  $i < 3$  is true as  $i$  is 1, so the loop executes

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

3. The value of i will be printed, which is 1

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

3. The value of i will be incremented, so now i is 2.

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

4. Conditional test  $i < 3$  is true as  $i$  is 2, so the loop executes

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

5. The value of i will be printed, which is 2

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

6. The value of i will be incremented, so now i is 3.

# for loop

```
for(i=1; i<3; i++)
 printf("%d\n", i);
```

7. Conditional test  $i < 3$  is false as  $i$  is 3, so the loop stops

# for loop

- for loop can run negatively
- *decrement* can be used instead of *increment*
  - `for(i=20; i>0; i--) ...`
- Can be incremented or decremented by more than one
  - `for(i=1; i<100; i+=5)`

# for loop

- All of the following loops will print 1 to 99
- ```
for(i=1; i<100; i++)
    printf("%d\n", i);
```
- ```
for(i=1; i<=99; i++)
 printf("%d\n", i);
```
- ```
for(i=0; i<99; i++)
    printf("%d\n", i+1);
```
- ```
for(i=0; i<=98; i++)
 printf("%d\n", i+1);
```
- So selection of initial value and loop control condition is important

# for loop

## GCD of two numbers:

```
#include <stdio.h>
int main(void) {
 int a, b, min, i, gcd;
 scanf ("%d %d", &a, &b);
 min=(a<b)?a:b;
 for(i=1; i<=min; i++)
 if(a%i==0 && b%i==0)
 gcd=i;
 printf ("gcd of %d & %d is %d\n", a, b, gcd);
 return 0;
}
```

# for loop

## Nth Fibonacci number:

```
#include <stdio.h>
int main(void) {
 int n, i, fibn=0,
 fibn_1=1, fibn_2=0;
 scanf ("%d", &n);
 for(i=0; i<n; i++)
 {
 if(n==1)
 fibn=i;
 else
 {
 fibn=fibn_1+fibn_2;
 fibn_2=fibn_1;
 fibn_1=fibn;
 }
 }
 printf ("%d\n", fibn);
 return 0;
}
```

# Nested for loop

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

```
int main()
{
 int i, j;
 for(i=1; i<=3; i++)
 {
 for(j=1; j<=i; j++)
 {
 printf("%d, %d\n", i, j);
 }
 }
 return 0;
}
```

**Output:**

1, 1  
2, 1  
2, 2  
3, 1  
3, 2  
3, 3

What if the condition is  
 $j \leq 3$ ?

# Nested for loop

```
#include<stdio.h>

int main()
{
 int n, sum, prod;
 prod=1;
 sum=0;
 scanf("%d", &n);
 for(int i=1; i<=n; i++) //if i<n?
 {
 prod=1;
 for(int j=0; j<i; j++) //if
 j=1?
 {
 prod=prod*i;
 }
 sum=sum+prod;
 }
 printf("sum is %d\n", sum);
 return 0;
}
```

# Nested for loop

```
#include<stdio.h>

int main()
{
 int n, i, j;
 scanf("%d", &n);
 for(i=1; i<=n; i++) // row
 by row
 {
 for(j=1; j<=i; j++)
 {
 printf("0 ");
 }
 for(;j<=n; j++)
 {
 printf("1 ");
 }
 printf("\n");
 }
 return 0;
}
```

# Loop variation

- `for( ; ; ){}`
- `for(ch=getche(); ch!='q'; ch=getche()) {}`
- `for(i=0; i<n; )`  
    {  
        `i++;`  
    }

# Homework on loop

- Given a number as input write a program to calculate the number of digits.
- Given a number as input write a program to calculate the sum of its digits.
- Write a program to find gcd of two given numbers
- Write a program to find  $x^m$  where x and m are inputs
- Write a program to convert a decimal number to a binary number
- Write a program to expand shorthand notation like a-z

# while loop

- `while(expression) statement;`
- `for(initialization; conditional-test; increment) statement;`
- `initialization;`  
`while(conditional-test)`  
`{`  
`statement;`  
`increment;`  
`}`

# while loop

## while

```
#include<stdio.h>
int main()
{
 int i=0;
 while(i<=9)
 {
 printf("%d\n", i);
 i++;
 }
 return 0;
}
```

## for

```
#include<stdio.h>
int main()
{
 int i;
 for(i=0; i<=9; i++)
 printf("%d\n", i);
 return 0;
}
```

# while loop

- Common error
  - Forgetting to increment
- Normally used when increment is not needed
  - `while(ch!='q')`

{

...

`ch=getche();`

}

# do while loop

- do
  - statement*
  - while(*expression*);
- for(*initialization; conditional-test; increment*) *statement*;
- *initialization*;
  - do
  - {
  - statement*;
  - increment*;
  - }
  - while(*conditional-test*);

# do while loop

- Test is at the bottom
- Will execute at least once
- do

{

...

```
 ch=getche();
```

```
} while(ch!='q');
```

- Common error
  - Forgetting the semicolon (;) after while

# while loop

## do while

```
#include<stdio.h>
int main()
{
 int i=0;
 do
 {
 printf("%d\n", i);
 i++;
 }while(i<=9);
 return 0;
}
```

## for

```
#include<stdio.h>
int main()
{
 int i=0;
 for(i=1; i<=9; i++)
 printf("%d\n", i);
 return 0;
}
```

# for loop

```
/*prime number tester*/ /*test for factors*/
#include<stdio.h>
int main()
{
 int i, num,
 is_prime=1;
 printf("Enter the
number to test: ");
 scanf("%d", &num);
 for(i=2; i<=num/2; i++)
 if((num%i)==0) is_prime=0;
 if(is_prime==1)
 printf("%d is prime\n", num)
 else
 printf("%d is not prime\n",
num);
 return 0;
}
```

# for loop

```
/*prime number tester*/
#include<stdio.h>
int main()
{
 int i, num,
 is_prime=1;
 printf("Enter the
number to test: ");
 scanf("%d", &num);
```

```
/*test for factors*/
 for(i=2; i<=num/2; i++)
 if(!(num%i)) is_prime=0;
 if(is_prime)
 printf("%d is prime\n",
 num);
 else
 printf("%d is not prime\n",
 num);
 return 0;
}
```

# while loop

```
/*prime number tester*/
#include<stdio.h>
int main()
{
 int i, num,
 is_prime=1;
 printf("Enter the
number to test: ");
 scanf("%d", &num);
 /*test for factors*/
 i=2;
```

```
 while(i<=num/2)
 {
 if(!(num%i)) is_prime=0;
 i++;
 }
 if(is_prime)
 printf("%d is prime\n", num);
 else
 printf("%d is not prime\n",
 num);
 return 0;
}
```

# do while loop

```
/*prime number tester*/
#include<stdio.h>
int main()
{
 int i, num, is_prime=1;
 printf("Enter the number to
test: ");
 scanf("%d", &num);
 /*test for factors*/
 i=2;
 do
 {
 if(!(num%i)) is_prime=0;
```

```
 i++;
 }while(i<=num/2);
 if(is_prime)
 printf("%d is prime\n",
 num);
 else
 printf("%d is not prime\n",
 num);
 return 0;
}
```

It will show that 2 is not prime

# Use of break

```
/*prime number tester*/
#include<stdio.h>

int main()
{
 int i, num, is_prime=1;
 printf("Enter the number
to test: ");
 scanf("%d", &num);
```

```
/*test for factors*/
for(i=2; i<=num/2; i++)
 if(! (num%i))
 {
 is_prime=0;
 break;
 }
if(is_prime)
 printf("%d is prime\n",
 num);
else
 printf("%d is not prime\n",
 num);
return 0;
}
```

# Use of continue

```
#include<stdio.h>

int main()
{
 for(int i=1; i<=3; i++)
 {
 for(int j=1; j<=i; j++)
 {
 if(i==j) continue;
 printf("%d, %d\n", i, j);
 }
 }

 return 0;
}
```

## Output:

2, 1  
3, 1  
3, 2

# Input characters

- `getche()`/`getch()`/`getchar` can be used
- `getchar()`
  - Compiler dependent
  - waits for carriage return
  - Read only one char
  - Other input and carriage return will be in buffer
  - Subsequent input (e.g, `scanf`) will consume them.
  - Defined in `stdio.h`
- `getche()`/`getch()`
  - Return immediately after a key is pressed
  - Defined in `conio.h`

# Symbolic Constant

- A name that substitutes for a sequence of characters
- `#define name replacement`
- Any occurrence of *name* (not in quotes and not part of another name) will be replaced by corresponding *replacement*
- `#define PI 3.141593`
- `#define TRUE 1`
- `#define FALSE 0`

# Type conversion

- Operands that differ in type may undergo type conversion
- In general the result will be expressed in the highest precision possible
- `int i=7;`
- `float f=5.5;`
  
- `i+f : 12.5`

# Type cast

- Value of an expression can be converted to a different data type if desired.
- *(data type) expression*
  
- $(i+f)\%2$  : error
- $((int)(i+f))\%2$

# Recursion : Supplementary Content

Prepared By : Faias Satter  
Lecturer, University of Information Technology & Sciences

# Fibonacci Series

- also known as the Fibonacci sequence
- a mathematical sequence of numbers that starts with 0 and 1
- each subsequent number is the sum of the two preceding ones
- an infinite sequence of integers where each number (after the first two) is the sum of the two preceding ones
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

# C Program for Fibonacci Series

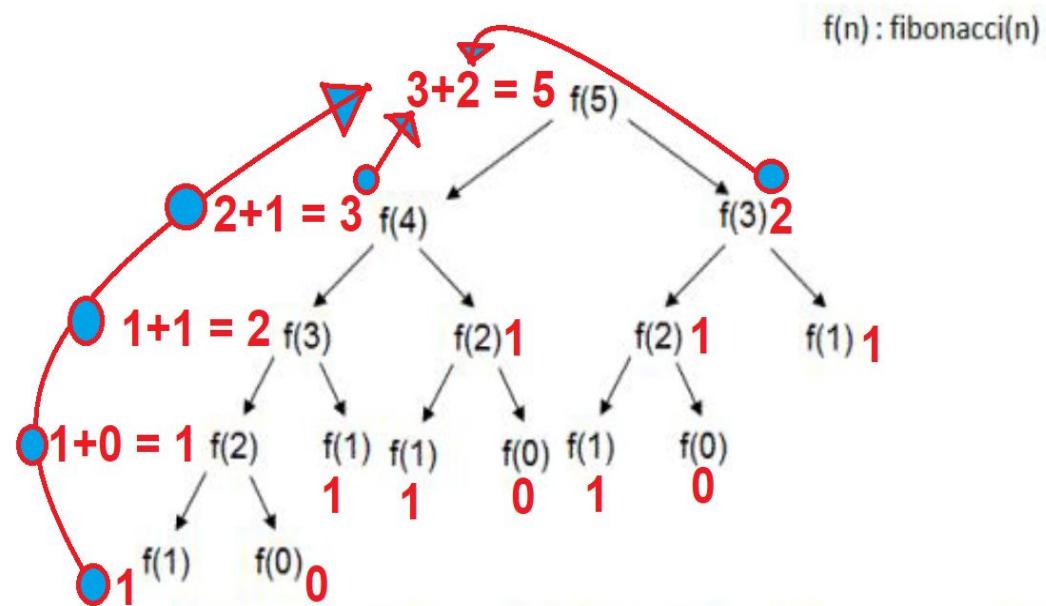
```
#include <stdio.h>

int main() {
 int n, first = 0, second = 1, next, c;
 printf("Enter the number of terms: ");
 scanf("%d", &n);
 printf("Fibonacci Series: ");
 for (c = 0; c < n; c++) {
 if (c <= 1) next = c;
 else { next = first + second; first = second; second = next; }
 printf("%d, ", next);
 }
 printf("\n");
 return 0;
}
```

# Recursive C Program for Fibonacci Series

```
#include <stdio.h>
int fibonacci(int n) {
 if (n <= 1) return n;
 else return (fibonacci(n - 1) + fibonacci(n - 2));
}
int main() {
 int n;
 printf("Enter the number of terms: ");
 scanf("%d", &n);
 printf("Fibonacci Series: ");
 for (int i = 0; i < n; i++) {
 printf("%d, ", fibonacci(i));
 }
 printf("\n"); return 0;
}
```

## Tree



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence

# Factorial

- mathematical function
- represents the product of all positive integers from 1 to a given positive integer
- denoted by an exclamation mark (!) following the number
- the factorial of a positive integer "n" is written as "n!" and is calculated as:
- $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$
- Examples :

$$5! \text{ (read as "five factorial")} = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$3! \text{ (read as "three factorial")} = 3 \times 2 \times 1 = 6$$

$$1! \text{ (read as "one factorial")} = 1$$

$$0! \text{ (by convention) is defined as 1}$$

# C Program for Factorial of a number

```
#include <stdio.h>

int main() {
 int n;
 unsigned long long factorial = 1; // WHY UNSIGNED LONG LONG ??!!
 printf("Enter a positive integer: ");
 scanf("%d", &n);
 if (n < 0) printf("Factorial is not defined for negative numbers.\n"); // Check if the input is negative
 else { // Calculate the factorial
 for (int i = 1; i <= n; i++)
 factorial *= i;
 printf("Factorial of %d = %llu\n", n, factorial); } }
```

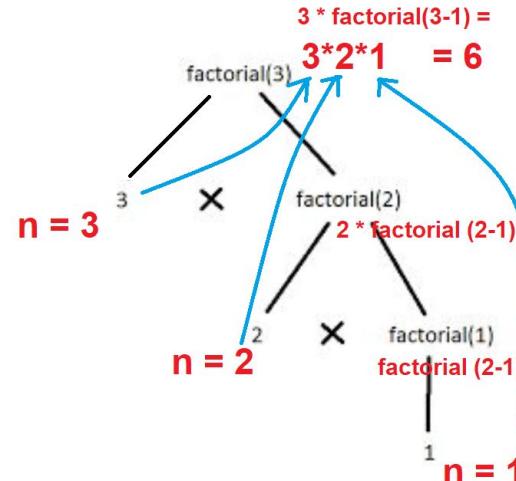
# Recursive C Program for Factorial of a number

```
#include <stdio.h>

unsigned long long factorial(int n);

int main() {
 int n; printf("Enter a positive integer: ");
 scanf("%d", &n);
 if (n < 0)
 { printf("Factorial is not defined for negative numbers.\n"); }
 else {
 unsigned long long result = factorial(n);
 printf("Factorial of %d = %llu\n", n, result);
 }
 return 0;
}
```

```
unsigned long long factorial(int n) {
 if (n == 0 || n == 1)
 { return 1; }
 else { return n * factorial(n - 1); }
}
```



# Resource Material

- What Is Recursion ?

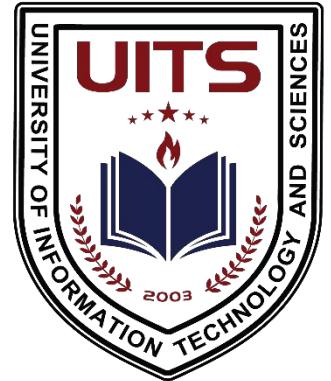
[https://sites.google.com/site/smilitude/recursion\\_and\\_dp](https://sites.google.com/site/smilitude/recursion_and_dp)

- Fibonacci Series :

<https://www.shafaetsplanet.com/?p=1022>

- Factorial :

<https://www.programiz.com/c-programming/examples/factorial-recursion>



# CSE-111

## Structured Programming Language

### ARRAY & STRING

Syeda Ajbina Nusrat  
Lecturer, CSE

# What is Array?

- An Array is a collection of same data type.
- An individual variable in the array is called an array element.
- The elements of an array are referred by a common name and are differentiate from one another by their position within an array.
- The elements of an array can be of any data type but all the elements in an array must be of the same type.

# Advantages of Array

- It is capable of storing many elements at a time
- It allows random accessing of elements i.e. any element of the array can be randomly accessed using indexes.

# Disadvantages of Array

- Predetermining the size of the array is a must.
- There is a chance of memory wastage or shortage.
- To delete one element in the array, we need to traverse throughout the array.

# Array declaration

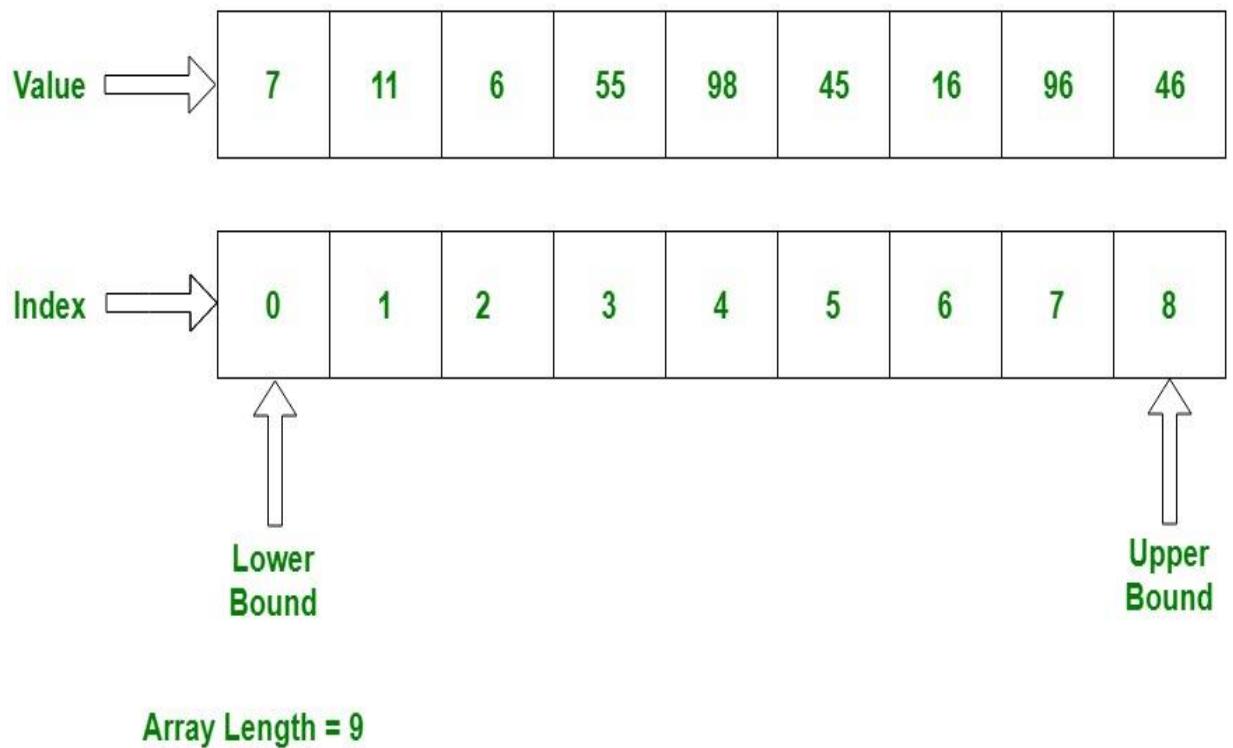
**Data\_type var\_name [array\_size];**

- array\_size specifies the number of elements in the array.
- Example:

```
int var[10];
float venus[20];
double earth[5];
char pluto[7];
```

```
int var[9];
```

- var[0] - 1st element, index 0
- var[1] - 2nd element, index 1
- var[2] - 3rd element, index 2
- var[3] - 4th element, index 3
- var[4] - 5th element, index 4
- var[5] - 6th element, index 5
- var[6] - 7th element, index 6
- var[7] - 8th element, index 7
- var[8] - 9th element, index 8



# Important things to remember in array

- Suppose, you declared the array of 10 students. For example: arr[10].
- You can use array members from arr[0] to arr[9]. But, what if you want to use element arr[10], arr[13] etc. Compiler may not show error using these elements but, may cause fatal error during program execution.

# Array Initialization (Compile time)

```
1 #include<stdio.h>
2
3 int main()
4 {
5 // Initialization type1
6 int var1[7] = {14, 22, 31, 18, 19, 7, 3};
7
8 // Initialization type2
9 int var2[] = {14, 22, 31, 18, 19, 7, 23};
10
11
12 return 0;
13 }
14 }
```

| 0  | 1  | 2  | 3  | 4  | 5 | 6  |
|----|----|----|----|----|---|----|
| 14 | 22 | 31 | 18 | 19 | 7 | 23 |

```
int var[7] = {14, 22, 31};
```

| 0  | 1  | 2  | 3 | 4 | 5 | 6 |
|----|----|----|---|---|---|---|
| 14 | 22 | 31 | 0 | 0 | 0 | 0 |

```
int var[7] = {9, 14, 0};
```

| 0 | 1  | 2 | 3 | 4 | 5 | 6 |
|---|----|---|---|---|---|---|
| 9 | 14 | 0 | 0 | 0 | 0 | 0 |

```
int var[7] = {9};
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | 0 | 0 | 0 | 0 | 0 | 0 |

```
int var[7] = {0};
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```
1 #include<stdio.h>
2 int main()
3 {
4 int var[4];
5 var[0] = 15;
6 var[1] = 13;
7 var[2] = 17;
8 var[3] = 19;
9
10 printf("%d ", var[0]);
11 printf("%d ", var[1]);
12 printf("%d ", var[2]);
13 printf("%d ", var[3]);
14
15 return 0;
16 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4];
5
6 scanf(" %d %d %d %d", &var[0], &var[1], &var[2], &var[3]);
7
8 printf("%d %d ", var[0], var[1]);
9 printf("%d %d ", var[2], var[3]);
10 }
```

---

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4];
5
6 scanf(" %d", &var[0]);
7 scanf(" %d", &var[1]);
8 scanf(" %d", &var[2]);
9 scanf(" %d", &var[3]);
10
11 printf("%d %d ", var[0], var[1]);
12 printf("%d %d ", var[2], var[3]);
13 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int i, var[4];
5
6 for(i=0 ; i<4 ; i++)
7 scanf(" %d", &var[i]);
8
9 printf("%d %d %d %d", var[0], var[1], var[2], var[3]);
10 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4], i;
5
6 for(i=0 ; i<4 ; i++)
7 scanf("%d", &var[i]);
8
9 for(i=0 ; i<4 ; i++)
10 printf("%d ", var[i]);
11 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[100], i, n;
5
6 printf("Number of Inputs: ");
7 scanf("%d", &n);
8
9 for(i=0; i<n; i++)
10 scanf("%d", &var[i]);
11
12 for(i=0; i<n; i++)
13 printf("%d ", var[i]);
14 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[100], i, n;
5 float avg=0;
6
7 printf("Number of Inputs: ");
8 scanf(" %d", &n);
9
10 for(i=0; i<n; i++) //Taking input of n numbers
11 scanf(" %d", &var[i]);
12
13 for(i=0; i<n; i++) //Calculating sum of n numbers
14 avg = avg + var[i];
15
16 avg = avg / n; //Calculating average of n numbers
17 printf("Avg: %.2f\n", avg);
18 }
```

```
1 #include<stdio.h>
2 void main()
3 { int var[100], i, n, min;
4
5 printf("Number of Inputs: ");
6 scanf("%d", &n);
7 for(i=0; i<n; i++)
8 {
9 scanf("%d", &var[i]); //Taking Input of n numbers
10 }
11 min = var[0];
12 for(i=0; i<n; i++)
13 {
14 if(var[i] < min)
15 {
16 min = var[i];
17 }
18 }
19
20 printf("Minimum %d", min);
21 }
```

# Character Array

- Character arrays have several unique features
- A character array can be initialized using a *string literal*

```
char string1[] = "first";
```

- “first” string literal contains five characters plus a special string termination character called null character (\0)
- string1 array actually has 6 elements-f, i, r, s, t and \0

- Character arrays can be initialized as follows as well-

```
char string1 []= {'f', 'i', 'r', 's', 't', '\0'};
```

- We can access individual characters in a string directly using array subscript notation. For example, string1 [3] is the character 's'

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| H | E | L | L | O |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

---

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[7], i;
5
6 for(i=0 ; i<5 ; i++)
7 scanf("%c", &str[i]);
8
9 for(i=0 ; i<5 ; i++)
10 printf("%c", str[i]);
11 }
12 }
```

OUTPUT H E L L O

- String is defined as a *NULL* terminated character array.
- Format Specifier: %s

```
char str[7];
```

|     |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| str | H | e | L | L | o |   |   |

Character Array

|     |   |   |   |   |   |    |   |
|-----|---|---|---|---|---|----|---|
|     | 0 | 1 | 2 | 3 | 4 | 5  | 6 |
| str | H | e | L | L | o | \0 |   |

String or NULL terminated character array

# String input - %s

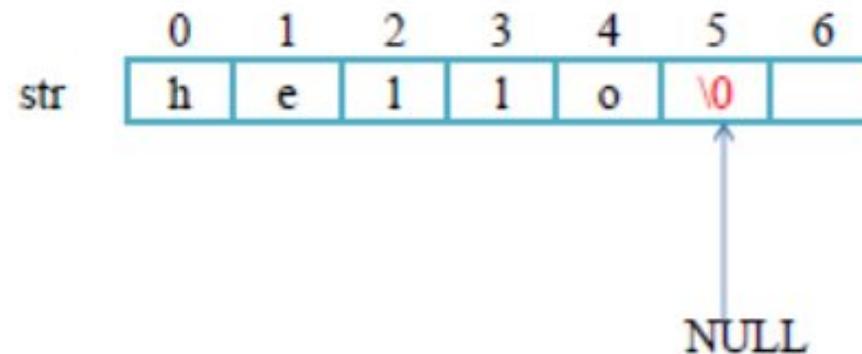
- We can also input a string directly from the keyboard using scanf () function and %s specifier.

```
char string1 [20];
scanf ("%s", string1);
```

- the name of the array is passed to `scanf()` without the preceding & used with other variables
- The & is normally used to provide `scanf()` with a variable's location in memory so a value can be stored there
- Array name is the address of the start of the array, therefore, & is not necessary
- `scanf()` reads characters from the keyboard until the first whitespace character is encountered
- `scanf()` takes upto the first whitespace

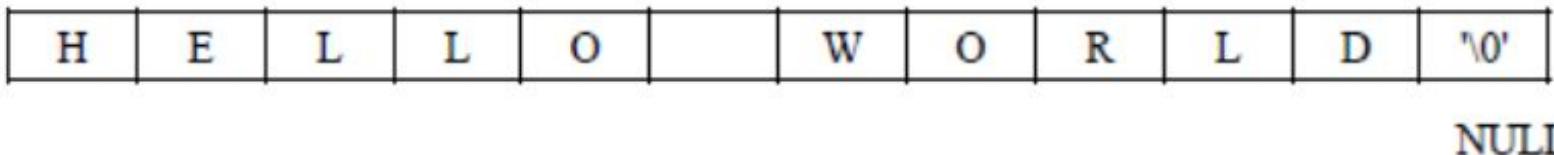
# String input - %s

```
#include<stdio.h>
void main()
{
 char str[7];
 scanf("%s", &str);
 printf("%s", str);
}
```



# Reading a line

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[7], i;
5 gets(str);
6 puts(str);
7 }
```



```
1 #include<stdio.h>
2 void main()
3 {
4 char str[20], i;
5 gets(str);
6
7 for(i=0 ;str[i] != '\0' ; i++)
8 printf("%c", str[i]);
9
10 }
```

|   |   |   |   |   |  |   |   |   |   |   |      |
|---|---|---|---|---|--|---|---|---|---|---|------|
| H | E | L | L | O |  | W | O | R | L | D | '\0' |
|---|---|---|---|---|--|---|---|---|---|---|------|

NULL

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[20], i;
5 gets(str);
6
7 for(i=0 ;str[i] != NULL; i++)
8 printf("%c", str[i]);
9
10 }
```

|   |   |   |   |   |  |   |   |   |   |   |    |
|---|---|---|---|---|--|---|---|---|---|---|----|
| H | E | L | L | O |  | W | O | R | L | D | \0 |
|---|---|---|---|---|--|---|---|---|---|---|----|

NULL

# Library functions (String)

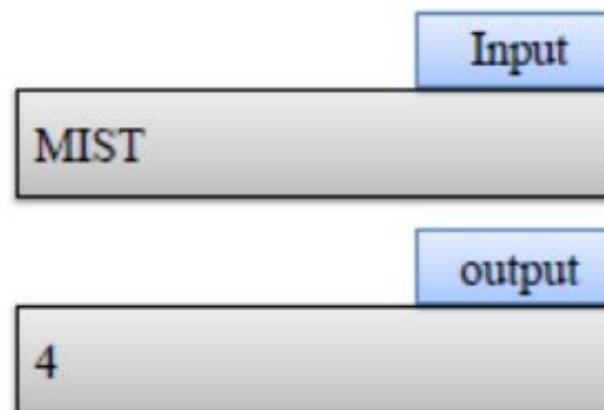
Every C compiler provides a large set of string handling library functions, which are contained in the header file “string.h”

- `strlen` : Finds the length of the string
- `strcat` : Appends one string at the end of the other
- `strcpy( to, from)`: Copies one string into another
- `strncpy` : Copies first n characters of one string into another
- `strcmp (s1, s2)`: Compares two strings
  - Returns 0 if same
  - -ve if s1 less than s2
  - +ve if s1 greater than s2
- `strchr` : Finds first occurrence of a given character in a string

# Strlen()

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

```
int main()
{
 char str[100];
 int len;
 gets(str);
 len = strlen(str);
 printf("%d", len);
}
```

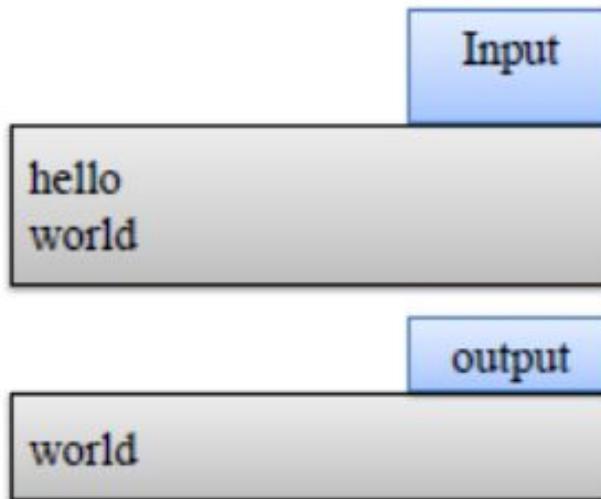


# Strcpy()

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

void main()
{
 char str1[100], str2[100];
 gets(str1);
 gets(str2);
 strcpy(str1, str2);

 puts(str1);
}
```

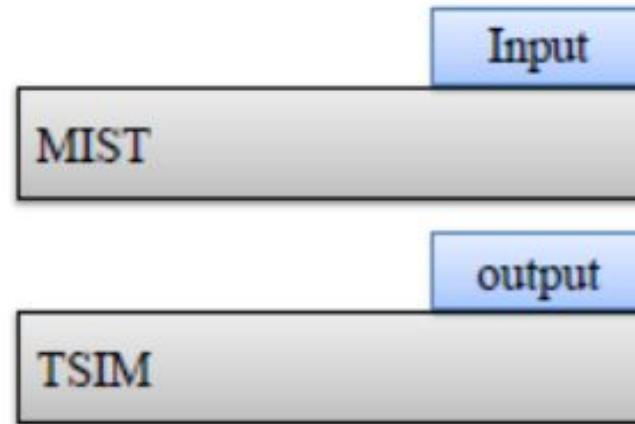


# Strrev()

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

void main()
{
 char str[100];
 gets(str);
 strrev(str);

 printf("%s", str);
}
```

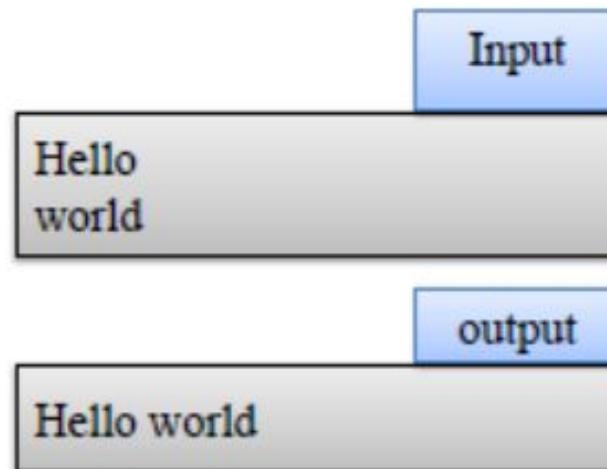


# Strcat()

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

void main()
{
 char str1[100], str2[100];
 gets(str1);
 gets(str2);
 strcat(str1, str2);

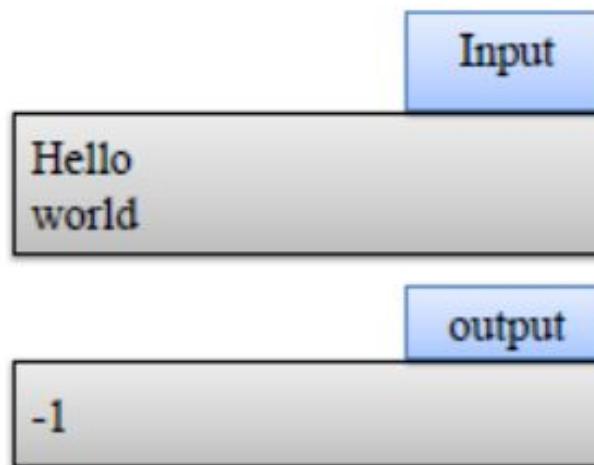
 printf("%s", str1);
}
```



# strcmp()

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

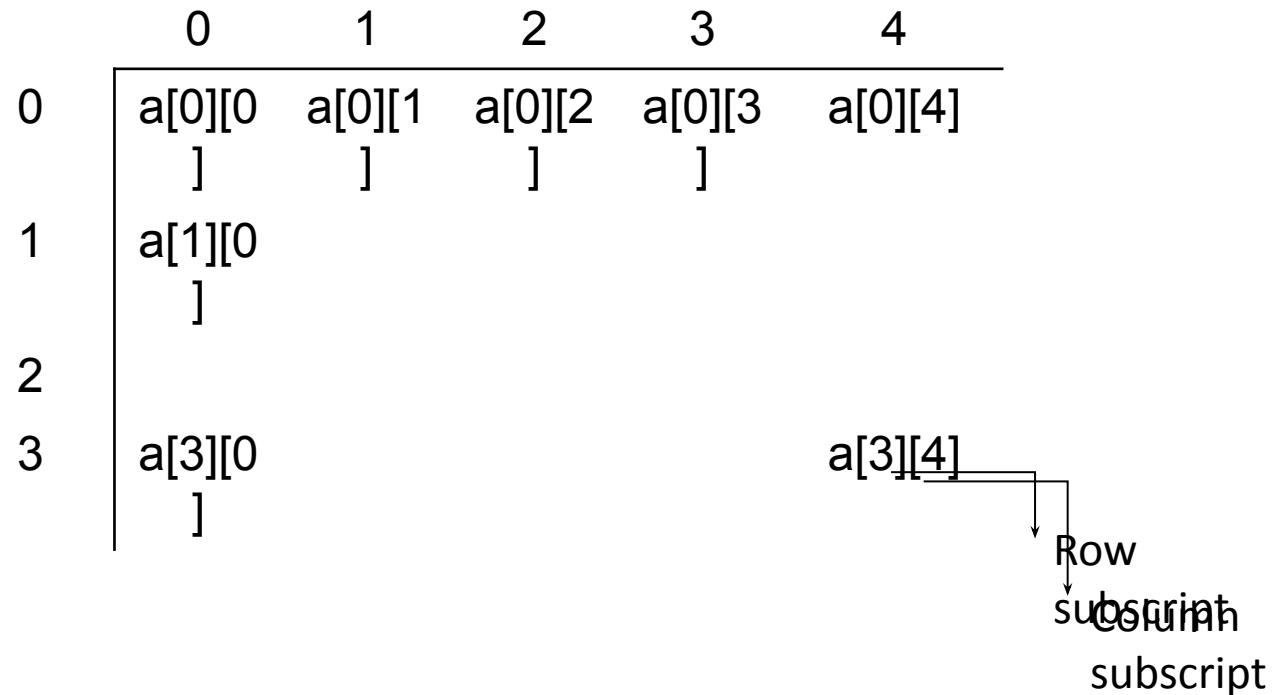
void main()
{
 char str1[100], str2[100];
 int r;
 gets(str1);
 gets(str2);
 r = strcmp(str1, str2);
 printf("%d", r);
}
```



# 2-D Array

- Array of one dimensional arrays
- Row, column format
- Accessed a row at a time from left to right

Example: float yeartemp[12][31];



# Example

```
#include<stdio.h>

int main()
{
 int td[4][5];
 int i, j;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++)
 td[i][j]=i;
 for(i=0; i<4; i++)
 {
 for(j=0; j<5; j++)
 printf("%d ", td[i][j]);
 printf("\n");
 }
 return 0;
}
```

|                |
|----------------|
| <b>Output:</b> |
| 0 0 0 0 0      |
| 1 1 1 1 1      |
| 2 2 2 2 2      |
| 3 3 3 3 3      |

# Example

```
#include<stdio.h>

int main()
{
 int td[4][5];
 int i, j;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++)
 td[i][j]=i*j;
 for(i=0; i<4; i++)
 {
 for(j=0; j<5; j++)
 printf("%d ", td[i][j]);
 printf("\n");
 }
 return 0;
}
```

**Output:**  
0 0 0 0  
0 1 2 3 4  
0 2 4 6 8  
0 3 6 9 12

# Example

- Initialization:

```
int sqr[3][3] ={
 1,2,3,
 4,5,6,
 7,8,9
};
```

|           | Col no. 0 | Col no. 1 | Col no. 2 |
|-----------|-----------|-----------|-----------|
| Row no. 0 | 1         | 2         | 3         |
| Row no. 1 | 4         | 5         | 6         |
| Row no. 2 | 7         | 8         | 9         |

- Initialization:

- Specify all but the leftmost dimension

```
int sqr[][3] ={
 1,2,3,
 4,5,6,
 7,8,9
};
```

- Initialization:

```
int sqr[3][3] ={
 {1,2,3},
 {4,5,6},
 {7,8,9}
};
```

```
int sqr[3][3] ={1,2,3,4,5,6,7,8,9};
int sqr[][][3] ={1,2,3,4,5,6,7,8,9};
```

- Initialization:

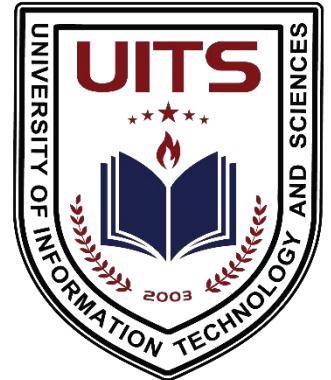
```
int sqr[3][] = {1,2,3,4,5,6,7,8,9};
```

```
int sqr[][] = {1,2,3,4,5,6,7,8,9};
```

- This would never work

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET



# CSE-111

## Structured Programming Language

### FUNCTION

Syeda Ajbina Nusrat  
Lecturer, CSE

- In programming, a function is a segment that groups code to perform a specific task.
- A C program must have main( ).
- Without main() function, there is technically no C program.

# Advantages

- Reduce code length
- Easy to detect a faulty function
- A function may be used by many other program
- Avoids the need for redundant (repeated) programming of the same instructions.
- Enables a programmer to build a customized library of frequently used routines or of routines containing

# Types of functions

- There are two types of functions in C programming:
  - Library function
  - User defined function

# Library functions

- Library functions are the in-built function in C programming system.
- In C, we used many library functions - printf (), scanf (), clrscr (), or getch ()
- printf() prints on the output
- scanf () takes input from the user and assigns that input to a variable by going to the variable's address
- clrscr () clears the output buffer
- getch () waits for a key-stroke from the user
- If you think uniform one thing is common to say- the function does one particular job. Isn't it?

# User defined functions

- C allows programmer to define their own function according to their requirement.
- These types of functions are known as user-defined functions.
- Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program.
- Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

# Important concepts

- Function Definition
- Function Call
- Function Declaration

```
1 #include<stdio.h>
2
3 int sumFunc(int m, int n)
4 {
5 int sum=0, i;
6 for(i=m; i<=n; i++)
7 {
8 sum = sum + i;
9 }
10
11 return sum;
12 }
13 int main()
14 {
15 int a = sumFunc(1, 10);
16 printf("First Sum %d\n", a);
17
18 printf("Second Sum %d\n", sumFunc(5, 20));
19 printf("Third Sum %d\n", sumFunc(2, 6));
20 printf("Fourth Sum %d\n", sumFunc(7, 9));
21
22 return 0;
23 }
```



Function Definition



Function Call

# Elements of function definition

- Function Name
- Function Type
- List of Parameters
- Local Variable Declaration
- Function Statements
- A Return Statement

```
1 #include<stdio.h>
2
3 /*Forward Declaration*/
4 void sumFunc(int m, int n);
5
6 int main()
7 {
8 sumFunc(1, 10);
9 return 0;
10}
11
12 /*Function Definition*/
13 void sumFunc(int m, int n)
14 {
15 int sum=0, i;
16 for(i=m; i<=n; i++)
17 sum = sum + i;
18 printf("First Sum %d\n", sum);
19}
20
```



Function  
Declaration or  
Function  
Prototype

# How User defined function works?

```
#include <stdio.h>
void function_name(){

}

int main() {

 step 1
 function_name(); step 1

}

}
```

step 2

The diagram illustrates the execution flow between the user-defined function and its caller. It consists of two main parts enclosed in a large rectangular frame:

- Function Definition (Top):** Contains the code: `#include <stdio.h>` and `void function\_name(){ ..... }`. A horizontal arrow points from the left towards the opening brace of the function body.
- Function Call in Main (Bottom):** Contains the code: `int main() { ..... }` followed by a call to the function: `function\_name();`. This call is highlighted with a red box. A horizontal arrow points from the right towards the closing brace of the function call.

Two labels are present:

- step 2**: Located to the left of the main function's opening brace.
- step 1**: Located above the function call `function\_name();` and to the right of the main function's closing brace.

# Some insights of User defined function

- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls.
- After each function has done its job, control returns to the place of calling that function.
- Well, then, should all the function calls take place inside the main () function? No! Functions defined by the user can call other functions as well.

- C program is a collection of one or more functions.
- A function gets called when the function name is followed by a semicolon.

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

void argentina();

main()
{
 argentina();
}
```

- A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

```
argentina()
{
 statement 1 ;
 statement 2 ;
 statement 3 ;
}
```

- Any function can be called from any other function. Even **main( )** can be called from other functions.

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

void message();

void main(){
 message();
}

message(){
 printf ("\nCan't imagine life without C") ;
 main();
}
```

- A function can be called any number of times.

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

void message();

main()
{
 message() ;
 message() ;
}

message()
{
 printf ("\nJewel Thief!!") ;
}
```

- The order in which the functions are defined in a program and the order in which they get called need not necessarily be same

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

void message1();
void message2();
main(){
 message1();
 message2();
}

message2(){
 printf ("\nBut the butter was bitter") ;
}
message1(){
 printf ("\nMary bought some butter") ;
}
```

- A function can be called from other function, but a function cannot be defined in another function. Thus, the following program code would be wrong, since **argentina( )** is being defined inside another function, **main( )**.

```
main(){
 printf ("\nI am in main") ;
 argentina(){
 printf ("\nI am in argentina") ;
 }
}
```

- There is no restriction on the number of **return** statements that may be present in a function
- **return** statement need not always be present at the end of the called function

```
fun()
{
 char ch;

 printf ("\nEnter any alphabet");
 scanf ("%c", &ch);

 if (ch >= 65 && ch <= 90)
 return (ch);
 else
 return (ch + 32);
}
```

- All the following are valid **return** statements
- In the last statement a garbage value is returned to the calling function since we are not returning any specific value
- Note that in this case the parentheses after **return** are dropped

```
return (a) ;
return (23) ;
return (12.34) ;
return ;
```

- A function can return only one value at a time
- the following statements are invalid

```
return (a, b) ;
return (x, 12) ;
```

# Categories of function

- A function can be categorized depending on argument and return type—
  - Function with no argument and no return value
  - Function with argument and no return value
  - Function with no argument and return value
  - Function with argument and return value

# Function with no argument and no return value

```
#include <stdio.h>
int main()
{
 add();
 return 4;
}

void add()
{
 int a=5, b=2, c;
 c=a+b;
 printf("%d\n", c);
}
```

# Function with argument and no return value

```
#include <stdio.h>
void add(float a, float b);
int main()
{
 float a=2.3, b=4.6;
 add(a, b);
 return 4;
}
void add(float a, float b)
{
 float c;
 c=a+b;
 printf("%f", c);
}
```

# Function with no argument and return value

```
15 int sumFunc()
16 {
17 int sum=0, i;
18 for(i=1; i<=100; i++)
19 sum = sum + i;
20
21 return sum;
22
23 }
24
```

# Function with argument and return value

```
#include <stdio.h>
float add(float a, float b)
{
 float c;
 c=a+b;
 return c;
}
int main()
{
 float a=2.3, b=4.6, c;
 c=add(a, b);
 printf ("%f", c);
 return 4;
}
```

# Function call

- Two types
  - 1. Call by value
  - 2. Call by reference

# Call by value

- If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include <stdio.h>

void call_by_value(int x) {
 printf("Inside call_by_value x = %d before adding 10.\n", x);
 x += 10;
 printf("Inside call_by_value x = %d after adding 10.\n", x);
}

int main() {
 int a=10;

 printf("a = %d before function call_by_value.\n", a);
 call_by_value(a);
 printf("a = %d after function call_by_value.\n", a);
 return 0;
}
```

The output of this call by value code example will look like this:

```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

# Call by reference

- If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example

```
#include <stdio.h>

void call_by_reference(int *y) {
 printf("Inside call_by_reference y = %d before adding 10.\n", *y);
 (*y) += 10;
 printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}

int main() {
 int b=10;

 printf("b = %d before function call_by_reference.\n", b);
 call_by_reference(&b);
 printf("b = %d after function call_by_reference.\n", b);

 return 0;
}
```

The output of this call by reference source code example will look like this:

```
b = 10 before function call_by_reference.
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.
```

# Advantages of user defined functions

- User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

# Library Functions vs User-defined Functions

- A user defined function is something the programmer has defined to perform a specific action or calculation in your program.

Example, you need to calculate somebody's age to the nearest minute. You create a calculateAge function:

```
double calculateAge(Date dateOfBirth)
{
}
```

So it is specific to your needs at this time.

A library function has been written by someone else, and is reusable by many programmers to solve the same problem over and over.

Example: strcpy in C. If you are using C, you will never need to write functions for string manipulation because somebody else did that a long time ago...

# Math Library function

- Math library functions allow the programmer to perform certain common mathematical calculations
- A programmer desiring to calculate and print the square root of 900.0 can write-

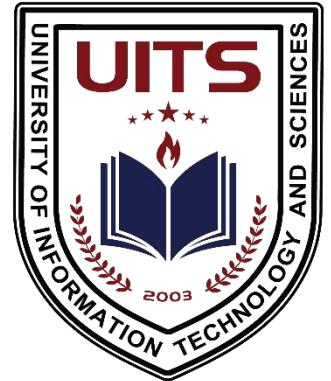
```
printf("%.2f", sqrt(900.0));
```

- When the statement is executed, `sqrt()` is called to calculate the square root of the number contained in the parenthesis (900.0)
- The number 900.0 is the argument of `sqrt()` function
- The statement will print 30.00

| Function               | Description                                        |
|------------------------|----------------------------------------------------|
| <code>exp(x)</code>    | Exponential function                               |
| <code>log(x)</code>    | Natural logarithm (base e)                         |
| <code>log10(x)</code>  | Logarithm (base 10)                                |
| <code>fabs(x)</code>   | Absolute value                                     |
| <code>ceil(x)</code>   | Rounds x to the smallest integer not less than x   |
| <code>floor(x)</code>  | Rounds x to the largest integer not greater than x |
| <code>pow(x,y)</code>  | x to the power y                                   |
| <code>fmod(x,y)</code> | Remainder of x/y as a floating point number        |
| <code>sin(x)</code>    | Trigonometric sine of x                            |
| <code>cos(x)</code>    | Trigonometric cosine of x                          |
| <code>tan(x)</code>    | Trigonometric tangent of x                         |

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET



# CSE-111

## Structured Programming Language

### POINTER

Syeda Ajbina Nusrat  
Lecturer, CSE

# What is pointer?

- Pointers are variables that stores *memory addresses* of another entity as their values.
- A variable name *directly* references a value.
- A pointer *directly* refers to a memory location, *indirectly* refers to a value. Referencing a value through a pointer is called *indirection*.
- A pointer variable must be declared before it can be used.

# Concept of Computer Memory

- Computers store data in memory slots
- Each slot has an *unique address*
- Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable.
- Example:

```
int quality = 179;
```

- $A[2] = 'a'$

| memory address | memory content |
|----------------|----------------|
| 1000           | quality : 179  |
| 1004           | A[2] : 'a'     |
|                |                |

- Altering the value of a variable is indeed changing the content of the memory

e.g. quality = 40; A[2] = 'z';

| memory address | memory content |
|----------------|----------------|
| 1000           | quality : 40   |
| 1004           | A[2] : 'z'     |

- `int i = 5;`
  - `int *ptr;` /\* declare a pointer variable \*/
  - `ptr = &i;` /\* store address-of i to ptr \*/
  - `printf("*ptr = %d\n", *ptr);` /\* refer to referee of ptr \*/

- **ptr** is a variable storing **an address**
- **ptr** is **NOT** storing the actual value of **i**

```
int i = 5;
```

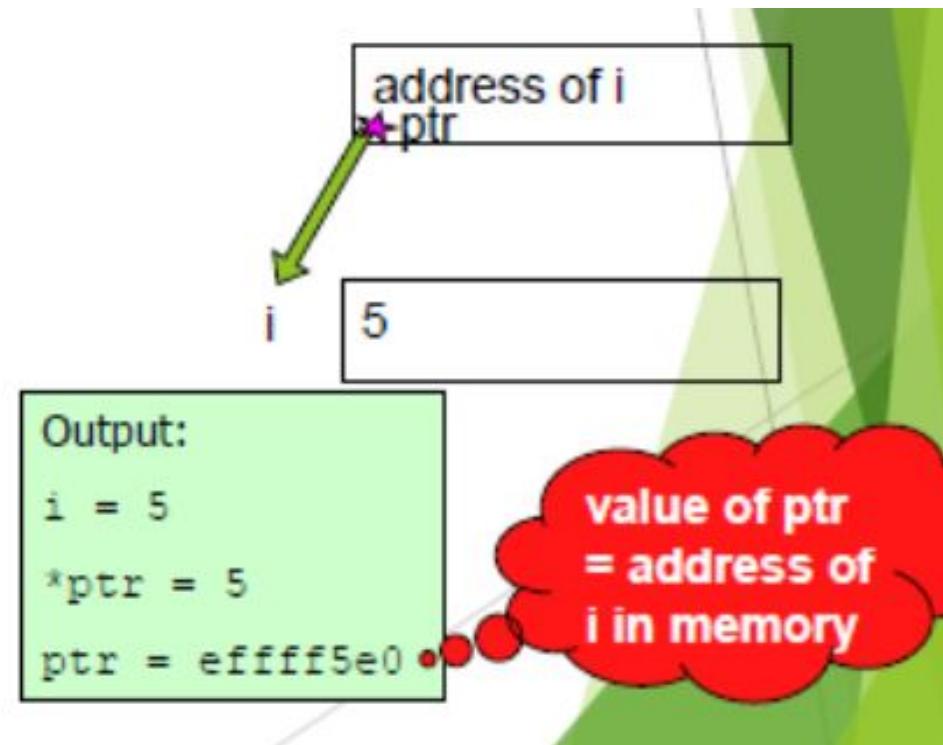
```
int *ptr;
```

```
ptr = &i;
```

```
printf("i = %d\n", i);
```

```
printf("*ptr = %d\n", *ptr);
```

```
printf("ptr = %p\n", ptr);
```



# Why do we need pointer?

- Increases execution speed thus decreases program execution time.
- Reduces length and complexity of the program.
- To return multiple values from a function via function arguments.
- Helps manipulating dynamic data structures like stack, linked lists, queues, structures etc.
- Supports dynamic memory management.

# Declaring a pointer variable

- `data_type *pt_name;`
  - \* : (asterisk) denotes that pt\_name is a pointer type variable
  - pt\_name needs a memory location.
  - pt\_name points a variable of type data\_type.
  - Example: `int *p;`
- Styles:
  - `int* p;`   `int * p;`   `int *p;`

# Use of & and \*

- When is & used?
- When is \* used?
- & -- "*address operator*" which gives or produces the memory address of a data variable
- \* -- "*dereferencing operator*" which provides the contents in the memory location specified by a pointer

# Initialization of Pointer Variables

- We use assignment operator to initialize the variable, once a variable is declared
  - int quantity;
  - int \*p;
  - p= &quantity;
- Initialization with the declaration:
  - int \*p = &quantity;
- Initialization of pointer in one step:
  - int x, \*p = &x; (VALID)
  - int \*p = &x , x; (INVALID)
- We can define a pointer with an initial value of NULL or 0 (zero). Initialization with other constant value is wrong step.

# NULL pointer in C

- At the very high level, we can think of NULL as a null pointer which is used in C for various purposes. Some of the most common use cases for NULL are
  - a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
  - b) To check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.
  - c) To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- The example of a is

```
int *pInt = NULL;
```

# Pointers and Function

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.

# Pointers and Function (contd.) Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include <stdio.h>

void call_by_value(int x) {
 printf("Inside call_by_value x = %d before adding 10.\n", x);
 x += 10;
 printf("Inside call_by_value x = %d after adding 10.\n", x);
}

int main() {
 int a=10;

 printf("a = %d before function call_by_value.\n", a);
 call_by_value(a);
 printf("a = %d after function call_by_value.\n", a);
 return 0;
}
```

The output of this call by value code example will look like this:

```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

# Pointers and Function (contd.) Call by Reference

If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example:

```
#include <stdio.h>

void call_by_reference(int *y) {
 printf("Inside call_by_reference y = %d before adding 10.\n", *y);
 (*y) += 10;
 printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}

int main() {
 int b=10;

 printf("b = %d before function call_by_reference.\n", b);
 call_by_reference(&b);
 printf("b = %d after function call_by_reference.\n", b);

 return 0;
}
```

The output of this call by reference source code example will look like this:

```
b = 10 before function call_by_reference.
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.
```

# Pointers and Function (Example)

```
#include <stdio.h>
void swap (int a, int b) ;
int main ()
{
 int a = 5, b = 6;
 printf("a=%d b=%d\n",a,b) ;
 swap (a, b) ;
 printf("a=%d b=%d\n",a,b) ;
 return 0 ;
}
void swap(int a, int b) {
 int temp;
 temp= a; a= b; b = temp ;
 printf ("a=%d b=%d\n", a, b);
}
```

*Results:* a=5 b=6 a=6 b=5 a=5 b=6

# Pointers and Function (Example)

```
#include <stdio.h>
void swap (int *a, int *b) ;
int main ()
{
 int a = 5, b = 6;
 printf("a=%d b=%d\n",a,b) ;
 swap (&a, &b) ;
 printf("a=%d b=%d\n",a,b) ;
 return 0 ;
}
```

```
void swap(int *a, int *b)
{
 int temp;
 temp= *a;
 *a= *b;
 *b = temp ;
 printf ("a=%d b=%d\n", *a, *b);
}
```

*Results:*

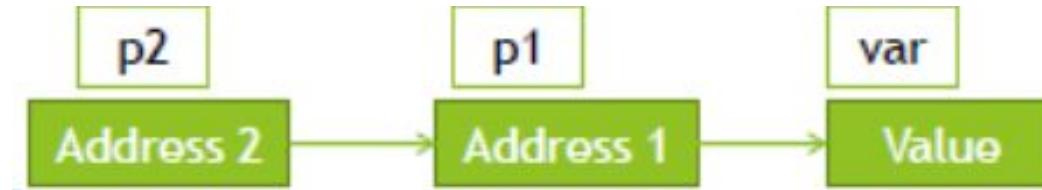
a=5 b=6

a=6 b=5

a=6 b=5

# Chain of Pointer

- Is it possible to point a pointer?
- `int **p;`



- This tells the compiler that **p2** is a pointer to a pointer of `int` type.
  - Remember: **p2** is not a pointer to an integer.
  - Pointer variable **p2** contains the address of the pointer variable **p1**, which points to the location that contains the desired value. This is known as ***multiple indirections***.

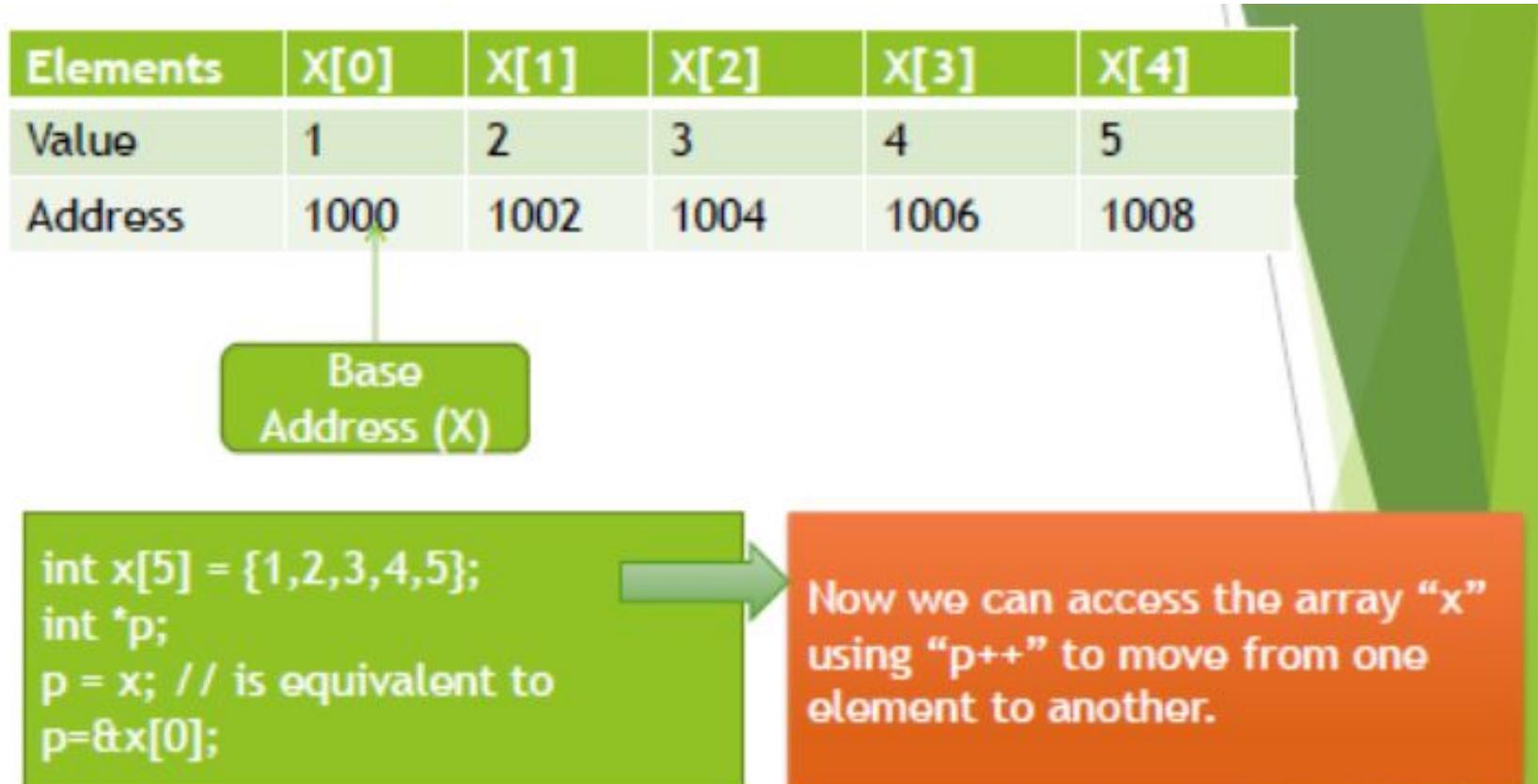
```
#include<stdio.h>
void main()
{
 int x, *p1, **p2;
 x = 100;
 p1 = &x;
 p2 = &p1;
 printf("%d", **p2);
}
```

Output:  
100

# Pointer Increment and Scale Factor

- A pointer can be incremented:
  - $P1 = P2 + 2;$
  - $P1 = P1 + 1;$
  - $P1++;$
- $P1 ++$  what does it mean?
  - This expression will cause the pointer  $P1$  to point to the next value of its type.
- Example:  $P1$  is an integer pointer, with an initial value 2800, then after the operation  $P1++$ , the value of  $P1$  will be 2802.
  - When we increment a pointer, its value is increased by the length of the type that it points to. The length called the scale factor.
  - # of bytes of various data types depends on the system. It is found by `sizeof` operator.

# Pointers and Arrays



# Pointers and Arrays

Write a program using pointers to compute the sum of all elements stored in an array

```

#include<stdio.h>
void main()
{
int *p, sum, i;
int x[5] = {5,9,6,3,7};
i=0; sum=0;
p = x;
printf("Element Value Address\n\n");
while(i < 5)
{
printf("x[%d] %d %d\n",i,*p,p);
sum=sum+*p;
i++; p++;
}
printf("\nSum = %d\n",sum);
printf("\n&x[0] = %d\n",&x[0]);
printf("\np = %d\n",p);
}

```

[This o/p may differs from system to system.]

Output:

Element Value Address

|      |   |     |
|------|---|-----|
| X[0] | 5 | 166 |
| X[1] | 9 | 168 |
| X[2] | 6 | 170 |
| X[3] | 3 | 172 |
| X[4] | 7 | 174 |

Sum=30

&x[0]=166

p=176

# Pointers and Strings

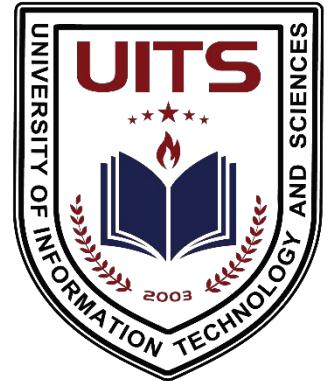
| Elements | X[0] | X[1] | X[2] | X[3] | X[4] |
|----------|------|------|------|------|------|
| Value    | g    | o    | o    | d    | y    |
| Address  | 1000 | 1004 | 1008 | 1012 | 1016 |

Base  
Address

```
char *X = "goody";
//variable X is a string pointer,
not a string.
```

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET



# CSE-111

## Structured Programming Language

### STRUCTURE

Syeda Ajbina Nusrat  
Lecturer, CSE

# What is structure?

- 1.A structure is an aggregate data type that is composed of two or more variables called members.
- 2.Unlike an array in which each element is of the same type, each member of a structure can have its own type, which may differ from types of the other members.

# General form:

```
struct tag-name
{
 type member1;
 type member2;
 .
 .
 type memerN;
};
```

3 parts of a structure:  
1. struct  
2. Tag-name  
3. Structure member

# Practical Example

## Student Info

|                |                 |
|----------------|-----------------|
| Student name:  | char name[100]; |
| Dept name:     | char dept[10];  |
| Id No:         | int id;         |
| Cgpa:          | float cgpa;     |
| Total Credits: | int credit;     |
| Total Courses: | int courses;    |

```
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int courses;
};
```

# Example-1

```
#include<stdio.h>
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int crs;
} ab, cd, x, y;

void function()
{
 struct student var1, var2;
}

void main()
{
 struct student var, var1;
}
```

## Example-2 (taking input)

```
#include<stdio.h>
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int crs;
};
```

```
void main()
{
 struct student var;
 gets(var.name);
 gets(var.dept);
 scanf(" %d", &var.id);
 scanf(" %f", &var.cgpa);
 scanf(" %d", &var.credit);
 scanf(" %d", &var.crs);
}
```

# Example-3 (taking input and giving output)

```
void main()
{
 struct student var;
 gets(var.name);
 gets(var.dept);
 scanf(" %d", &var.id);
 scanf(" %f", &var.cgpa);
 scanf(" %d", &var.credit);
 scanf(" %d", &var.crs);
 puts(var.name);
 puts(var.dept);
 printf(" %d", var.id);
 printf(" %f", var.cgpa);
 printf(" %d", var.credit);
 printf(" %d", var.crs);
}
```

# Example-4 (Structure with function)

```
#include<stdio.h>
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int crs;
};
```

```
void to_print(struct student var)
{
 puts(var.name);
 puts(var.dept);
 printf(" %d %f\n", var.id, var.cgpa);
 printf(" %d %d\n", var.credit, var.crs);
}
void main()
{
 struct student var;
 gets(var.name);
 gets(var.dept);
 scanf(" %d %f", &var.id, var.cgpa);
 scanf(" %d %d", &var.credit, &var.crs);
 to_print(var);
}
```

# Example-5 (Structure with array)

```
#include<stdio.h>
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int crs;
};
```

```
void main()
{ struct student var[10] ;
 int i;
 for(i=0 ; i<10 ; i++)
 {
 gets(var[i].name);
 gets(var[i].dept);
 scanf(" %d", &var[i].id);
 scanf(" %f", &var[i].cgpa);
 scanf(" %d", &var[i].credit);
 scanf(" %d", &var[i].crs);
 }
}
```

# Example-6 (Structure with array)

```
void main()
{ student var[10] ; int i;
for(i=0 ; i<10 ; i++)
{
 gets(var[i].name);
 gets(var[i].dept);
 scanf(" %d", &var[i].id);
 scanf(" %f", &var[i].cgpa);
 scanf(" %d", &var[i].credit);
 scanf(" %d", &var[i].crs);
}
for(i=0 ; i<10 ; i++)
{
 puts(var[i].name);
 puts(var[i].dept);
 printf(" %d", var[i].id);
 printf(" %f", var[i].cgpa);
 printf(" %d", var[i].credit);
 printf(" %d", var[i].crs);
}
```

## Example-7 (Structure with array)

```
#include<stdio.h>
struct student
{
 char name[100];
 char dept[10];
 int id;
 float cgpa;
 int credit;
 int crs;
};

void to_print(struct student var)
{
 puts(var.name);
 puts(var.dept);
 printf(" %d %.2f\n", var.id,
 var.cgpa);
 printf(" %d %d\n", var.credit,
 var.crs);
}
```

```
struct student to_input()
{
 struct student X;
 gets(X.name);
 gets(X.dept);
 scanf(" %d", &X.id);
 scanf(" %f", &X.cgpa);
 scanf(" %d", &X.credit);
 scanf(" %d", &X.crs);
 return X;
}
```

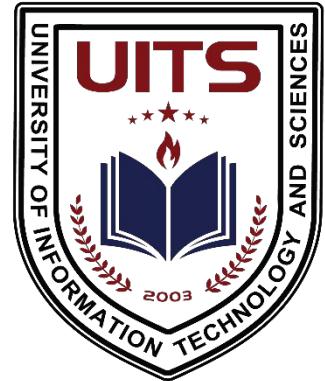
```
void main()
{
 struct student var;
 var=to_input();
 to_print(var);
}
```

# Practice Problem

You have to build a system that will keep track of the records of all bowlers in T20 world cup. For each bowler you have to store name, number of matches played, total number of bowled overs, total number of wickets, run per over. Create a structure for this purpose. Take input from user of the total number of bowlers and then information for each bowler.

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET



# CSE-111

## Structured Programming Language

### RECURSION

Syeda Ajbina Nusrat  
Lecturer, CSE

- Recursion is a process in which a module achieves a repetition of algorithmic steps by calling itself.
- Recursion is something of a divide and conquer, top-down approach to problem solving.
- It divides the problem into pieces

# Advantages and Disadvantages

## Advantage and Disadvantage

- **Advantages**

- \*Easy solution for recursively defined problems.
- \*Complex programs can be easily written in less code.

- **Disadvantage**

- \*Recursive code is difficult to understand and debug
- \*Terminating condition is must, otherwise it will go in infinite loop.
- \*Execution speed decreases because of function call and return activity many times.

# 4 fundamental rules

1. Base Case: Always have at least one case that can be solved without recursion.
2. Make Progress: Any recursive call must progress towards a base case.
3. Always Believe: Always assume the recursive call works.
4. Compound Interest Rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.

# Basic form

```
void recurse()
{
 recurse(); //Function calls itself
}

intmain ()
{
 recurse(); //Sets off the recursion
}
```

# How does it work?

- 1.The module calls itself
- 2.New variables and parameters are allocated storage on the stack
- 3.Function code is executed with the new variables from its beginning.  
It does not make a new copy of the function. Only the arguments and local variables are new.
- 4.As each call returns, old local variables and parameters are removed from the stack.
- 5.Then execution resumes at the point of the recursive call inside the function.

# To build a recursion tree

- root = the initial call
- Each node = a particular call

Each new call becomes a child of the node that called it

- A tree branch (solid line) = a call-return path between any 2 call instances

# Factorial

Factorial (n):

IF (n = 0)

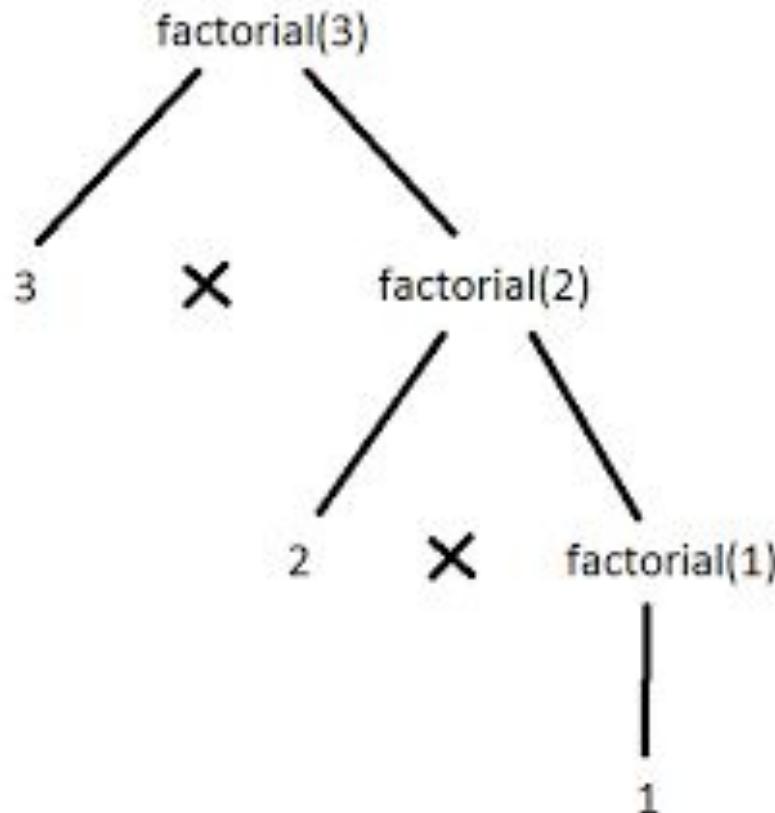
    RETURN 1

ELSE

    RETURN n \* Factorial (n-1)

Calculates  $n * (n-1) * (n-2) * \dots * (1) * (1)$

# Tree

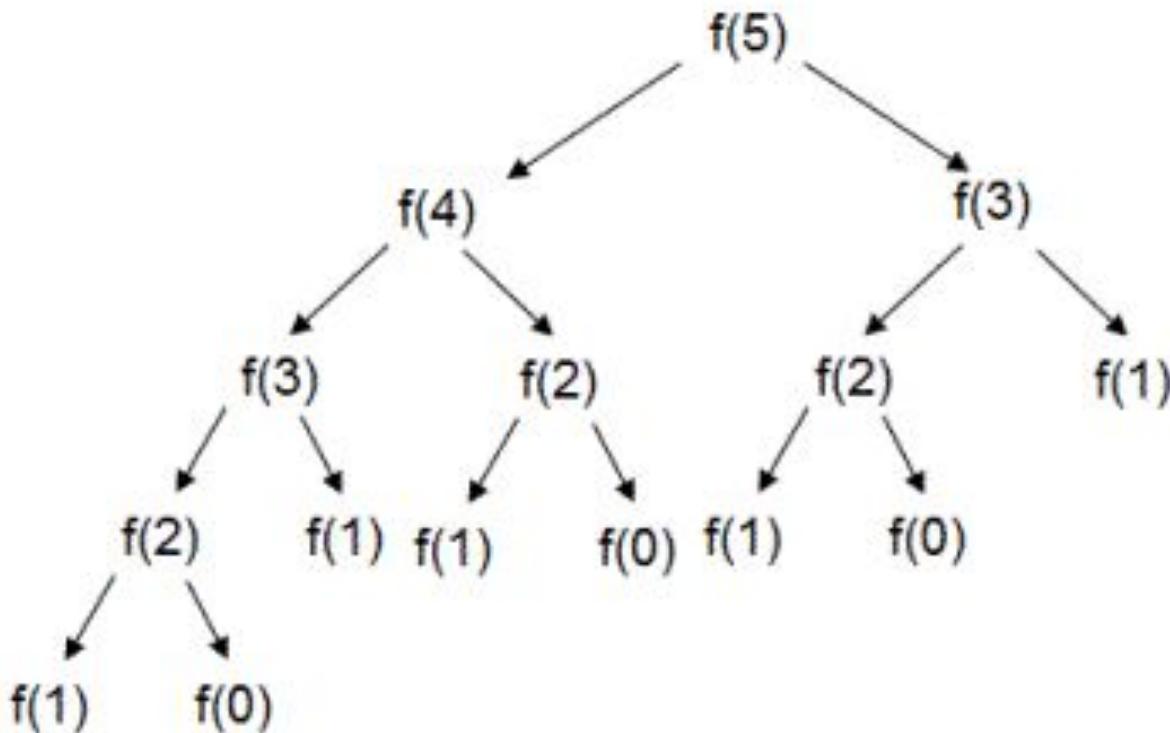


# Fibonacci

```
Int fibonaci(int i) {
 if(i == 0) {
 return 0;
 }
 if(i == 1) {
 return 1;
 }
 return fibonaci (i-1) + fibonaci (i-2);
}
```

# Tree

$f(n) : \text{fibonacci}(n)$



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET

# Recursion : Supplementary Content

Prepared By : Faias Satter  
Lecturer, University of Information Technology & Sciences

# Fibonacci Series

- also known as the Fibonacci sequence
- a mathematical sequence of numbers that starts with 0 and 1
- each subsequent number is the sum of the two preceding ones
- an infinite sequence of integers where each number (after the first two) is the sum of the two preceding ones
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

# C Program for Fibonacci Series

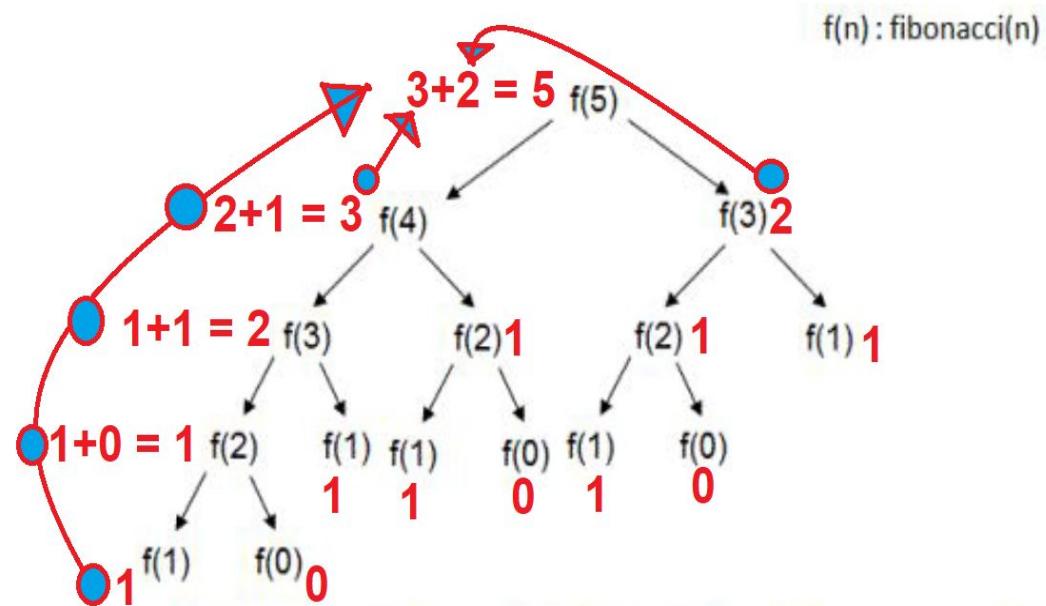
```
#include <stdio.h>

int main() {
 int n, first = 0, second = 1, next, c;
 printf("Enter the number of terms: ");
 scanf("%d", &n);
 printf("Fibonacci Series: ");
 for (c = 0; c < n; c++) {
 if (c <= 1) next = c;
 else { next = first + second; first = second; second = next; }
 printf("%d, ", next);
 }
 printf("\n");
 return 0;
}
```

# Recursive C Program for Fibonacci Series

```
#include <stdio.h>
int fibonacci(int n) {
 if (n <= 1) return n;
 else return (fibonacci(n - 1) + fibonacci(n - 2));
}
int main() {
 int n;
 printf("Enter the number of terms: ");
 scanf("%d", &n);
 printf("Fibonacci Series: ");
 for (int i = 0; i < n; i++) {
 printf("%d, ", fibonacci(i));
 }
 printf("\n"); return 0;
}
```

## Tree



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence

# Factorial

- mathematical function
- represents the product of all positive integers from 1 to a given positive integer
- denoted by an exclamation mark (!) following the number
- the factorial of a positive integer "n" is written as "n!" and is calculated as:
- $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$
- Examples :

$$5! \text{ (read as "five factorial")} = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$3! \text{ (read as "three factorial")} = 3 \times 2 \times 1 = 6$$

$$1! \text{ (read as "one factorial")} = 1$$

$$0! \text{ (by convention) is defined as 1}$$

# C Program for Factorial of a number

```
#include <stdio.h>

int main() {
 int n;
 unsigned long long factorial = 1; // WHY UNSIGNED LONG LONG ??!!
 printf("Enter a positive integer: ");
 scanf("%d", &n);
 if (n < 0) printf("Factorial is not defined for negative numbers.\n"); // Check if the input is negative
 else { // Calculate the factorial
 for (int i = 1; i <= n; i++)
 factorial *= i;
 printf("Factorial of %d = %llu\n", n, factorial); } }
```

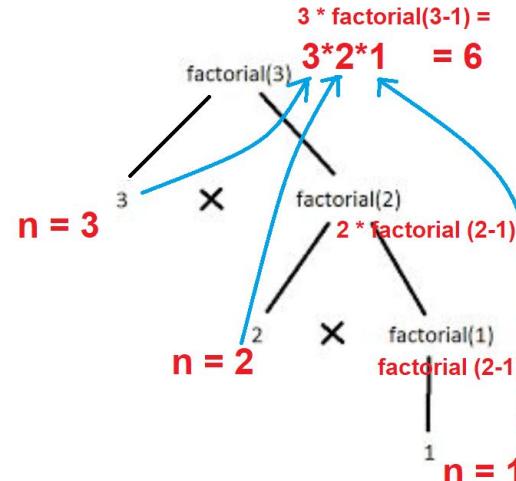
# Recursive C Program for Factorial of a number

```
#include <stdio.h>

unsigned long long factorial(int n);

int main() {
 int n; printf("Enter a positive integer: ");
 scanf("%d", &n);
 if (n < 0)
 { printf("Factorial is not defined for negative numbers.\n"); }
 else {
 unsigned long long result = factorial(n);
 printf("Factorial of %d = %llu\n", n, result);
 }
 return 0;
}
```

```
unsigned long long factorial(int n) {
 if (n == 0 || n == 1)
 { return 1; }
 else { return n * factorial(n - 1); }
}
```



# Resource Material

- What Is Recursion ?

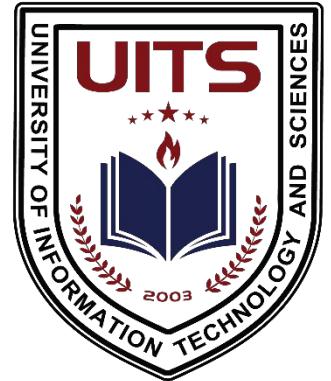
[https://sites.google.com/site/smilitude/recursion\\_and\\_dp](https://sites.google.com/site/smilitude/recursion_and_dp)

- Fibonacci Series :

<https://www.shafaetsplanet.com/?p=1022>

- Factorial :

<https://www.programiz.com/c-programming/examples/factorial-recursion>



# CSE-111

## Structured Programming Language

### ARRAY & STRING

Syeda Ajbina Nusrat  
Lecturer, CSE

# What is Array?

```
int ch[] = { 1 , 2 , 3 }
```

index  
number

0 1 2

```
int ch[30];
```

- An Array is a collection of **same data type**.
- An individual variable in the array is called an **array element**.
- The elements of an array are **referred by** a common name and are differentiate from one another by their **position within an array**.
- The elements of an array can be of any data type but all the elements in an array must be of the same type.

# Advantages of Array

- It is capable of storing many elements at a time
- It allows random accessing of elements i.e. any element of the array can be randomly accessed using indexes.

# Disadvantages of Array

- Predetermining the size of the array is a must. `int ch[ 30 ];`
- There is a chance of memory wastage or shortage.
- To delete one element in the array, we need to traverse throughout the array.

# Array declaration

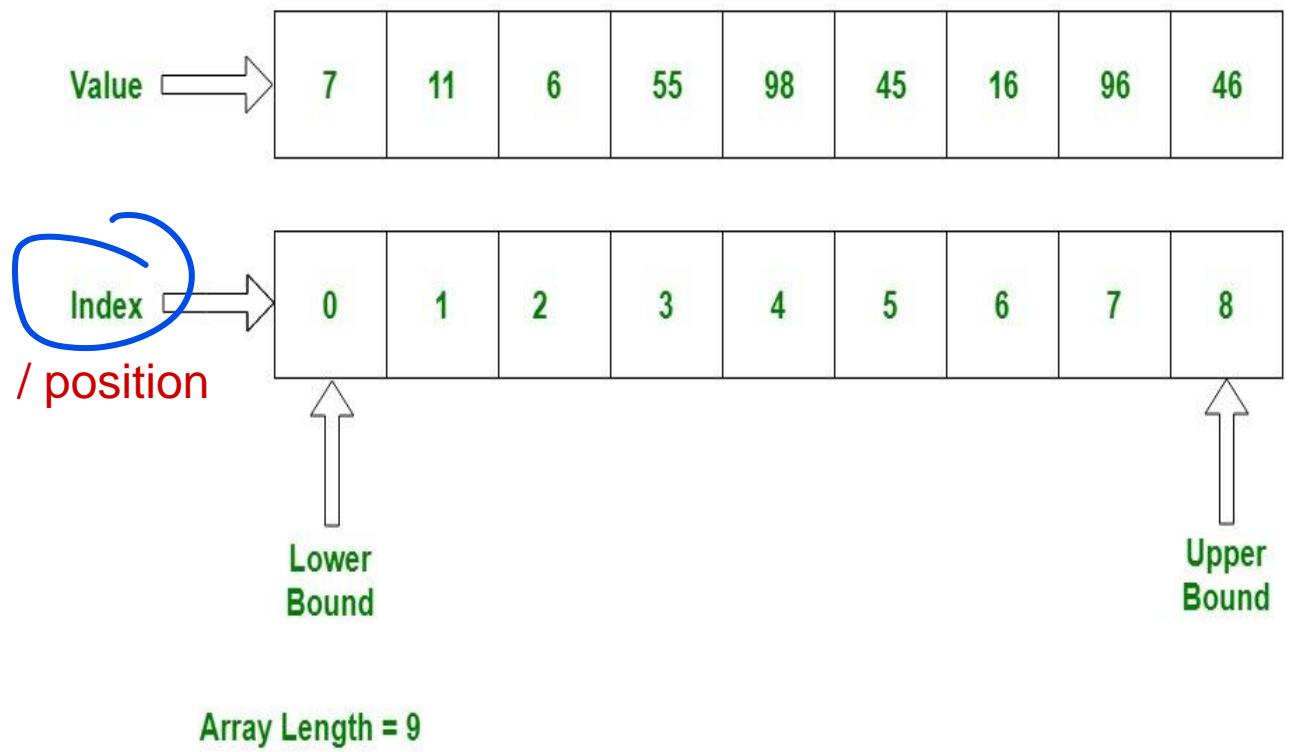
```
int ch [30] ;
Data_type var_name [array_size];
```

- array\_size specifies the number of elements in the array.
- Example:

```
int var[10];
float venus[20];
double earth[5];
char pluto[7];
```

```
int var[9];
```

- var[0] - 1st element, index 0
- var[1] - 2nd element, index 1
- var[2] - 3rd element, index 2
- var[3] - 4th element, index 3
- var[4] - 5th element, index 4
- var[5] - 6th element, index 5
- var[6] - 7th element, index 6
- var[7] - 8th element, index 7
- var[8] - 9th element, index 8



# Important things to remember in array

- Suppose, you declared the array of 10 students. For example: arr[10].
- You can use array members from arr[0] to arr[9]. But, what if you want to use element arr[10], arr[13] etc. Compiler may not show error using these elements but, may cause fatal error during program execution.

# Array Initialization (Compile time)

```
1 #include<stdio.h>
2
3 int main()
4 {
5 // Initialization type1
6 int var1[7] = {14, 22, 31, 18, 19, 7, 3};
7
8 // Initialization type2
9 int var2[] = {14, 22, 31, 18, 19, 7, 23};
10
11
12 return 0;
13 }
14 }
```

| 0  | 1  | 2  | 3  | 4  | 5 | 6  |
|----|----|----|----|----|---|----|
| 14 | 22 | 31 | 18 | 19 | 7 | 23 |

```
int var[7] = {14, 22, 31};
```

| 0  | 1  | 2  | 3 | 4 | 5 | 6 |
|----|----|----|---|---|---|---|
| 14 | 22 | 31 | 0 | 0 | 0 | 0 |

```
int var[7] = {9, 14, 0};
```

| 0 | 1  | 2 | 3 | 4 | 5 | 6 |
|---|----|---|---|---|---|---|
| 9 | 14 | 0 | 0 | 0 | 0 | 0 |

```
int var[7] = {9};
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | 0 | 0 | 0 | 0 | 0 | 0 |

```
int var[7] = {0};
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

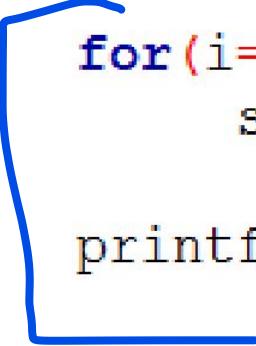
```
1 #include<stdio.h>
2 int main()
3 {
4 int var[4];
5 var[0] = 15;
6 var[1] = 13;
7 var[2] = 17;
8 var[3] = 19;
9
10 printf("%d ", var[0]);
11 printf("%d ", var[1]);
12 printf("%d ", var[2]);
13 printf("%d ", var[3]);
14
15 return 0;
16 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4];
5
6 scanf(" %d %d %d %d", &var[0], &var[1], &var[2], &var[3]);
7
8 printf("%d %d ", var[0], var[1]);
9 printf("%d %d ", var[2], var[3]);
10 }
```

---

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4];
5
6 scanf(" %d", &var[0]);
7 scanf(" %d", &var[1]);
8 scanf(" %d", &var[2]);
9 scanf(" %d", &var[3]);
10
11 printf("%d %d ", var[0], var[1]);
12 printf("%d %d ", var[2], var[3]);
13 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int i, var[4];
5
6 for(i=0 ; i<4 ; i++)
7 scanf(" %d", &var[i]);
8
9 printf("%d %d %d %d", var[0], var[1], var[2], var[3]);
10 }
```



```
1 #include<stdio.h>
2 void main()
3 {
4 int var[4], i;
5
6 for(i=0 ; i<4 ; i++)
7 scanf("%d", &var[i]);
8
9 for(i=0 ; i<4 ; i++)
10 printf("%d ", var[i]);
11 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[100], i, n;
5
6 printf("Number of Inputs: ");
7 scanf("%d", &n);
8
9 for(i=0; i<n; i++)
10 scanf("%d", &var[i]);
11
12 for(i=0; i<n; i++)
13 printf("%d ", var[i]);
14 }
```

```
1 #include<stdio.h>
2 void main()
3 {
4 int var[100], i, n;
5 float avg=0;
6
7 printf("Number of Inputs: ");
8 scanf(" %d", &n);
9
10 for(i=0; i<n; i++) //Taking input of n numbers
11 scanf(" %d", &var[i]);
12
13 for(i=0; i<n; i++) //Calculating sum of n numbers
14 avg = avg + var[i];
15
16 avg = avg / n; //Calculating average of n numbers
17 printf("Avg: %.2f\n", avg);
18 }
```

```
1 #include<stdio.h>
2 void main()
3 { int var[100], i, n, min;
4
5 printf("Number of Inputs: ");
6 scanf("%d", &n);
7 for(i=0; i<n; i++)
8 {
9 scanf("%d", &var[i]); //Taking Input of n numbers
10 }
11 min = var[0];
12 for(i=0; i<n; i++)
13 {
14 if(var[i] < min)
15 {
16 min = var[i];
17 }
18 }
19
20 printf("Minimum %d", min);
21 }
```

# Character Array

- Character arrays have several unique features
- A character array can be initialized using a *string literal*

```
char string1[] = "first";
```

- “first” string literal contains five characters plus a special string termination character called null character (\0)
- string1 array actually has 6 elements-f, i, r, s, t and \0

- Character arrays can be initialized as follows as well-

```
char string1 []= {'f', 'i', 'r', 's', 't', '\0'};
```

- We can access individual characters in a string directly using array subscript notation. For example, string1 [3] is the character 's'

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| H | E | L | L | O |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[7], i;
5
6 for(i=0 ; i<5 ; i++)
7 scanf("%c", &str[i]);
8
9 for(i=0 ; i<5 ; i++)
10 printf("%c", str[i]);
11 }
12 }
```

OUTPUT H E L L O

- String is defined as a *NULL* terminated character array.
- Format Specifier: %s

```
char str[7];
```

|     |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| str | H | e | L | L | o |   |   |

Character Array

|     |   |   |   |   |   |    |   |
|-----|---|---|---|---|---|----|---|
|     | 0 | 1 | 2 | 3 | 4 | 5  | 6 |
| str | H | e | L | L | o | \0 |   |

String or NULL terminated character array

# String input - %s

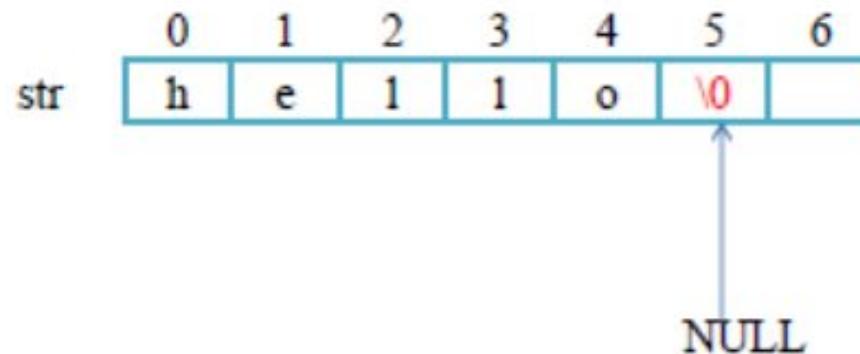
- We can also input a string directly from the keyboard using scanf () function and %s specifier.

```
char string1 [20];
scanf ("%s", string1);
```

- the name of the array is passed to `scanf()` without the preceding & used with other variables
- The & is normally used to provide `scanf()` with a variable's location in memory so a value can be stored there
- Array name is the address of the start of the array, therefore, & is not necessary
- `scanf()` reads characters from the keyboard until the first whitespace character is encountered
- `scanf()` takes upto the first whitespace

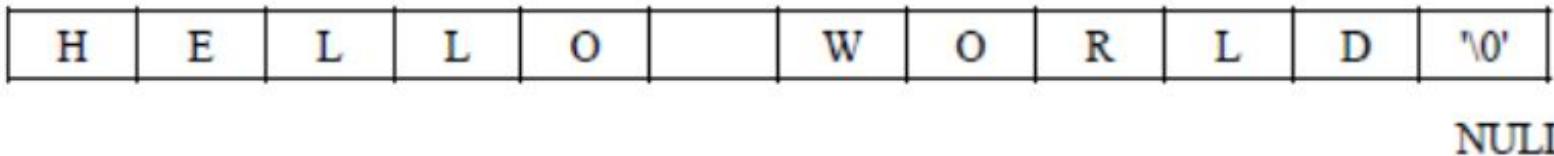
# String input - %s

```
#include<stdio.h>
void main()
{
 char str[7];
 scanf("%s", &str);
 printf("%s", str);
}
```



# Reading a line

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[7], i;
5 gets(str);
6 puts(str);
7 }
```



```
1 #include<stdio.h>
2 void main()
3 {
4 char str[20], i;
5 gets(str);
6
7 for(i=0 ;str[i] != '\0' ; i++)
8 printf("%c", str[i]);
9
10 }
```

|   |   |   |   |   |  |   |   |   |   |   |      |
|---|---|---|---|---|--|---|---|---|---|---|------|
| H | E | L | L | O |  | W | O | R | L | D | '\0' |
|---|---|---|---|---|--|---|---|---|---|---|------|

NULL

```
1 #include<stdio.h>
2 void main()
3 {
4 char str[20], i;
5 gets(str);
6
7 for(i=0 ;str[i] != NULL; i++)
8 printf("%c", str[i]);
9
10 }
```

|   |   |   |   |   |  |   |   |   |   |   |    |
|---|---|---|---|---|--|---|---|---|---|---|----|
| H | E | L | L | O |  | W | O | R | L | D | \0 |
|---|---|---|---|---|--|---|---|---|---|---|----|

NULL

# Library functions (String)

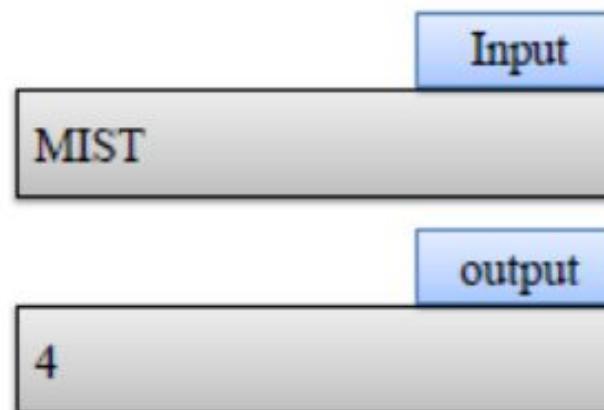
Every C compiler provides a large set of string handling library functions, which are contained in the header file “string.h”

- `strlen` : Finds the length of the string
- `strcat` : Appends one string at the end of the other
- `strcpy( to, from)`: Copies one string into another
- `strncpy` : Copies first n characters of one string into another
- `strcmp (s1, s2)`: Compares two strings
  - Returns 0 if same
  - -ve if s1 less than s2
  - +ve if s1 greater than s2
- `strchr` : Finds first occurrence of a given character in a string

# Strlen()

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

```
int main()
{
 char str[100];
 int len;
 gets(str);
 len = strlen(str);
 printf("%d", len);
}
```

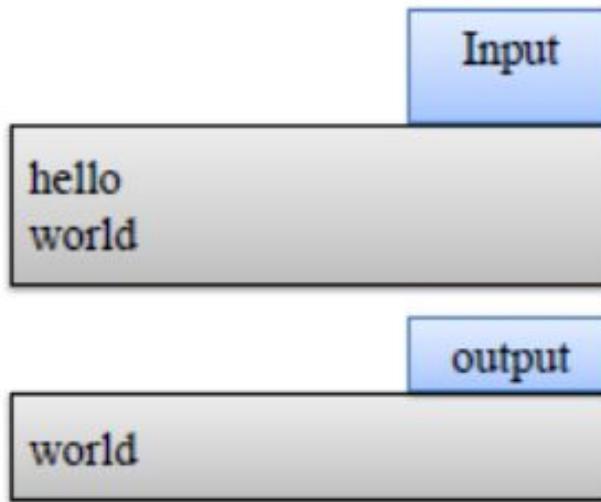


# Strcpy()

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

void main()
{
 char str1[100], str2[100];
 gets(str1);
 gets(str2);
 strcpy(str1, str2);

 puts(str1);
}
```

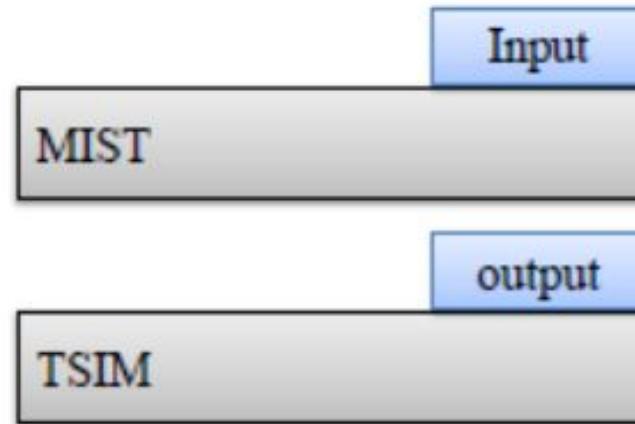


# Strrev()

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

void main()
{
 char str[100];
 gets(str);
 strrev(str);

 printf("%s", str);
}
```

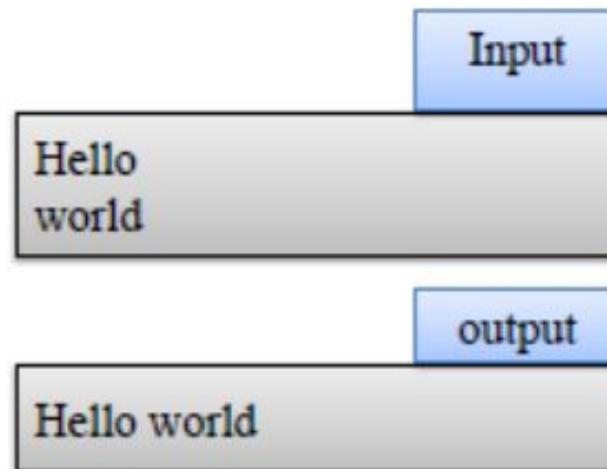


# Strcat()

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

void main()
{
 char str1[100], str2[100];
 gets(str1);
 gets(str2);
 strcat(str1, str2);

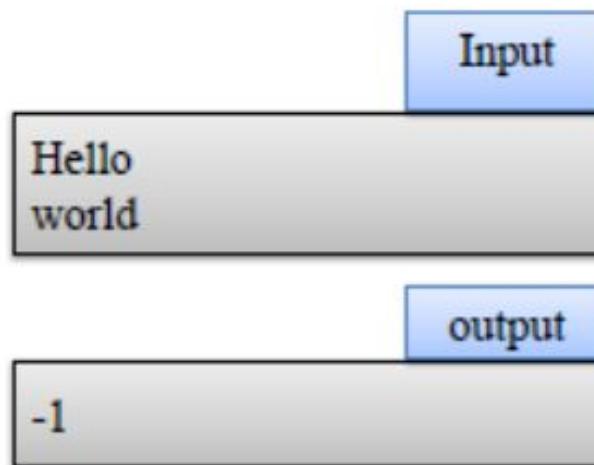
 printf("%s", str1);
}
```



# strcmp()

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

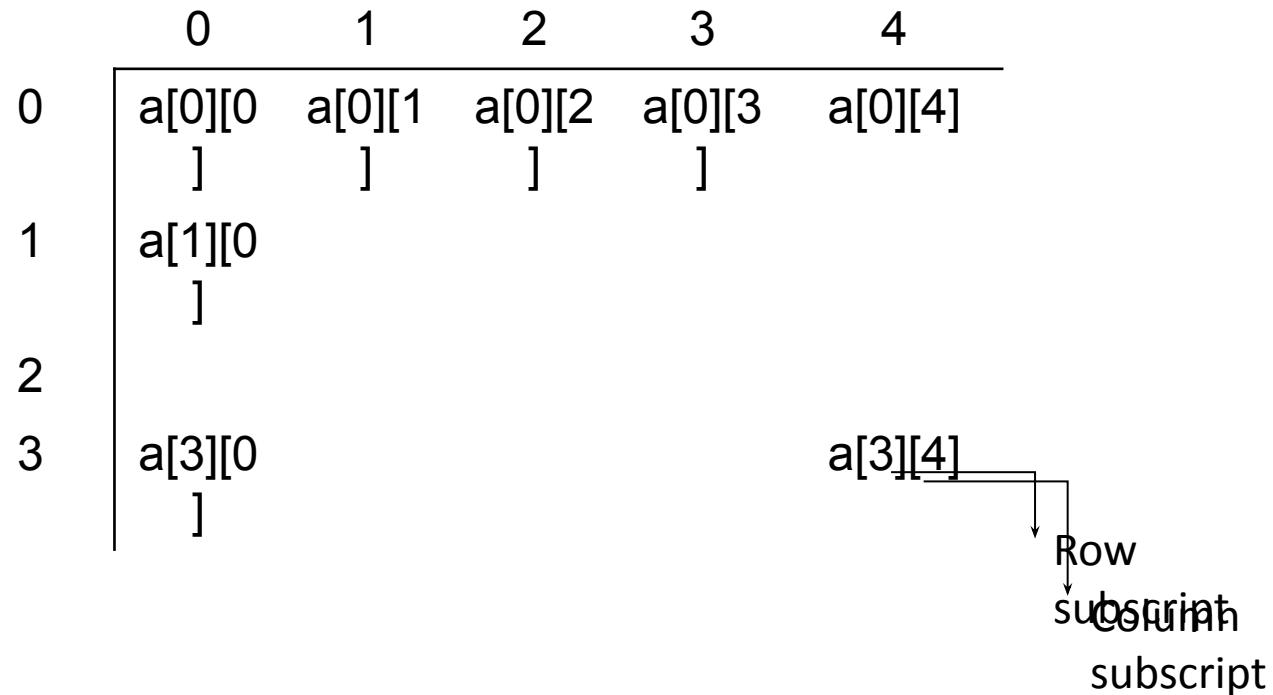
void main()
{
 char str1[100], str2[100];
 int r;
 gets(str1);
 gets(str2);
 r = strcmp(str1, str2);
 printf("%d", r);
}
```



# 2-D Array

- Array of one dimensional arrays
- Row, column format
- Accessed a row at a time from left to right

Example: float yeartemp[12][31];



# Example

```
#include<stdio.h>

int main()
{
 int td[4][5];
 int i, j;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++)
 td[i][j]=i;
 for(i=0; i<4; i++)
 {
 for(j=0; j<5; j++)
 printf("%d ", td[i][j]);
 printf("\n");
 }
 return 0;
}
```

|                |
|----------------|
| <b>Output:</b> |
| 0 0 0 0 0      |
| 1 1 1 1 1      |
| 2 2 2 2 2      |
| 3 3 3 3 3      |

# Example

```
#include<stdio.h>

int main()
{
 int td[4][5];
 int i, j;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++)
 td[i][j]=i*j;
 for(i=0; i<4; i++)
 {
 for(j=0; j<5; j++)
 printf("%d ", td[i][j]);
 printf("\n");
 }
 return 0;
}
```

**Output:**  
0 0 0 0  
0 1 2 3 4  
0 2 4 6 8  
0 3 6 9 12

# Example

- Initialization:

```
int sqr[3][3] ={
 1,2,3,
 4,5,6,
 7,8,9
};
```

|           | Col no. 0 | Col no. 1 | Col no. 2 |
|-----------|-----------|-----------|-----------|
| Row no. 0 | 1         | 2         | 3         |
| Row no. 1 | 4         | 5         | 6         |
| Row no. 2 | 7         | 8         | 9         |

- Initialization:

- Specify all but the leftmost dimension

```
int sqr[][3] ={
 1,2,3,
 4,5,6,
 7,8,9
};
```

- Initialization:

```
int sqr[3][3] ={
 {1,2,3},
 {4,5,6},
 {7,8,9}
};
```

```
int sqr[3][3] ={1,2,3,4,5,6,7,8,9};
int sqr[][][3] ={1,2,3,4,5,6,7,8,9};
```

- Initialization:

```
int sqr[3][] = {1,2,3,4,5,6,7,8,9};
```

```
int sqr[][] = {1,2,3,4,5,6,7,8,9};
```

- This would never work

# Reference

- TEACH YOURSELF C byHerbert Schildt
- Programming in ANSI C – E Balagurusamy
- Sanjida Nasreen Tumpa, MIST
- Rushdi Shams
- Md. Kowsar Hossain,Assistant Professor,KUET