SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 1 – Part 1

Algorithms and Flowcharts

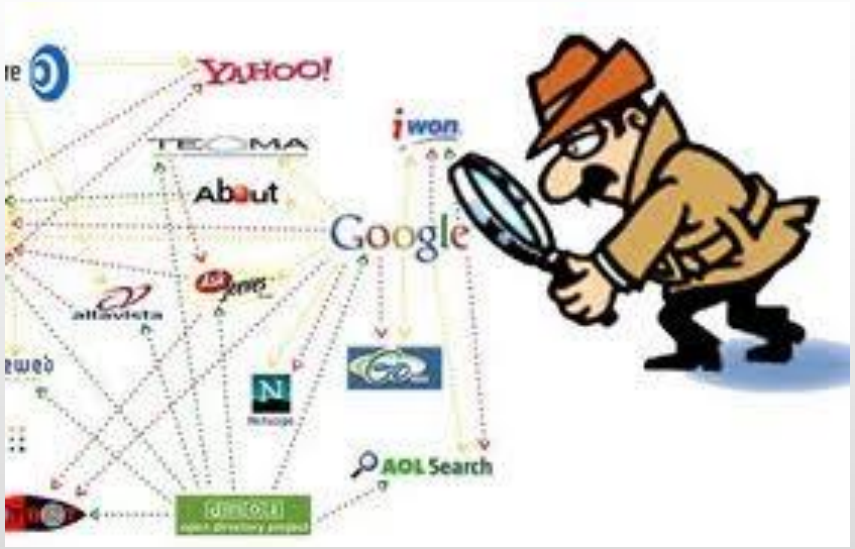
SLIIT
FACULTY OF COMPUTING

What is a computer ?

- A computer is a machine for processing data to produce information.

What can it do ?

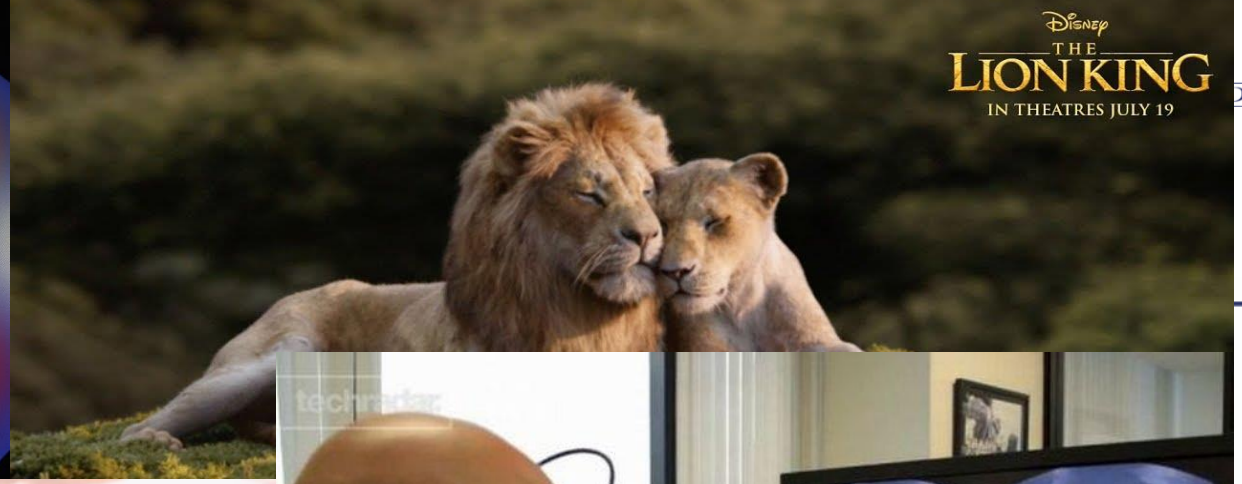
- Store large amounts of data.
- Process large amount of data quickly.
- Fast access to information and records
- Increase productivity

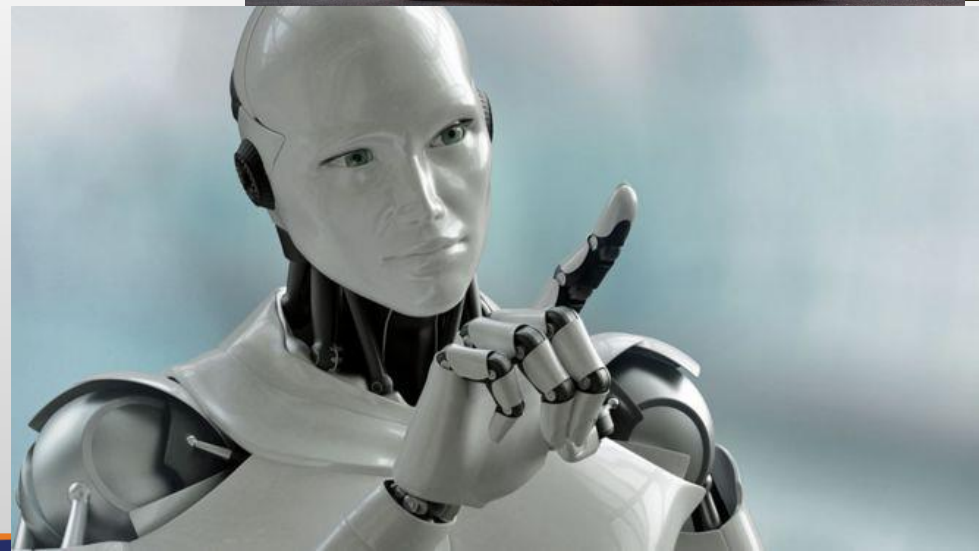


What can it do ?

- What else……
  - Calculate
  - Process (word, numbers, pictures, sound)
  - Store and retrieve
  - Match / compare / sort
  - Merge
  - And some more…..


Social Media
Snapchat
Facebook
99
Instagram


Disney
THE
LION KING
IN THEATRES JULY 19



How do computers do all these

How do you instruct a computer to do tasks?

- Write a program???
- Program
  - is a set of step-by-step instructions

  written to perform a specific task…
  - Program consist of a set of instructions

  that are executed by the computer

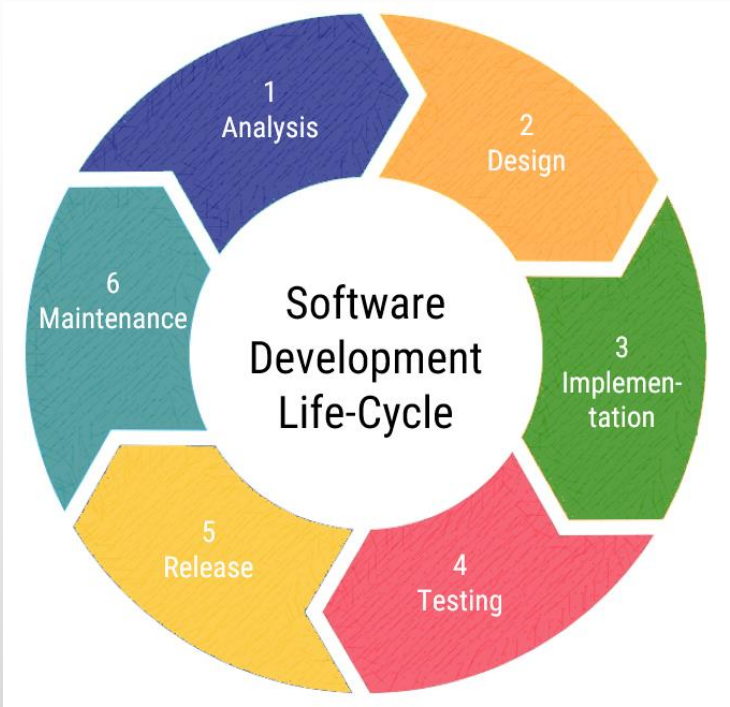
How do computers do all these…?

- This process of writing programs is called ***programming***
- These programs are written by a ***programmer***
- A set of rules that tell a computer what operations to perform are called ***programming language***
- In other words, we can call this program ***Software***

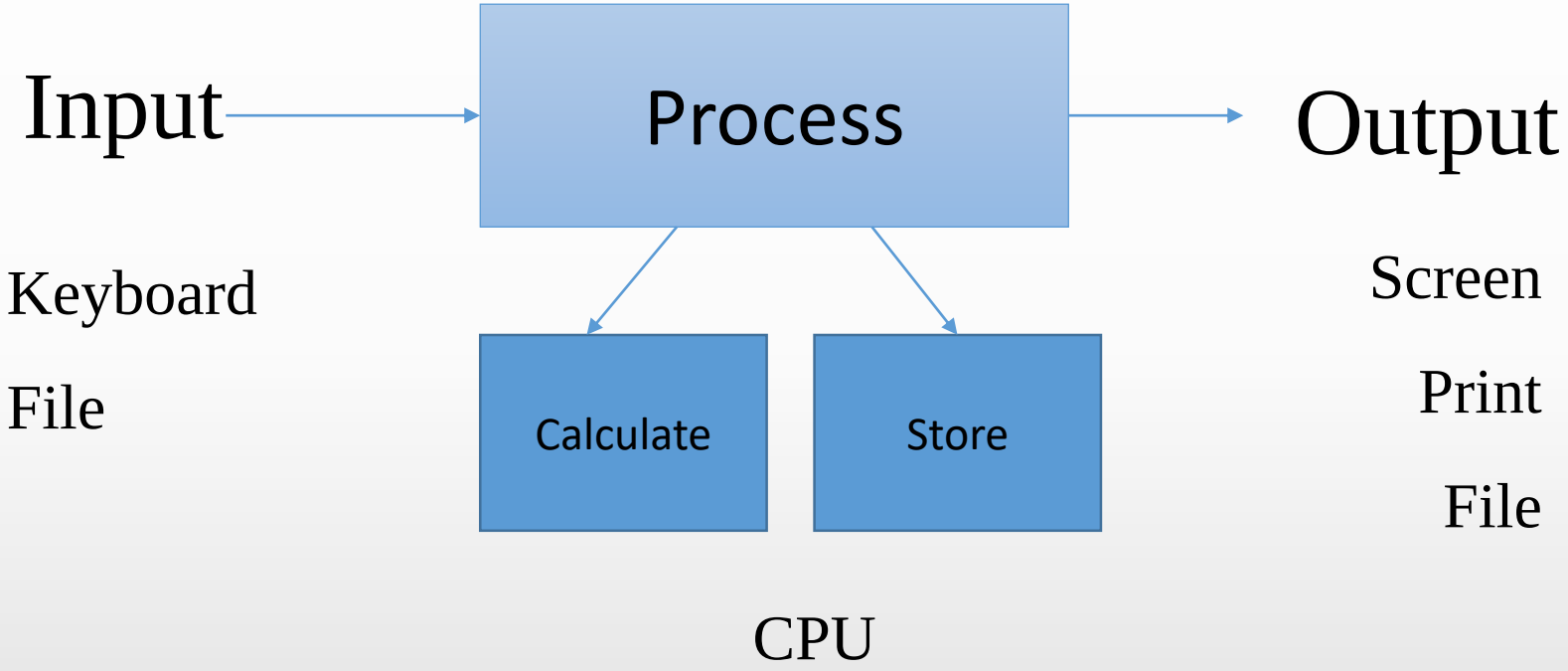
How do you write programs

In developing a computer program the programmer must:

- Analysis - Understand/Define the problem
- Design – plan the solution
- Implement - Write/Code the program
- Test - Compile, Debug & Test
- Release - Implement the program
- Maintain - Document the program



What can the computer do ?



Analyze / Understand the problem

Problem : Given 2 numbers (say a and b) where a < (is less than) b , I want to find the sum of numbers between a and b

- Input : Integers a & b (where a<b)
- Task : Find the sum of numbers between a & b
- Output : The sum

Planning the solution

- It is important to spend considerable time in planning your solution (program) in order to ensure it is properly structured
- If properly planned and presented, the time and effort in 'coding' the solution, will be minimal
- Use algorithms to prepare a solution

Algorithms

- An algorithm is a complete step-by-step set of instruction of how to solve a problem.
- Consist of
  - The instructions
  - The order of the instructions

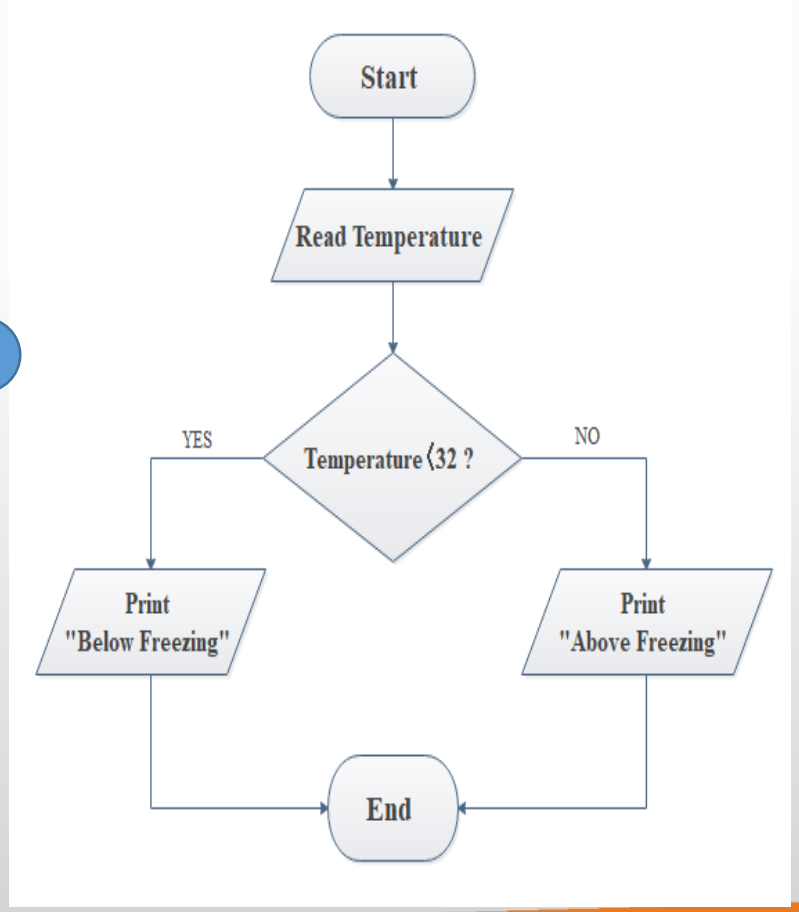
How to make a cup of Tea ?

- Fill electric tea kettle
- Bring it to a boil
- Pour hot water in cup
- Put teabag in cup
- Steep for 4 minutes
- Remove teabag

Algorithms

- Algorithms is Programming are represented by
  - Flowcharts
  - Pseudo codes

```
Start
    Input Temperature
    If Temperature < 32
      Then Print "Below freezing"
    Else
      Print " Above Freezing
End
```



Flow charts

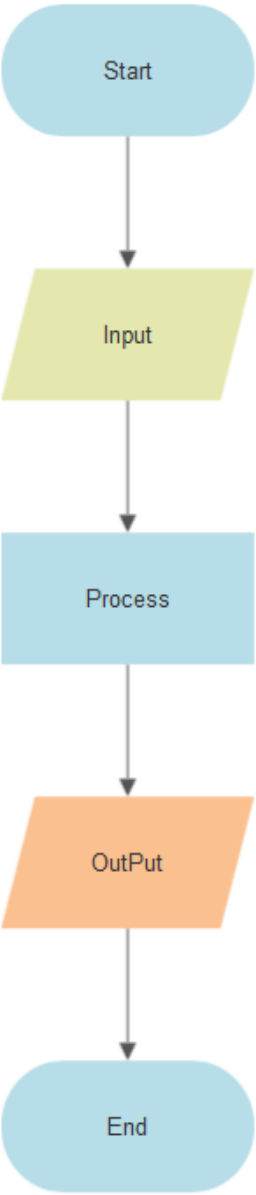
- It is a step by step diagrammatic representation of the program
- Each instruction is represented by a Symbol.
- Arrow shows the order in which the instructions are executed.

Flow chart symbols

Symbol	Name	Function
	Start/end	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/Output	A parallelogram represents input or ouptut.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

Flow charts

- Most simple algorithms that you will develop will involve inputting some data, performing some calculation and finally displaying the output.



The Logical Constructs / Control structures

- Three basic constructs that controls the flow of an algorithm;
  - Sequence
  - Selection
  - Iteration

Sequence

- **SEQUENCE** is a linear progression where one task is performed sequentially after another…
- solution steps must follow each other in a logical sequence
- Statements are executed in the same order as they are written.

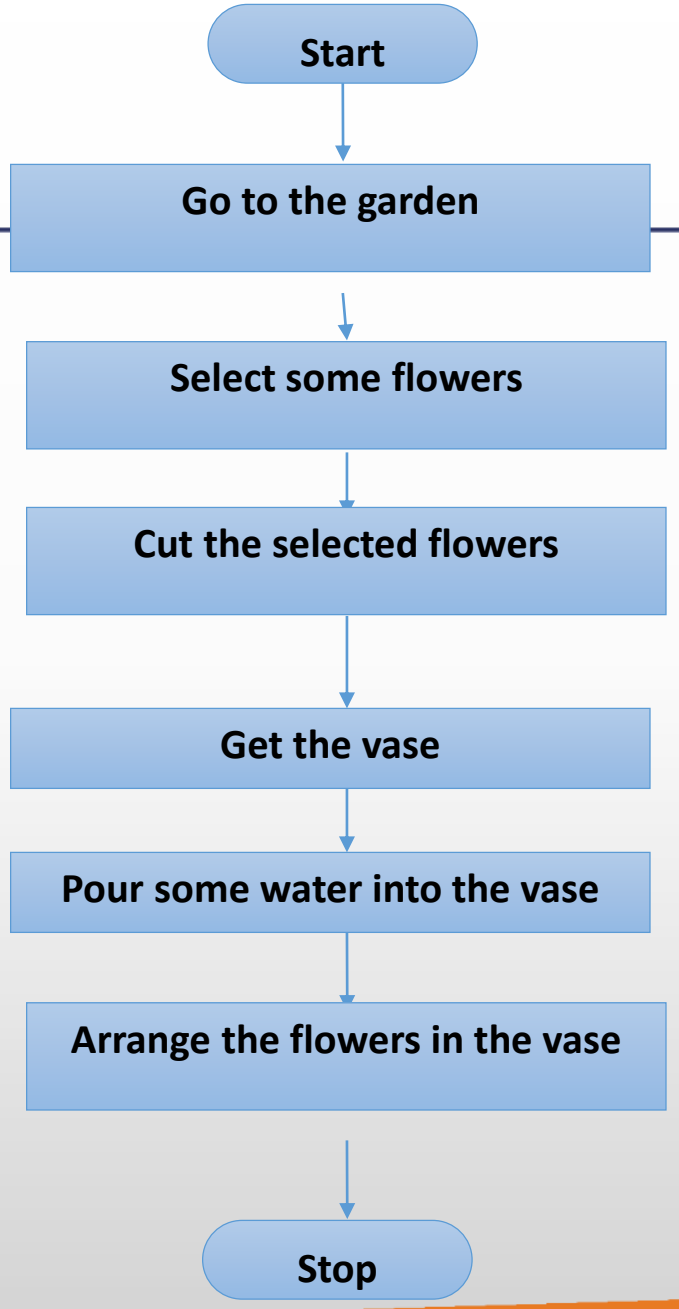
Example 01

- Put the following steps in correct order and draw a flowchart to show

“ How to arrange a flower vase”.

1. Cut the selected flower
2. Go to the garden
3. Arrange the flowers in the vase
4. Get the vase
5. Select some flowers
6. Pour some water into the vase

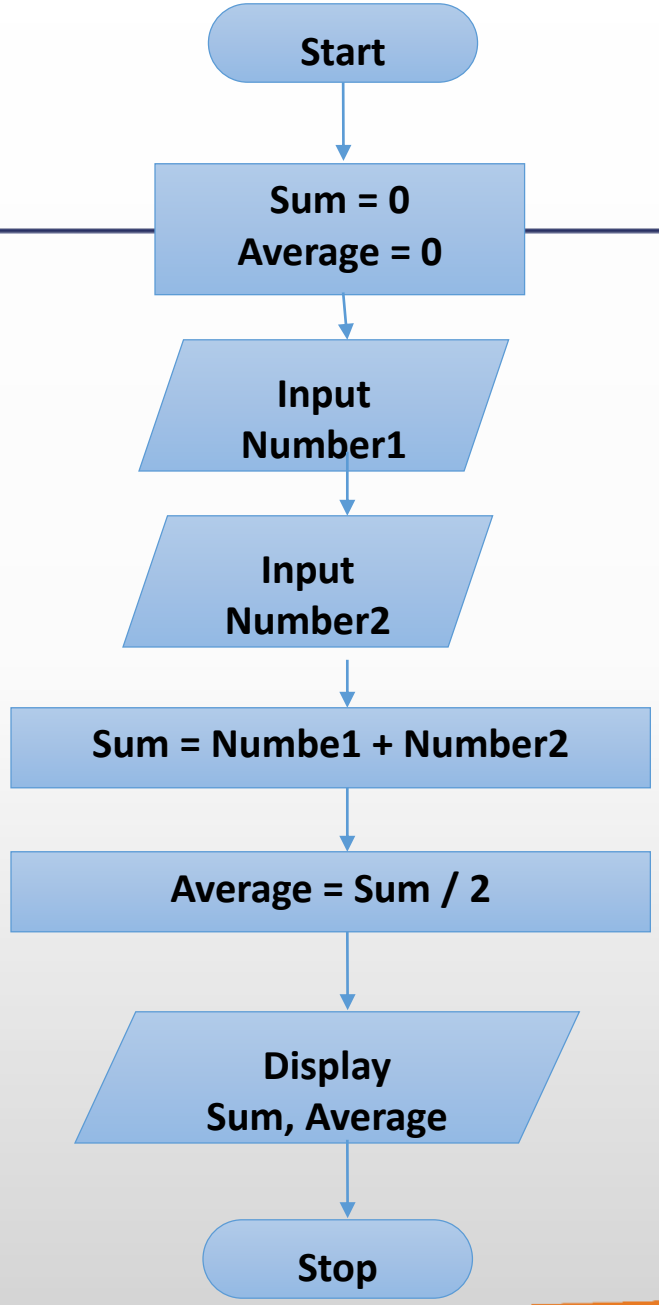
Example 01 - Solution



Example 02

- Draw a flowchart to input two numbers and find the sum and average.

Example 02 - Solution



Exercise

- Draw a flowchart to input the radius of a circle and calculate and display the area.

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 1 – Part 2

Introduction to C programming

Objectives

- At the end of the Lecture students should be able to
  - Describe the evolution of programming languages
  - Write a simple C program

Computer Program

Computer Program

Set of instructions given to the computer.

Programmers write programs in various programming languages.

Programming languages can be mainly divided into :

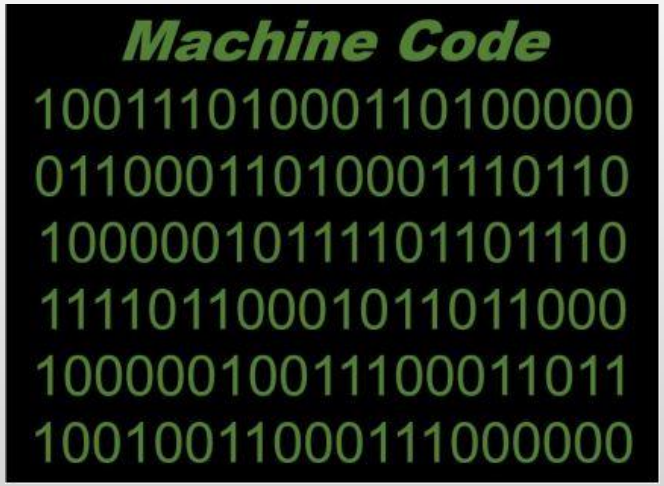
1. Low-level programming languages
   - Machine Language
   - Assembly Language
2. High-level programming languages

Programming Languages

Low-level programming Languages

Machine Languages

- Consist of 1 s and 0 s
- Machine dependent
- Computer can directly understand its own machine language
- Tedious and error prone for programmers



Programming Languages

Assembly Languages

- English-like abbreviations called mnemonics formed the basis
- Clearer to humans but incomprehensible to computers
- Need to translate to machine language using translator programs called assemblers

Example:

```
load  salary
add   bonus
store total
```

Programming Languages

High Level Programming Languages

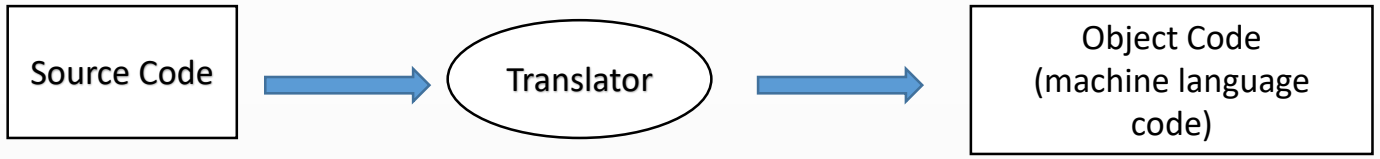
- Instructions look almost like English and mathematical notations
- Substantial tasks can be accomplished from a single statement
- Easy for humans to understand
- Translator programs convert high-level programming languages into machine language

Example:

```
total = salary + bonus;
```

- C, C++, Python, Visual Basic and Java are some of the high level programming languages.

Program Code Translation



Translator

- Assemblers (convert assembly language programs to machine language)
- Compilers (convert high-level language programs to machine language)
- Interpreters (execute high-level language programs line by line)

History of C

- C language was evolved from two previous languages, BCPL and B by Dennis Ritchie at Bell Laboratories in 1972.
- C initially became widely known as the developing language of the UNIX operating system.
- C99 and C11 are revised standards for C programming language that refines and expands the capabilities of C.

Simple C Program

Source Code

```
/* First program in C
This program displays a message */
#include <stdio.h>

// function main begins program
execution
int main(void)
{
        printf("welcome to C!");

        return 0;
} // end of function main
```

Output

```
welcome to C!
```

Simple C Program Description

```
/* First program in C
   This program displays a message */
```

```
// function main begins program execution
```

These are comments. Comments are used to document programs and it improves the program readability. C compiler ignores comments. Line comments begin with // and continue for the rest of the line. /* … */ multi-line comments can also be used. Everything from /* to */ is a comment.

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

This is a directive to C preprocessor. Lines begin with # are processed by the preprocessor before compiling the program. Here, preprocessor includes the content of stdio.h in the program. stdio.h is a header file which contains information about standard input/output library function calls such as *printf*.

Simple C Program Description

```
int main(void)
```

Every C program has this *main* function and it begins executing at main. “int” to left of main indicates that the function returns an integer. “void” in parentheses indicates that *main* does not receive any information

```
{
```

Left brace , { , begins the body of every function. Functions ends with a corresponding right brace. The portion of the program between the braces is called a block.

```
printf( “welcome to C! ” );
```

Entire line is called a statement. It instructs the computer to perform an action. A statement must end with a semicolon. This statement prints a string of characters marked by the quotation marks on the screen.

Simple C Program Description

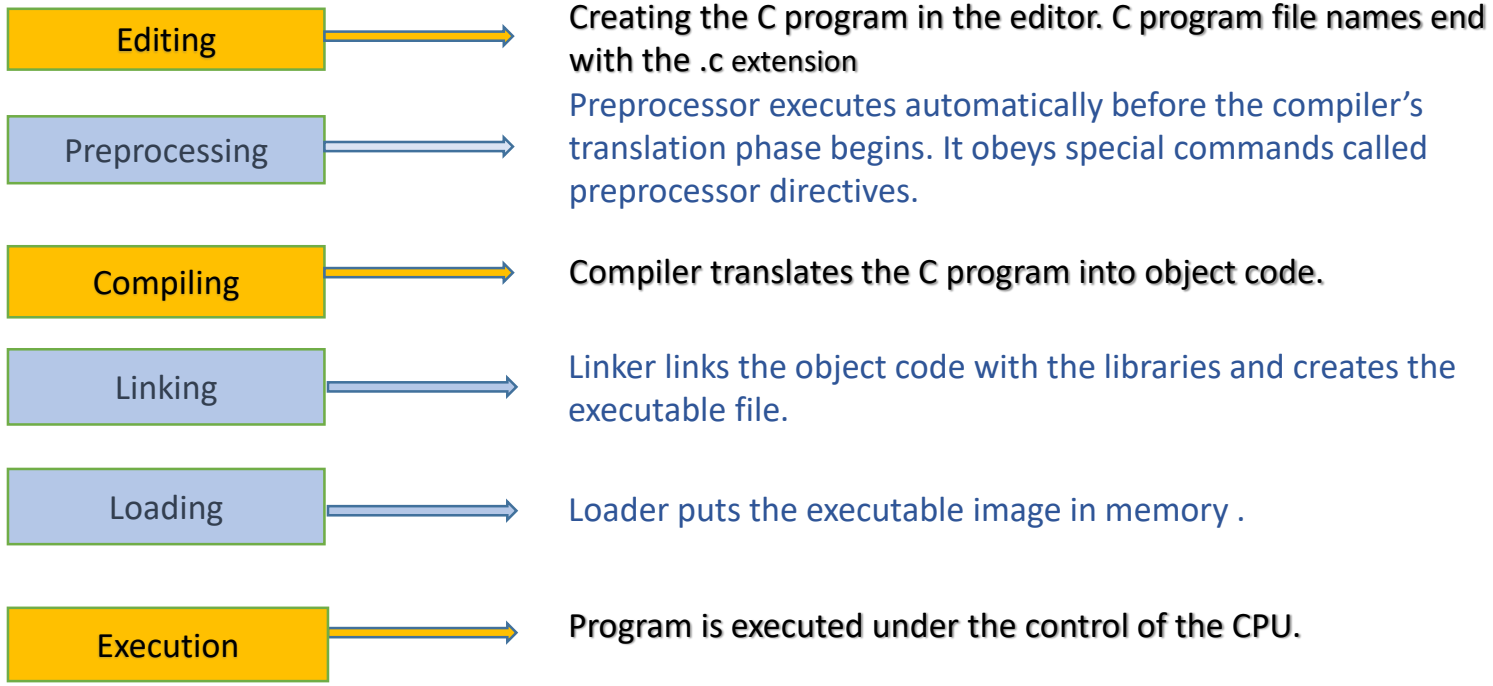
```
return 0;
```

Included at the end of every main function. It is used to exit the main function and the value 0 indicates a successful termination.

Blank Lines and White Space

Blank lines, space characters and tab characters are known as white space. White-space characters are normally ignored by the compiler.

C Program Development Environment



More Examples

```
// A text break into two printf statements

#include <stdio.h>

/* function main begins program execution */
int main(void)
{
         printf( "welcome ");
         printf( "to C!\n");

         return 0;
} // end of main function
```

```
// A text printed in multiple lines using single
printf

#include <stdio.h>

/* function main begins program execution */
int main(void)
{
         printf("welcome \nto C! \n");

         return 0;
} // end of main function
```

- The backslash (\) is called escape character.
- The escape sequence \n means newline.

printf() function

printf(“welcome to C!\n”);

can be written as,

```
puts( “welcome to C!” );
```

puts function adds newline automatically.

printf(“welcome “);

can be written as,

```
printf ( “%s”, “welcome ” );
```

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 2 – Data Types and C Formatted Input/Output

SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - Use fundamental data types.
  - Write simple output statements to display values stored in variables.
  - Write simple input statements to read values from the key board.
  - Define and use derived data types.

Variables

- Variable is a location in memory where a value can be stored for use by a program
- Variables must be declared, before they can be given a value. When declaring a variable, its **name** and **data type** should be specified. Every variable has a name, type and a value.
- The declaration allocates the storage location of the appropriate size and associates the name and data type with that location.

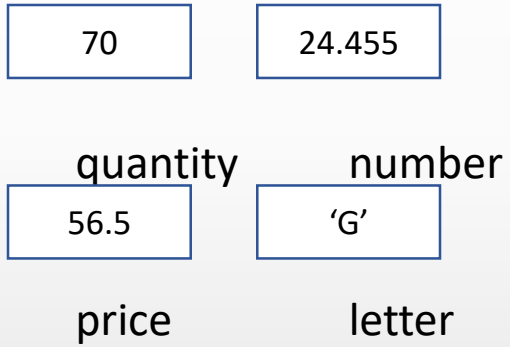
Variable Declaration

- The format for the declaration of a variable

  <data type> < Name of the variable>;

Example:

```
int quantity;
float price;
double number;
char letter;
```



Variable Names

- Variable name in C is a valid identifier.

  An identifier

  - can be a series of characters consisting of letters, digits and underscores (_)
  - does not begin with a digit
  - may not contain embedded blank spaces
  - may not be a reserved word (ex: int, return, if, while, for….)

  C is case sensitive (uppercase and lowercase letters are different in C)

  i.e. total , Total and TOTAL are three different variable names

Quiz

Which of the following can be considered as valid variables?

- *name*
- *Number Of Values*
- *Tax_Rate*
- *DistanceInFeet*
- *2BeOrNot2Be*
- *Number3*
- *Tax%*
- *for*

Data Types

- Integers
  - short
  - int
  - long int

An integer is a whole number without a decimal point or a fractional part.

There is a maximum and a minimum integer that can be stored in the computer

- Real Numbers
  - float
  - double
  - long double

Numbers which contain a decimal point and a fractional part are called floating point or real numbers

- Characters - char

Data types - Examples

- Integers

462 –39 31285

- Real Numbers

-21.73 15.0 6.252e-3

- Characters

'A' '@' '7' 'v' '.'

Exponential Notation

Exponential notation represents a floating point number as a decimal fraction times a power of 10

example,
1.645e2 is 1.645×10^2 or 164.5

Storing values into variables

- The assignment operation can be used to store a value in a variable or to change the value stored in a variable
- The assignment operator is the equal sign =
- An assignment expression has the form

variable(lvalue) = expression(rvalue)

- It stores the value of the expression (rvalue) into the memory location for the variable (lvalue)

Example:

```
quantity = 50;
number = 100.5;
amount = quantity * 55.25;
```

Quiz

Q1

```
int qty;
qty = 5 + 1;
```

What is the value of qty?

Q2

```
int count = 5;
count = count + 1;
```

What is the value of count?

C Formatted Input and Output

- All input and output is performed with streams(sequence of bytes)

  Input – bytes flow from a device (e.g. keyboard, disk drive) to main memory

  Output – bytes flow from main memory to a device (e.g. screen, disk drive)

- Normally standard input stream is connected to the keyboard and the standard output stream is connected to the screen

Formatting output with printf

- printf function output data to the standard output stream.
- printf call contains a *format control string* that describes the output format.

printf(*format-control-string, other-arguments*);

format-control-string describes the output format.

other-arguments correspond to each **conversion specification** in format-control-string.

Example:

```
printf( “%d”,  455);
```

printf Conversion Specification

Type	Conversion Specification	Description
Integer	%d	Display as a signed decimal integer
	%i	Display as a signed decimal integer (d and i are same in printf)
Floating-Point	%f	Display floating-point values in fixed-point notation (float or double data type)
Character	%c	Display a character
String	%s	Display a string

Example 01 – How to use different conversion specifier in printf

```
/* using conversion specifiers in c a
program*/
#include <stdio.h>

int main(void)
{
          printf( "%d\n ",   455);
          printf( "%d\n ", -455);
          printf( "%i\n ",   455);
          printf( "%f\n ", 1234567.89);
          printf( "%.2f\n ",   3.446);
          printf( "%c \n", 'A' );
          return 0;

} // end of main function
```

Output

```
455
-455
455
1234567.890000
3.45
A
```

conversion specifier %.2f specifies that a floating point value will be displayed with two digits to the right of the decimal point.

If %f is used without specifying the precision, the default precision of 6 is used.

When floating values are displayed with precision, the value is rounded to the indicated number of decimal positions for display purposes.

Example 02 – How to display the output of a simple calculation

```
/* adding two numbers and display output*/
#include <stdio.h>
int main(void){
        int no1, no2;
        int sum;
        no1 = 25;    //  assign value to no1 variable

        no2 = 12;    //  assign value to no2 variable

        sum = no1 + no2;  // add numbers
        printf( "Sum is %d\n",  sum);     // print sum

        return 0;
} // end of main function
```

output

```
Sum is 37
```

Reading Formatted Input with scanf

- scanf function reads from the standard input stream
- scanf contains a format control string that indicates the type of data that should be entered.

scanf(format-control-string, other-arguments);

format-control-string describes the input format.

other-arguments are pointers to variables in which the input will be stored.

Example:

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

scanf Conversion Specification

Conversion Specification	Description
%d	Read signed decimal integer. Argument is a pointer to an int
%i	Read a signed decimal, octal or hexadecimal integer. Argument is a pointer to an int
%f	Reading a floating point value. Argument is a pointer to a float
%lf	Reading a floating point value. Argument is a pointer to a double
%c	Read a character. Argument is a pointer to a char
%s	Read a string. Argument is a pointer to an array of type char

Example 03 – Input two numbers from the keyboard and display the sum

```
/* input two number from the keyboard and add two
numbers*/
#include <stdio.h>
int main(void){
        int no1, no2;
        int sum;
        printf("Enter first number:");    /* prompt */
        scanf("%d", &no1);  /* read the value */
        printf("Enter second number:");     /* prompt*/
        scanf("%d", &no2);  /* read the value */
        sum =  no1  + no2;     /* assign total to sum */
        printf("Sum is %d\n",  sum);     /* print sum */
        return 0;
} // end of main function
```

output

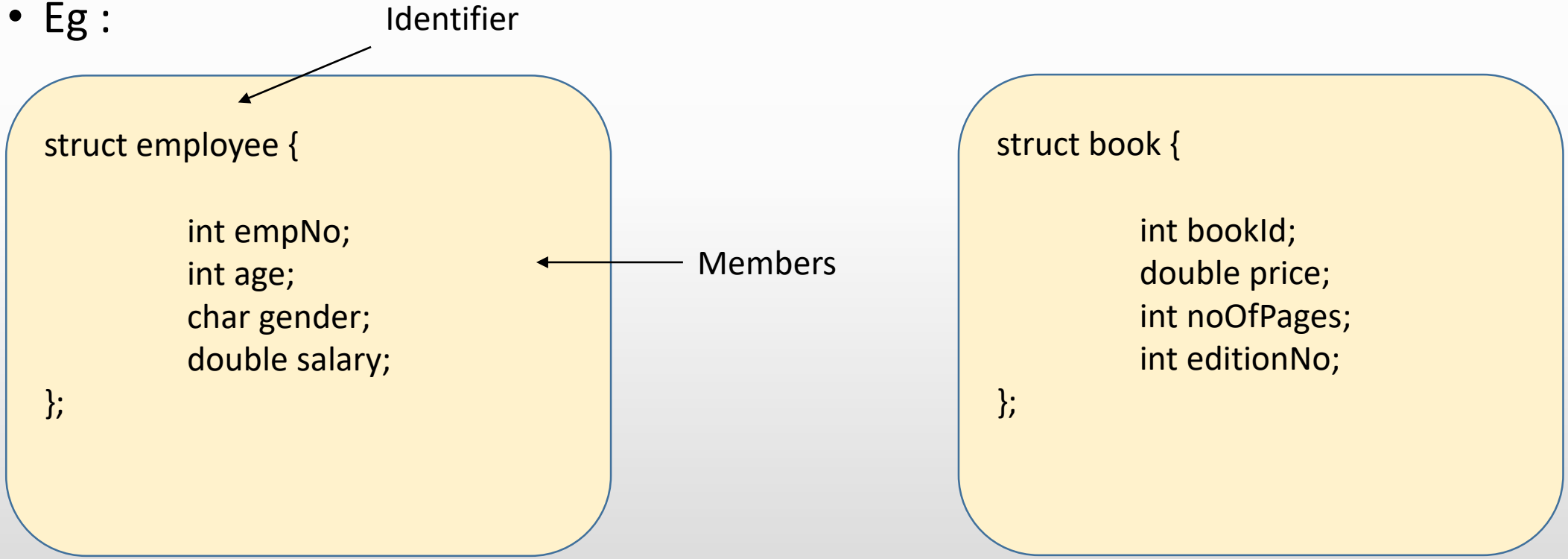
```
Enter first number: 54
Enter second number: 40
Sum is 94
```

C Structures

- Structures are derived data types.
- They are constructed using objects of other data types.
- Simply, a structure is a collection of related variables under one name
- May contain variables of different types.

Structure Definition

- Keyword **struct** is used to declare a structure
- Eg :



Declaring variables of structure type

```
struct employee {

        int empNo;
        int age;
        char gender;
        double salary;
} emp1, emp2 ;
```

emp1 and emp2 are two variables of the structure employee.

Accessing members of a structure

- // Input empNo for emp1

```
scanf("%d", &emp1.empNo);
```

- //print the salary of emp1

```
printf("%.2f", emp1.salary);
```

- //assign the gender for emp1

```
emp1.gender = 'M';
```

Example 04 – How to define and use a structure in C

```
#include <stdio.h>
struct book {
    int bookId;
    double price;
    int noOfPages;
    int editionNo;
} book1;
int main(){
    book1.bookId = 6495407;
    book1.price = 350.00;
    book1.noOfPages = 200;
    book1.editionNo = 8;
    printf("Book 1 book ID : %d\n", book1. bookId);
    printf("Book 1 price : %.2f\n", book1.price);
    printf("Book 1 no Of Pages : %d\n", book1.noOfPages);
    printf("Book 1 edition No : %d\n", book1.editionNo);
    return 0;
}
```

OUTPUT

Book 1 book ID : 6495407
Book 1 price : 350.00
Book 1 no Of Pages : 200
Book 1 edition No : 8

Summary

- Variables
- Printf statement
- Scanf statement
- Conversion specifier
- Strutures

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 3 – Operators in C

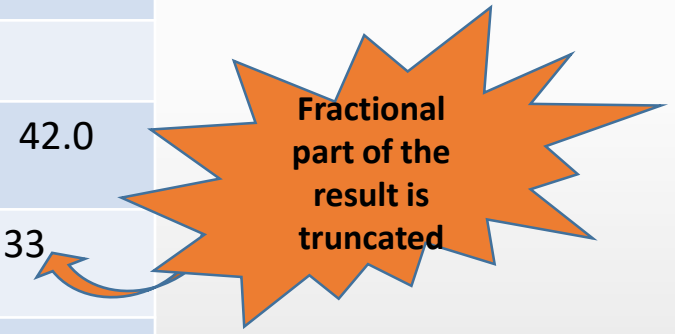
SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - Use arithmetic operators in C programs.
  - Correctly apply the precedence of arithmetic operators
  - Use relational operators in C programs.

Arithmetic Operators

Operation	Arithmetic Operator	C Expression	Example
Addition	+	no1 + 8	5 + 6 = 11
Subtraction	-	value – no2	7 - 2 = 5
Multiplication	*	qty * price	4 * 10.5 = 42.0
Division	/	tot / 3	100 / 3 = 33
Remainder	%	no1 % no2	10 % 3 = 1



Arithmetic operators are binary operators.

Operator Precedence and Associativity

- Operator precedence establishes the priority of an operator in relationship to all other operators.
- Parentheses can be used to modify the normal order of execution of an expression.
- Operator associativity establishes the order in which operators of the same precedence are to be executed.

Operator Precedence of Arithmetic Operators

Order	Operator(s)	Associativity
1	() Parentheses	Left to right
2	* Multiplication / Division % Remainder	Left to right
3	+ Addition - Subtraction	Left to right

Example

74 / 10 % 2 * 5 – 10 % (5 – 1)

74 / 10 % 2 * 5 – 10 % 4 → parentheses

7 % 2 * 5 – 10 % 4

1 * 5 – 10 % 4 } multiplication, division, remainder according to associativity

5 - 10 % 4

5 - 2

3 → subtraction

Quiz

- Find the result of the following expressions

Q1

```
y = 2 * 5 * 5 + 3 * 5 + 7;
```

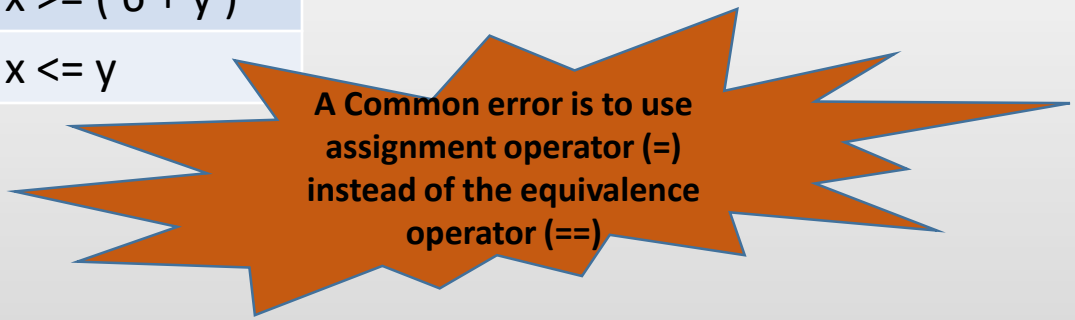
Q2

```
y = -75 / 25 + 5 * 3 + 2 / 3
```

Equality and Relational Operators

Equality and relational operators test the relationship between two expressions and yields true or false

Operation	Operator	C Expression
equal to	==	x == y
not equal to	!=	x != y
greater than	>	x > 30
less than	<	x < y
greater than or equal to	>=	x >= (6 + y)
less than or equal to	<=	x <= y



Operator Precedence Revisited

Order	Operator(s)	Associativity
1	()	Left to right
2	!	Right to left
3	* / %	Left to right
4	+ -	Left to right
5	< <= > >=	Left to right
6	== !=	Left to right
7	=	Right to left

Quiz

Assume no1 = 5 and no2 = 4. Determine whether the following expressions yield a *true* or *false*

Q1
no1 == -5

Q2
no1 != no2

Q3
no1 > (no2 + 1)

Q4
no1 >= (no2 + 1)

Q5
no1 <= 12

Q7
no1 == no2

Logical Operators

Used to form more complex conditions by combining simple conditions.

Operation	Operator	C Expression
Logical AND	&&	gender = 1 && age >= 65
Logical OR	||	semesterAverage >= 90 || finalExam >= 90
Logical NOT	!	! (grade == 'F')

Cast operator

- Cast operators force the conversation of a value to a specified type. It is called **explicit** conversion.
- It is formed by placing parentheses around a data type name.

Format:

(type) expression

Example

```
int total = 203;
int count= 5;
float average;

average = ( float ) total / count;
```

Converting between types implicitly

- Arithmetic expressions can be evaluated only in which the operands' data types are identical. To ensure this, the compiler performs an operation called implicit conversion on selected operands.

Example: In an expression containing the data types int and float, copies of *int* operands are made and promoted to *float.*

Assignment Operators

- There are several assignment operators for abbreviating assignment expressions.

variable = *variable* operator *expression*;

can be written as

variable operator= *expression;*

where operator is one of the binary operators +, -, *, / or %

Example

```
c  =  c  +  3;   // this is same as c += 3;
```

Increment and Decrement operators

++ increment operator

-- decrement operator

```
k = ++n;    // prefix increment  :   n = n + 1;  then  k = n;

k = n++;    // postfix increment  :   k = n;   then n = n + 1;

k = --n;    // prefix decrement  :   n = n - 1;  then  k = n;

k = n--;    // postfix decrement  :   k = n;   then n = n - 1;
```

Quiz

Find the result of following C statements if no = 10 and x = 5.

Q1

```
no  -= 4;
printf("%d", no);
```

Q2

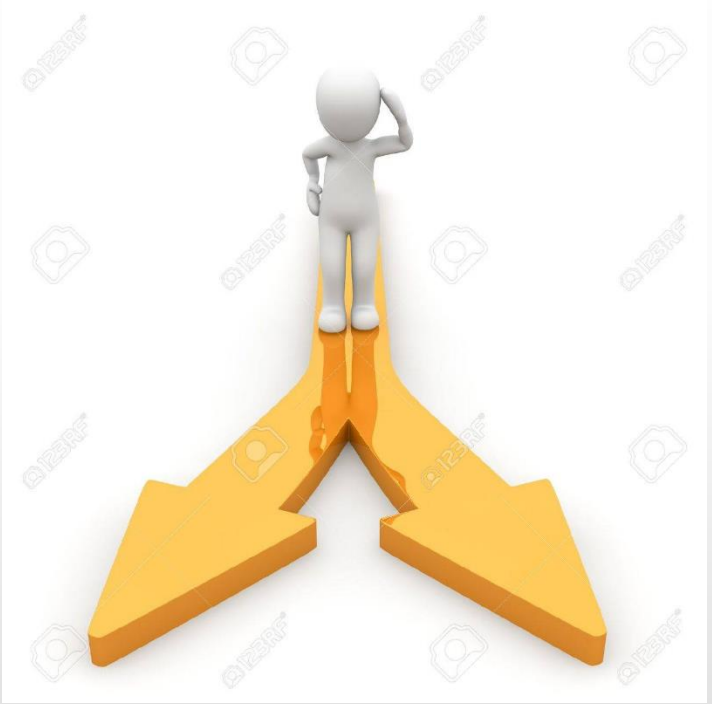
```
printf("%d\n",  x++);
printf("%d\n",  x);
```

Q3

```
printf("%d\n",  ++x);
printf("%d\n",  x);
```

Selection

- Obviously in the real world we make choices

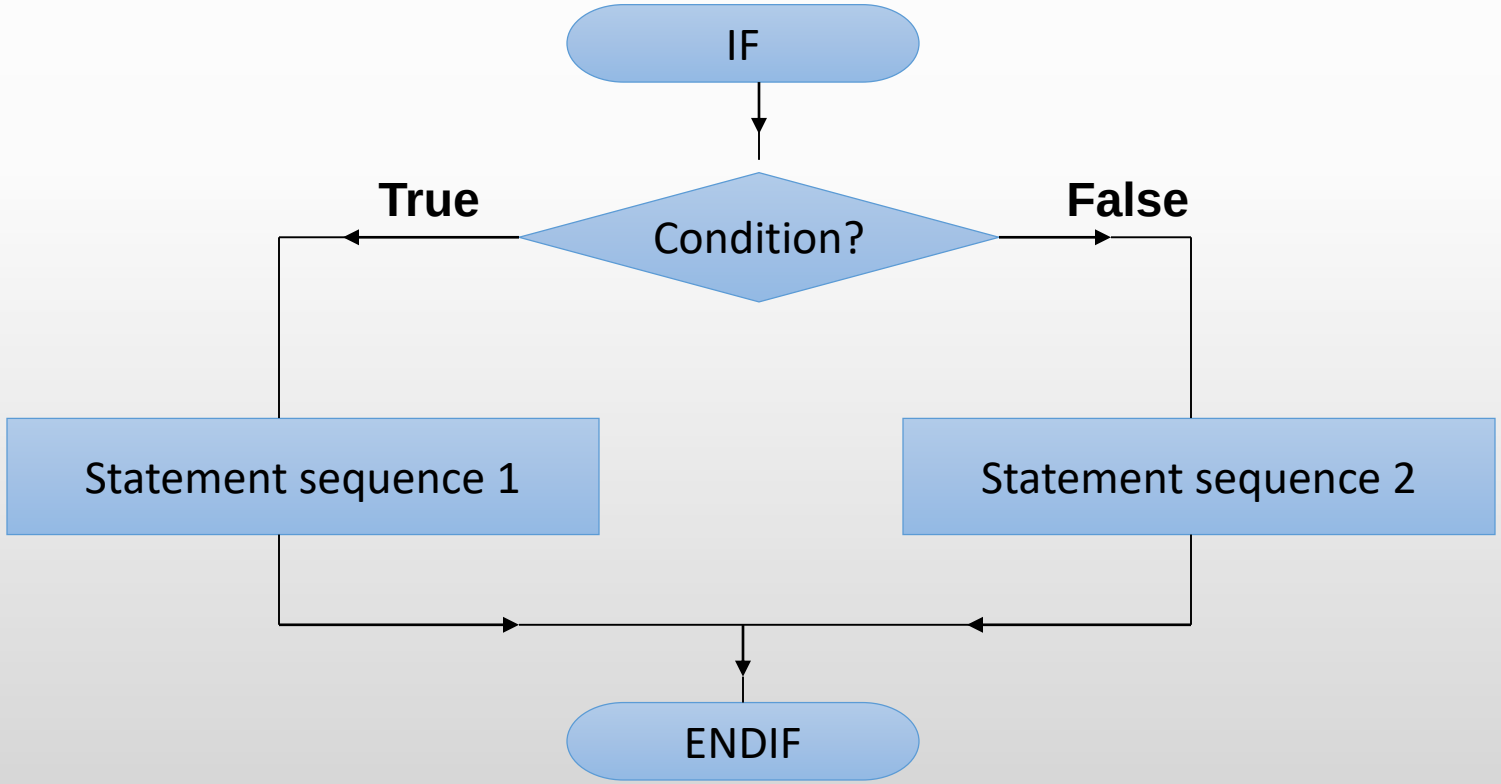


Selection

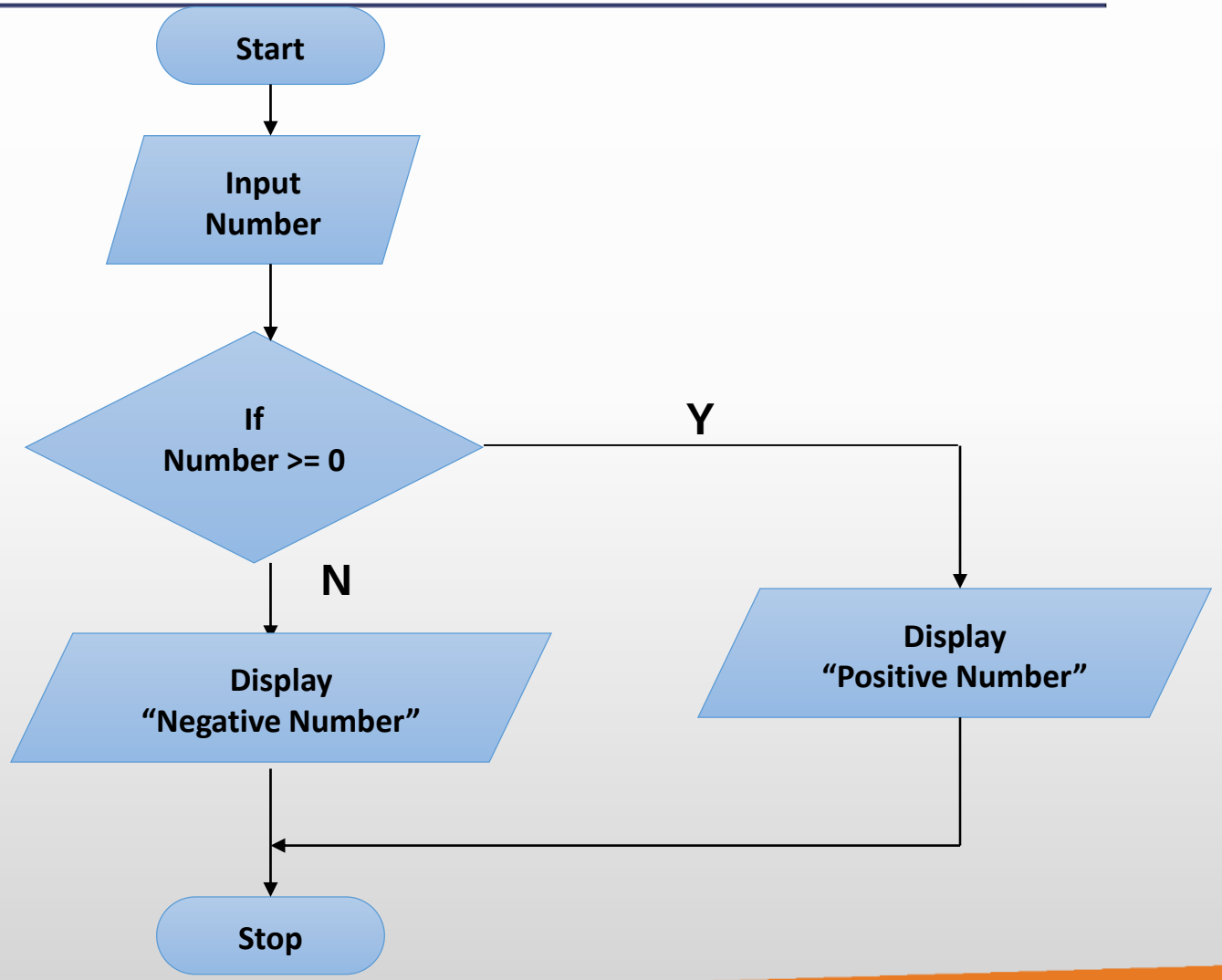
- In solving a problem we can make different choices depending on certain conditions - i.e. we make decisions
- The same can be done in programming as a part of decision making
- Note that even though selection is a separate construct to sequence, the two are combined in the overall solution, and remember that one doesn't replace the other

Selection

- There may be alternative steps that could be taken subject to a particular condition



Example - 01


Start
Input Number
If Number >= 0
Y
N
Display "Negative Number"
Display "Positive Number"
Stop

Exercise

- Draw a flowchart to input two numbers from the keyboard and display the largest number.

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 4 – Selection Statements in C / Character Handling

SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to

  - use the different types of selection statements to select actions (if, if …else, nested if, conditional operator, switch).

  - use getchar( ) function to read characters from the key board

Decision Making using If statement

if statement performs an action if a **condition** is **true**. Conditions in if statements are formed by using the equality operators and relational operators

```
// using if statement
#include <stdio.h>
int main(void)
{

        int no1, no2;
        printf( "%s", "Enter two integers : " );
        scanf( "%d%d", &no1, &no2);        // read two ntegers
        if ( no1 == no2 )     // checking equal
                printf( "%d is equal to %d\n" , no1, no2 );

        if ( no1 != no2 ) // checking not equal
                printf( "%d is not equal to %d\n" , no1, no2 );


        return 0;
 } // end of main function
```

output

```
Enter two integers :3  3
3 is equal to 3
```

```
Enter two integers :5  3
5 is not equal to 3
```

if statement cont…

```
if ( no1 == no2 )
{
     printf( "%d is equal to %d\n" , no1, no2 );
     printf( "%s", "Numbers are same " );
}
```

- To include several statements in the body of an if, enclose the set of statements in braces ({ and })
- A left brace ({) begins the body of each if statement
- A corresponding right brace (}) ends each if statement body
- Any number of statements can be placed in the body of an if statement
- A set of statements contained within a pair of braces is called a **compound statement** or a **block**

Exercise 01

Write a program in C to read two integer numbers from the keyboard and display the largest number.

if …. else Statement

- if ….. else statement performs an action if a condition is true and performs a different action if the condition is false

```
/* printing pass or fail using if .. else  statement */
#include <stdio.h>
int main(void)
{
        int mark;

        printf( "Enter marks : " );
        scanf( "%d", &mark );        // read marks

        if ( mark >= 60 )    // check whether mark greater than or equal to  60
        {
                printf( "%s", "Passed" );
        }
        else
        {
                printf( "%s", "Failed " );
                printf( "You must take this again \n" );
        }

         return 0;
}
```

Conditional Operator

- Conditional operator(?:) is related to the if ….else statement.
- It takes three operands. First operand is a condition. Second is the value if the condition is true. Third is the value if the condition is false.

Example

```
mark >= 60 ? printf( "Passed\n" ) : printf( "Failed\n" );
```

Above statement is same as,

```
if ( mark >= 60 )
    printf( "Passed" );
else
    printf( "Failed" );
```

Nested if…. else statements

- Nested if … else statements handle multiple cases by placing if …else statements inside if …else statements.

```
/* printing grade using nested if .. else  statement */
#include <stdio.h>
int main(void)
{
        int mark;

        printf("%s", "Enter marks : ");
        scanf("%d", &mark);        // read marks

        if ( mark >= 80 )
             printf( "%s", "Grade A" );
        else if ( mark >= 50 )
             printf( "%s", "Grade B " );
        else if ( mark >= 40 )
             printf( "%s", "Grade C " );
         else
              printf( "%s", "Grade F " );

        return 0;
}
```

Switch Statement

- The switch statement is an alternative to the nested if-else statement provided the expressions can be written as:

  (variable == value)

- The switch statement consists of a series of case labels
- Multiple statements can be executed for a given condition and break statement terminates the execution of the condition

Switch Statement - Example

Syntax

```
switch (variable)
{
   case c1:  any_number_of_statements;
             break;

   case c2: any_number_of_statements;
             break;
              …

   default: any_number_of_statements;
}
```

Example

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

        printf( "%s", "Enter score  : " );
        scanf( "%d", &score );        // read score

        switch ( score )
        {
              case 3 : printf( " Congratulations\n" );
                       printf( " Gold Winner\n" );
                       break;
              case 2 : printf( " Silver Winner\n" );
                       break;
              case 1 : printf( " Bronze Winner\n" );
                      break;
              default : printf( " Invalid Score\n" );

        }
        return 0;
}  // end of main function
```

char data type

- Characters are normally stored in variable type char
- Characters can be stored in any integer type variable too
- Characters can be treated as either an *integer* or a *character*
- **getchar** function reads one character from the keyboard
- Characters can be read with **scanf**

by using the conversion specifier %c

```
// reading a character and print messages appropriately
#include <stdio.h>
int main(void)
{
          int grade;

          printf( "%s", "Enter grade  : " );
          grade = getchar();      // read a character

          switch( grade )
          {
                case 'A' :  printf( "%s", "Excellent" );
                            break;
                case 'B' :  printf( "%s", "Good" );
                            break;
                 ……..
                 ……..


          return 0;
}       // end of main function
```

char data type cont…

- Many computers today use ASCII(American Standard Code for Information Interchange) character set

Example:

```
printf( “The character (%c) has the value %d.\n”, ‘a’, ‘a’ );
```

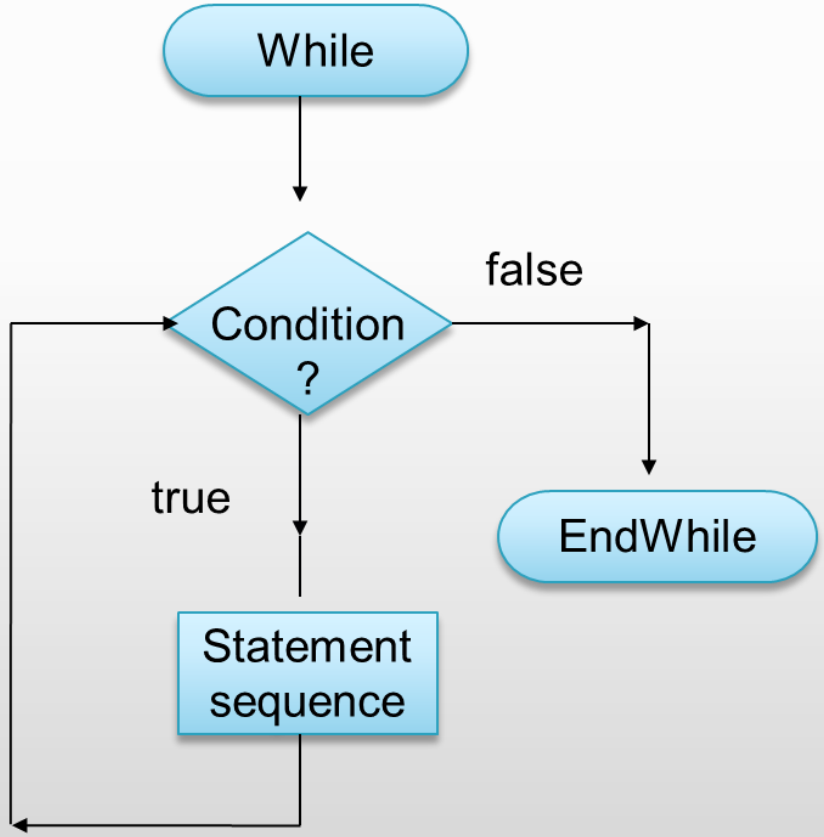
Output :

```
The character (a) has the value 97.
```

- Conversion specifier %c and %d can be used to print character ‘a’ and its integer value
- 97 is the numerical representation of character ‘a’ in the computer.

Iteration

- Certain steps may need to be repeated while, or until, a certain condition is true.
- We call it as a Loop

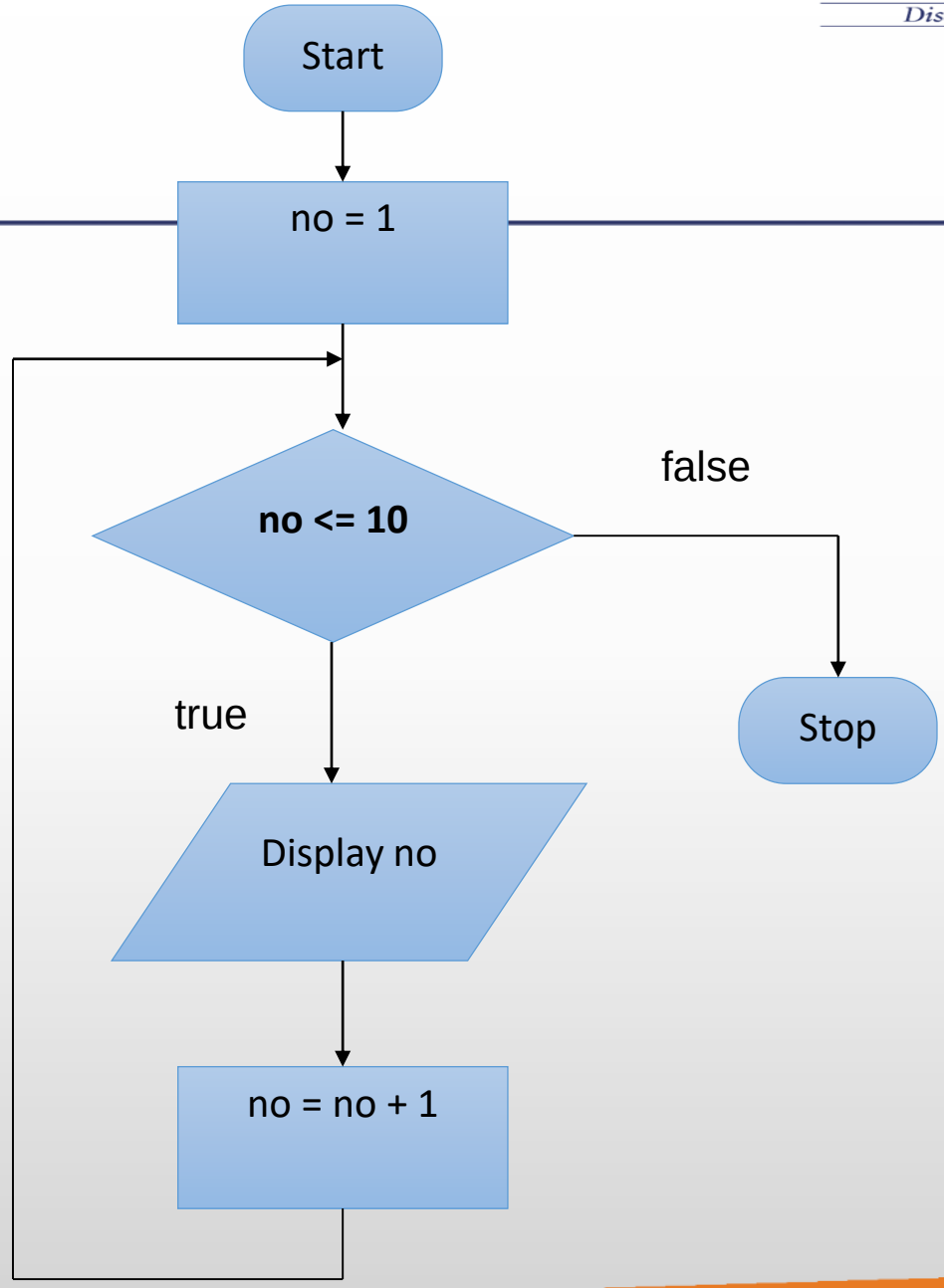


Iteration

- Section(s) of an algorithm are repeated over and over (obviously these loops must eventually terminate)
- This is achieved by a test of whether a condition is true or false
- In a while loop we continue to repeat something **while a condition is true – we terminate the loop when it is false**

Example

- Draw a flowchart to represent an algorithm to display the numbers 1, 2, 3, 4, 5, ….., 10



Exercise 02

- Draw a flowchart to find the sum of 10 numbers entered through the keyboard.

Exercise 03

- Draw a flowchart to find the average of 10 numbers entered through the keyboard.

Summary

- If statement
- If .. Else statement
- Conditional operator
- Nested selection
- Switch statement
- getchar ( )
- Iteration

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 5 – Repetition statements in C

SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - use the **while** repetition statement to execute statements in a program repeatedly.
  - use the **for** repetition statement to execute statements in a program repeatedly.
  - use the **do…while** repetition statement to execute statements in a program repeatedly.
  - Use **break** and **continue** statements to alter the flow of control.

Counter-Controlled Repetition

- Number of repetitions is known before the loop begins execution.
- A control variable is used to count the number of repetitions.
- The control variable is incremented (usually by 1) each time the group of instructions is performed.
- The repetition terminates when the counter exceeds number of repetitions.

Counter Controlled Repetition cont…

- Counter-controlled repetition requires:
  - The name of a control variable.
  - The initial value of the control variable
  - The increment (or decrement) by which the control variable is modified each time through the loop.
  - The condition that tests for the final value of the control variable.

Counter-Controlled Repetition with the while statement

```
//Counter-controlled repetition
# include <stdio.h>
int main(void)
{
        int counter =  1; // initialization
        while (counter  <=  10)  { //repetition condition
                printf("%d ",  counter); // display counter
                ++ counter; // increment
        } // end while
} //end function main
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

General Format of a while Statement

```
while (condition) {
    statements
}
```

- The while statement body may contain single or a compound statement.

Example 02 - while statement

```
// class average program with counter-controlled repetition
# include <stdio.h>
//function main begins program execution
int main(void)
{
        int counter,  grade,  total;
        float average;
        total =0;
        counter = 1;
        while(counter <= 10){ //loop ten times
                printf("Enter grade :");
                scanf("%d", &grade); // read grade from user
                total = total + grade;
                counter = counter + 1; // increment counter
        } //end while
        average = (float)total / 10;
        printf("Class average is %.2f\n", average);
}//end function main
```

output

```
Enter grade : 98
Enter grade : 76
Enter grade : 71
Enter grade : 87
Enter grade : 83
Enter grade : 90
Enter grade : 57
Enter grade : 79
Enter grade : 82
Enter grade : 94
Class average is 81
```

Quiz

- What does the following program print?

```
# include <stdio.h>
int main(void)
{
        int x =  1,  y;
        while ( x  <=  5)  {
                y = x * x;
                printf("%d\n ",  y);
                ++ x;
        } // end while
} //end function main
```

Exercise 01

- Write a program that print all the even integers from 0 to 20.

Sentinel-Controlled Repetition

- When no indication is given of how many times the loop should execute, a sentinel value is used to terminate the loop.
- E.g : type -1 to terminate entering of marks
- A loop should have a statement to obtain data each time the loop is performed.
- sentinel value must chosen so that it cannot be confused with an acceptable input value.

Sentinel-Controlled Repetition

```
// class average program with sentinel-controlled repetition
# include <stdio.h>
int main(void)
{
        int grade,  total, counter;
        float average;

        total = 0;
        counter = 0;

        //get first input from the user
        printf("Enter grade, -1 to end :");
        scanf("%d",&grade);

        while(grade != -1){
                total = total + grade;
                counter = counter + 1;
                // get next grade from user
                printf("Enter grade, -1 to end :");
                scanf("%d",&grade);
        } //end while
        average = (float)total / counter;
        printf("Class average is %.2f\n", average);
}//end function main
```

Output

```
Enter grade, -1 to end : 75
Enter grade, -1 to end : 94
Enter grade, -1 to end : 97
Enter grade, -1 to end : 88
Enter grade, -1 to end : 70
Enter grade, -1 to end : 64
Enter grade, -1 to end : 83
Enter grade, -1 to end : 89
Enter grade, -1 to end : -1
Class average is 82.50
```

Exercise 02

- Write a program that calculates and prints the average of several integers. Assume the last value read with scanf is the sentinel 9999.

Counter-Controlled Repetition with the for statement

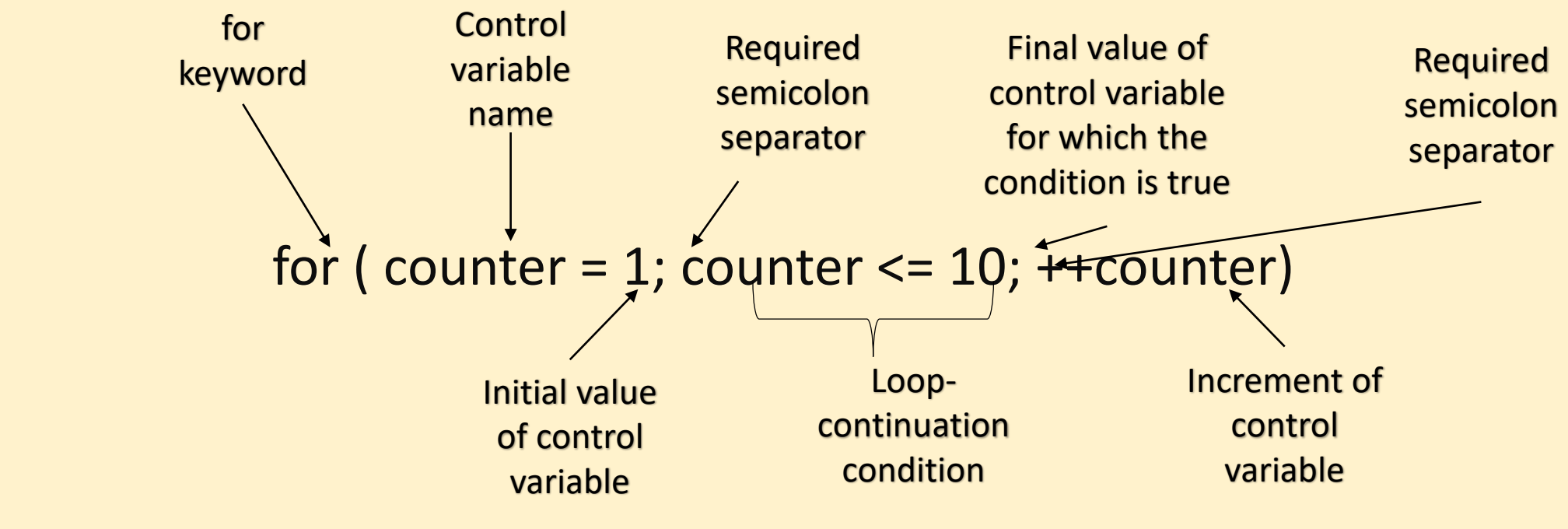
```
# include <stdio.h>
int main(void)
{
        int counter; // define counter

        for( counter  =  1;  counter  <= 10;  ++counter ){
                printf("%d\n", counter);
        }
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

for Statement Header Components



General Format of a for Statement

```
for ( expression1; expression2; expression3){
        statement
    {
```

- Expressions in the for statement's header are optional
- Increment Expression acts like a standalone statement

Quiz

- What does the following program print?

```
# include <stdio.h>
int main(void)
{
        int  x;
        for( x = 3; x <= 15; x +=3 ){
                printf("%d\n", x);
        }
}
```

Exercise 03

- Write a program that will print the following sequence of values . (Hint : use a for loop)

3, 8, 13, 18, 23

do…while Repetition Statement

- Loop continuation condition is checked after the loop body is performed.
- Therefore the loop body will be executed at least once.

```
do{
    statement
}while (condition);
```

Counter-Controlled Repetition with the do…while statement

```
# include <stdio.h>
int main(void)
{
        int counter = 1;

        do{
            printf("%d ", counter);
        } while (++counter <= 10);

}
```

Output

1 2 3 4 5 6 7 8 9 10

break statement

- The break statement, when executed in a while, for, do…while statement causes immediate exit from that statement.
- Program execution continues with the next statement.
- Common uses of the break statement are to escape early from a loop.

break statement example

```
//Using a break statement in a while statement
# include <stdio.h>
int main(void)
{
        int x =  1;
        while ( x  <=  10)  {
                if ( x == 5) {
                    break;
                }
                printf("%d ",  x);
                ++ x;
        } // end while
} //end function main
```

Output

```
1 2 3 4
```

continue statement

- The continue statement, when executed in a while, for and do...while statement, skips the remaining statements in the body of that control statement and perform the next iteration of the loop.
- In while and do...while , loop continuation test is evaluated immediately after the continue statement is executed.
- In the for statement, the increment expression is executed.

continue statement example

```
//Using  the continue statement in a while statement
# include <stdio.h>
int main(void)
{
        int x =  1;
        while ( x  <=  10)  {
                if ( x == 5) {
                    ++x;
                    continue;
                }
                printf("%d ",  x);
                ++ x;
        } // end while
} //end function main
```

Output

1 2 3 4 6 7 8 9 10

Nested iteration

```
# include <stdio.h>
int main(void)
{
        int i, j;
        for ( i = ; i <= 5 ; ++i){
                for ( j = 1; j <= i; ++j){
                        printf(" *" );
                }
                printf("\n");
        }
}
```

Output

```
*
**
***
****
*****
```

Exercise 04

- Write a program that will print the following output.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 6 - Functions

SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - Construct programs using functions.
  - Use math functions in C standard library.
  - Use assertions to test whether the values of expressions are correct.

Introduction

- Most computer programs that solve real-world problems are much larger than the programs that we discuss in the class.
- The best way to develop and maintain large program is to construct it from smaller pieces or modules.
- This technique is called divide and conquer.
- In C language these modules are called functions.
- Another motivation to use functions is software reusability.

C standard library

- C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, input/output.
- e.g: printf

  scanf

  pow

Math Library Functions

- Allows the user to perform certain common mathematical calculations.

Function	Description	Example
sqrt(x)	square root of x	sqrt (900.0) is 30.0
pow(x, y)	x raised to power y (x^y)	pow (2, 7) is 128.0
exp(x)	exponential function e^x	exp (1.0) is 2.718282
log (x)	natural logarithm of x	log(2.718282) is 1.0
ceil (x)	rounds x to the smallest integer not less than x	ceil (9.2) is 10.0 ceil (- 9.8) is – 9.0
floor (x)	rounds x to the smallest integer not greater than x	floor (9.2) is 9.0 floor (- 9.8) is – 10.0

Using math functions in C programs

```
# include <stdio.h>
# include <math.h>
int  main (void)
{
        printf("%.2f", sqrt(900.0);
        return 0;
}
```

output

30.00

```
# include <stdio.h>
# include <math.h>
int  main (void)
{
        float c1 = 13;
        float d = 3.0;
        float f = 4.0;
        printf("%.2f", sqrt(c1 + d * f);
        return 0;
}
```

output

5.00

Quiz

- Write the value of x after each of the following statements is performed:
  a) x = floor (7.5)
  b) x = floor (0.0)
  c) x = ceil (-6.4)
  d) x = pow (5, 2)

Programmer – defined functions

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

        for (x =1; x <= 10; ++x)
                printf(%d “, square( x ));
        puts(“”);
}
int square (int y)
{
        return y * y;
}
```

1 4 9 16 25 36 49 64 81 100

Calling and Called Functions

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

    for (x =1; x <= 10; ++x)
            printf(%d “, square( x ));
    puts(“”);
}
int square (int y)
{
    return y * y;
}
```

Function prototype

Calling Function / Caller

Called Function

- Functions are invoked by a function call.
- The function which invokes a function is called the calling function or caller.
- The function being activated is referred to as the called function.
- A function prototype gives all of the information needed by the calling function to invoke the called function.
- The function prototype is the declaration of the function and must appear before the function is invoked.

Local variables

- All variables defined in function definitions are local variables.
- They can be accessed only in the function in which they are defined.
- Example : int x in the square function

Parameter List

- The parameter list is a comma-separated list that specifies the parameters received by the function when it's called.
- If a function does not receive any values, parameter list is void.
- A type must be listed explicitly for each parameter.
- Example : int y in the square function is a parameter

Function prototype

- The compiler uses function prototypes to validate function calls.

  e.g:

```
int square (int y)
```

- The *int* in parenthesis informs the compiler that square expects to receive an integer.
- The int to the left of the function name informs the compiler that square returns an integer result to the caller.

return statement

- return statement helps the called function to return a value to the calling function.
- If a function does not return a value, the statement

```
return;
```

- If a function does return a result, the statement

```
return expression;
```

The format of a function

```
return-value-type function-name( parameter-list)
{
        definitions
        statements
}
```

- Function name is any valid identifier
- The return-value-type is the data type of the result returned to caller.
- Definitions and statements within the braces form the function body.
- If a function does not return a value, the return-value-type should be indicated as void.

Exercise 1

- Write a function that displays a solid square of asterisks whose side is specified in integer parameter side. For example, if side is 4, the function displays:

```
****
****
****
****
```

Exercise 2

- Write a function called *max* to determine and return the largest of two integers. The integers should be input from the key board in the main program and pass to *max* function.

Passing Arguments By Value

- When arguments are passed by value, a copy of the argument's value is made and passed to the called function.
- Changes to the copy do not affect an original variable's value in the caller.

Passing Arguments By Value - Example

```
# include <stdio.h>
int cubeByValue ( int n ) ;

int main ( void )
{
  int  number = 5;

   number  = cubeByValue (number);

}
```

number

5

```
int cubeByValue (int n)
{
   return n * n * n;
}
```

n

undefined

Passing Arguments By Value - Example

```
# include <stdio.h>
int cubeByValue ( int n ) ;

int main ( void )
{
   int  number = 5;

   number  = cubeByValue (number);

}
```

number

5

```
int cubeByValue (  int n  )
{
   return n * n * n;
}
```

n

5

Passing Arguments By Value - Example

```
# include <stdio.h>
int cubeByValue ( int n ) ;

int main ( void )
{
   int  number = 5;

   number  = cubeByValue (number);

}
```

number

5

```
int cubeByValue (  int n  )
{
   return n * n * n;
}
```

125

n

5

Passing Arguments By Value - Example

```
# include <stdio.h>
int cubeByValue ( int n ) ;

int main ( void )
{
   int  number = 5;

   number  = cubeByValue (number);

}
```

number

5

125

```
int cubeByValue (  int n  )
{
   return n * n * n;
}
```

n

undefined

Passing Arguments By Value - Example

```
# include <stdio.h>
int cubeByValue ( int n ) ;

int main ( void )
{
   int  number = 5;

   number  = cubeByValue (number);

}
```

number

125

125

125

```
int cubeByValue (  int n  )
{
   return n * n * n;
}
```

n

undefined

Block Scope

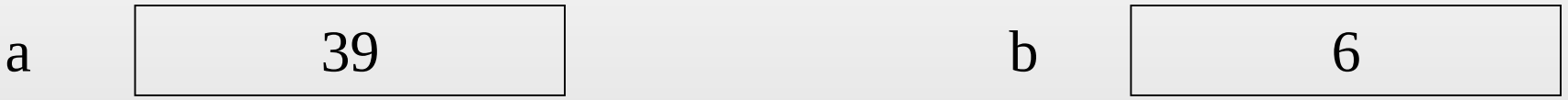
- The scope of an identifier is the portion of the program in which the identifier can be referenced.
- Identifiers defined inside a block have a block scope.
- Block scope ends at the terminating right brace.
- Local variables defined at the beginning of a function have block scope.
- when blocks are nested and inner and outer blocks both have the same identifier name, identifier in the outer block is hidden until the inner block terminates.

Block scope example

```
{  //start of outer block
   int a = 39;
   int b = 6;
   printf( “a=   %d  and b= %d \n“, a, b);
   {  // start of inner block
            float a = 26.25;
            int c = 30;
            printf(“Now a= %.2f and b= % d  and c= %d\n”, a, b, c);
    }  //end inner block
   printf( “Finally a= %d  and b = %d \n, a, b);
} // end of outer block
```

Block scope example

```
{  //start of outer block
   int a = 39;
   int b = 6;
   printf( “a=   %d  and b= %d \n“, a, b);
```



a = 39 and b = 6

Block scope example

```
{  //start of outer block
   int a = 39;
   int b = 6;
   printf( "a=   %d  and b= %d \n", a, b);
   {  // start of inner block
         float a = 26.25;
         int c = 30;
         printf("Now a= %.2f and b= % d  and c= %d\n", a, b, c);
    }  //end inner block
```

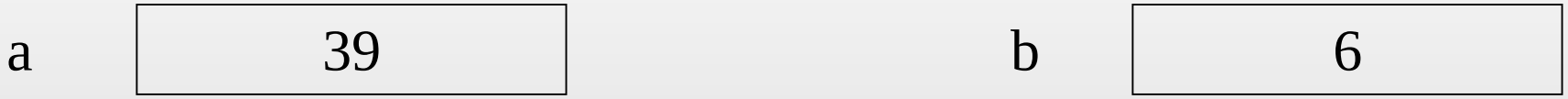
a	26.25	b	6	c	30

a = 39 and b = 6

Now a = 26.25 and b =6 and c = 30

Block scope example

```
{    //start of outer block
     int a = 39;
     int b = 6;
     printf( “a=   %d  and b= %d \n“, a, b);
     {  // start of inner block
                    float a = 26.25;
                    int c = 30;
                    printf(“Now a= %.2f and b= % d  and c= %d\n”, a, b, c);
    }  //end inner block
     printf( “Finally a= %d  and b = %d \n, a, b);
} // end of outer block
```



a = 39 and b = 6

Now a = 26.25 and b =6 and c = 30

Finally a = 39 and b = 6

File scope

- An identifier declared outside any function has file scope.
- Such identifies are known to all the function in the program
- Global variables, function definitions and function prototypes

has file scope.

```
# include <stdio.h>

int x = 1; // global variable

int main(void )
{
        printf("%d", x);
        return 0;
}
```

output

```
1
```

Assert

- assert.h contains information for adding diagnostics that aid program debugging.
- Assert test the value of an expression at execution time.
- If the value is false (0) , assert print an error message and terminate the program.

Assert – Example 1

- Write a program which print numbers greater than 10.

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


int main(void )
{
        int x;
        printf("Pls input a number");
        scanf("%d", &x);
        assert(x >= 10);
        printf("The value of x is  %d", x);
        return 0;
}
```

Output

Pls input a number : 12
The value of x is 12

Pls input a number : 8
Assertion 'x>=10' failed

Assert – Example 2

- Write a function called grade( ) which takes a mark as a argument and return the grade according to the following table. Write another function called test_grade( ) which contain test cases to debug the grade( ) function.

Marks Range	Grade
0 to 39	F
40 to 59	C
60 to 74	B
75 to 100	A
Mark< 0 and Mark >100	X

Assert – Example 2 – grade( ) function

```
char grade(int marks) {
  char result;
  if (marks < 0)
    result = 'X';
  if (marks < 40)
    result = 'F';
  else if (marks < 60)
    result = 'C';
  else if (marks < 75)
    result = 'B';
  else if (marks <= 100)
    result = 'A';
  else
    result = 'X'; // Error (invalid mark)
    return result;
}
```

Assert – Example 2 – test_grade( ) function

```
void test_grade() {
  assert(grade(20) == 'F');
  assert(grade(50) == 'C');
  assert(grade(70) == 'B');
  assert(grade(78) == 'A');
  assert(grade(-10) == 'X');
  assert(grade(110) == 'X');
  // boundary conditions
  assert(grade(0) == 'F');
  assert(grade(40) == 'C');
  assert(grade(60) == 'B');
  assert(grade(75) == 'A');
  assert(grade(100) == 'A');
  printf("grade() unit tests passed\n");
}
```

Assert – Example 2 – main function

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

char grade(int marks);
void test_grade();

int main( void ) {

   test_grade();
   return 0;
}
```

Output

Assertion ‘grade(-10) ==‘X’ failed.

Modify the grade( ) function as follows and run the program

```
if (marks < 0)
      result = 'X';
 else  if (marks < 40)
      result = 'F';
……..……..
……..……..
```

Summary

- C math library functions
- User-defined functions
- Scope of a variable
- Parameter passing by value
- Assert statement

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 7 - Arrays

SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - To define an array, initialize an array and refer to individual elements of an array.
  - To pass array to functions.
  - To define and manipulate multidimensional arrays.
  - To use string functions to handle character arrays.

Introduction

- Array is a data structure which store the data items of the same data type.
- Array store all the data items in continuous memory locations.

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62

Defining Arrays

- To define a array, we need to specify the type of data elements , name and the number of elements (size).

```
int c[8]
```

- The above definition reserves 8 elements for integer array c.
- Array name, like other variables can contain only letters, digits and underscore and cannot begin with a digit.

Using Arrays

- To refer to a particular location or element in the array, we need to specify array's name followed by the position number (index, subscript) of the particular element in square brackets.
- First element in the array is the zeroth element. Last element is size – 1.

c[0], c[1], c[2],c[3].....c[7]

Using Arrays

- Print the first element in the array.

```
printf("%d", c[0]);
```

- Print the sum of first three elements in the array.

```
printf("%d", c[0] + c[1] + c[2]);
```

- Add 2 to the fifth element

```
c[4] += 2;
```

Defining and initializing an array

```
//initializing the elements of an array to zeros
 # include <stdio.h>
int  main(void)
{
        int n[ 5 ]; // n is an array of 5 integers
        int i; // counter

        //initialize elements of array  n to 0
        for( i  = 0; i  < 5; ++i)
                n[ i ] = 0;

        printf("%s%13s\n",  "Element",  " Value");

        //output contents of array n in a tabular format
        for( i  = 0; i < 5; ++i)
                printf("%7d %13d\n", i , n[ i ]);
        return 0;
}
```

Initializing array using an initializer list

```
/*initializing the elements of an array using an initializer list */
 # include <stdio.h>
int  main(void)
{
    int n[5] = {5, 12, 34, 56, 23};
    int i;
    printf("%s%13s\n",  "Element",  " Value");
    //output contents of array n in a tabular format
    for( i  = 0; i < 5; ++i)
                              printf("%7d %13d\n", i , n[ i ]);
      return 0;
}
```

Initializing array using an initializer list

```
/*initializing the elements of an to zero
# include <stdio.h>
int  main(void)
{
    int n[5] = { 0 };
    int  i;

    printf("%s%13s\n",  "Element",  " Value");

    //output contents of array n in a tabular format
    for( i  = 0; i < 5; ++i)
                        printf("%7d %13d\n", i , n[ i ]);
}
```

Specifying an array's size with a symbolic constant

```
 # include <stdio.h>
# define SIZE 10
int  main(void)
{
        int a[ SIZE ];
        int  j; // counter

        for( j = 0; j   < SIZE; ++j)
                a[  j  ] = 2 + 2 * j;

        printf("%s%13s\n",  "Element",  " Value");

        for( j  = 0; j < SIZE; ++j)
                printf("%7d %13d\n", j , a[ j ]);
}
```

Summing the Elements of an Array

```
# include <stdio.h>
# define SIZE 12
int  main(void)
{
        int a[ SIZE ] ;
        int i;
        int total = 0; // sum of array

        for( i = 0; i   < SIZE; ++i)
        {
                printf("\na[ i ] = ");
                scanf("%d", &a[ i ]);
        }
        for( j = 0; j   < SIZE; ++j)
                total += a[ j ];
        printf("Total of array elements is %d \n", total);
}
```

Exercise 1

- Write a C program to the following.
  - Define an integer array *counts* with 10 elements.
  - Initialize all elements to zeros.
  - Read and store 10 numbers each of which is between 10 to 100.
  - Find the maximum number from the stored numbers.

Storing *strings* in character arrays

- A string can be stored in a character array as follows:

```
char  string1 [ ] = “first”;
```

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

```
scanf( “%19s”, string1);
```

- Function scanf will read characters until space, tab, newline or end-of-file indicator is encountered.

Display character strings

- A character array representing a string can be printed as follows:

```
printf("string1 is : %s\n", string1);
```

```
for ( i= 0; i < SIZE && string1 [ i ] != '\0'; ++i){
        printf("%c", string1[ i ]);
}
```

Function strcpy

- strcpy copy the entire string in array x into y

```
# include < stdio.h>
# include <string.h>
# define SIZE1 25
# define SIZE2 15
int main ( void )
{
      char x[ ]= 'Happy Birthday to You";
      char y[ SIZE1];

      strcpy( y , x );
      printf("The string in array y is : %s\n", y);
      return 0;
}
```

Output: The string in array y is : Happy Birthday to You

Function strlen

- strlen takes a string as an argument and return the number of characters in the string.

```
# include < stdio.h>
# include <string.h>
int main ( void )
{
      char string1[ ]= 'I love C programming";
      printf("The length of string1 is %d", strlen(string1));
return 0;
}
```

Output: The length of string1 is 20

Passing Arrays to Functions

- To pass an array argument to a function, specify the array's name without any brackets.

```
# include < stdio.h>
# define SIZE 5
void modifyArray(int b[ ], size_t size);

int main ( void )
{
      int a[ SIZE]  = { 0, 1, 2, 3, 4};
      modifyArray(a, SIZE);
      return 0;
}
```

```
void modifyArray( int b [  ], size_t size)
{
            size  j;
           // multiply each array element by 2
           for( j  = 0; j < size; ++j)
                      b[j] *= 2;
}
```

Passing Arrays to Functions

```
# include <stdio.h>
# define SIZE 5
void modifyArray( int b[ ], size_t size);
int  main(void)
{
    int a[ SIZE ] = {0, 1, 2, 3, 4};
    size_t  i; // counter
    //output original array
    for( i = 0; i   < SIZE; ++i)
                        printf("%3d", a[ i ]);
    puts(" ");
    modifyArray( a , SIZE);
    // output modified array
    for( i  = 0; i < SIZE; ++i)
                        printf("%3d ", a[i ]);
}
```

Output

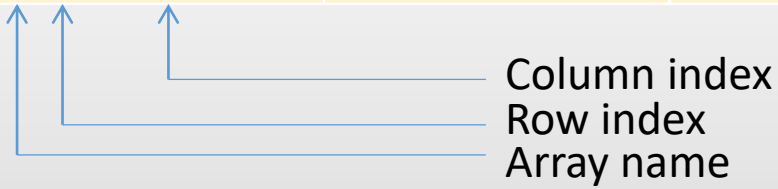
Original Array : 0 1 2 3 4
Modified Array : 0 2 4 6 8

Multidimensional Arrays

- C language have arrays with multiple subscripts.
- These arrays are refers to as multidimensional arrays.
- Multidimensional arrays are used to represent table of values consisting of information arranged in rows and columns.
- A array with two subscripts is called **double-subscripted or Two-Dimensional** array.

Two-Dimensional Array

	Column 0	Column 1	Column 2	Column 3
Row 0	a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]
Row 1	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]
Row 2	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]



Define and initialize 2D array

```
//initializing multidimensional arrays
 # include <stdio.h>
int  main(void)
{
    int array1[ 2 ][ 3 ] = { { 1, 2, 3},{4,5, 6}};
    int array2[ 2 ][ 3 ] = {  1, 2, 3, 4, 5};
    int array3[ 2 ][ 3 ] = {{1, 2}, {4}};

    for( i  = 0; i  < =1; ++i){
                        for( j  = 0; j <= 2; ++j)
                            printf("%d\n", array1[i][j]);
            printf("\n");
     }
```

```
for( i  = 0; i  < =1; ++i){
            for( j  = 0; j <= 2; ++j)
                printf("%d\n", array2[i][j]);
            printf("\n");
        }
            for( i  = 0; i  < =1; ++i){
              for( j  = 0; j <= 2; ++j)
                printf("%d\n", array3[i][j]);
            printf("\n");
        }
return 0;
}
```

Define and initialize 2D array

Values in array1 by row are:

1 2 3

4 5 6

Values in array2 by row are:

1 2 3

4 5 0

Values in array3 by row are:

1 2 0

4 0 0

Summing the Elements of a 2D Array

```
# include <stdio.h>
# define SIZE 12
int  main(void)
{
     int row. column;
     int a[ 2][3];
     int total = 0;
    for( row  = 0; row  < =1; ++row){
                for( column  = 0; column <= 2; ++ column)
                {
                                printf("\na[ row][column ] = ", row, column);
                                scanf("%d",  &a[ row ][column ]);
                }
                for( row  = 0; row  < =1; ++row)
                                for( column  = 0; column <= 2; ++ column)
                                                total += a [row] [column];
    pritntf("The total of the elements of the array : %d", total);
  return 0;
}
```

Summary

- Handling 1D arrays
- String manipulation
- Passing arrays to functions
- Handling 2D arrays

SLIIT

Discover Your Future

IT1010 – Introduction to Programming

Lecture 8 – File Handling

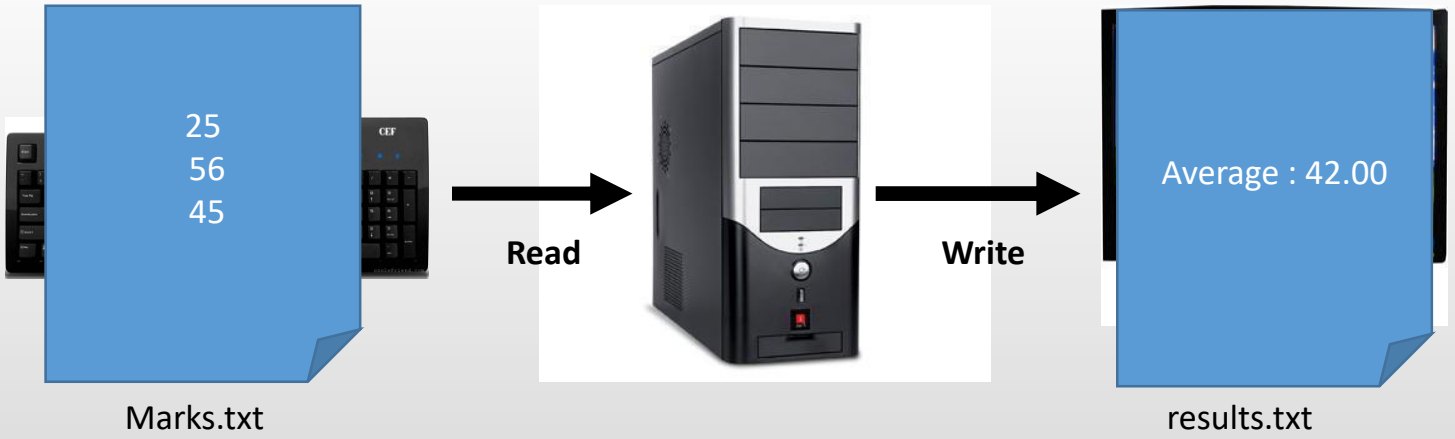
SLIIT
FACULTY OF COMPUTING

Objectives

- At the end of the Lecture students should be able to
  - Create, update and process data files for storing and reading data.

Sequential Files

- Storage of data in variables and arrays is temporary.
- Data is lost when program terminates.
- Files are used to store data permanently.



Creating Sequential Access Files

Declaring a file pointer

```
FILE *cfPtr;
```

Open a file to write data.

```
cfPtr = fopen("number.dat", "w");
```

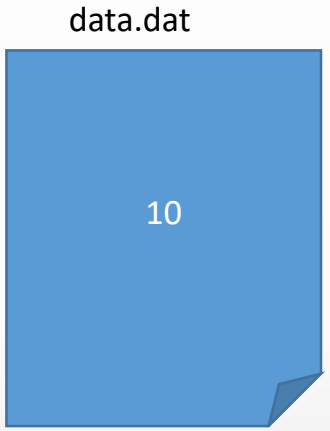
Creates "number.dat" file to store/write data

Writing data to a sequential-access file

```
#include <stdio.h>
int main(void)
{
      int number = 10;

      FILE *cfPtr;
      cftr = fopen("number.dat", "w");

      if ( cfPtr == NULL)
      {
          printf("Cannot create file\n");
          return -1;
       }
       fprintf(cfPtr, "%d\n", number);
       fclose(cfPtr);
       return 0;
}
```



Close each file as soon as it's no longer needed.

Exercise 1

- Write a program to input the id, name and average marks of a student from the keyboard and write the data to “marks.dat” file.

Writing multiple records to a sequential file

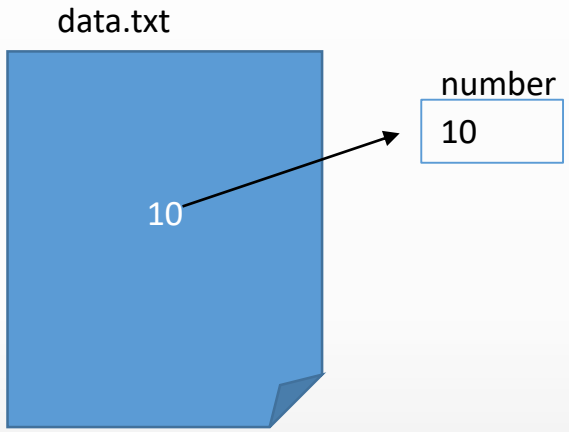
```
int main(void)
{
        char ID[10];
        char name[ 30];
        double avgMarks;
        int i;
        FILE *cfPtr;
        cfPtr = fopen(“marks.dat", "w");
        if (cfPtr == NULL)
        {
                printf("File cannot be open");
                return -1;
        }
        for(i = 1; i <= 5; ++i)
        {
                printf("Pls input the student ID");
                scanf("%s”, ID);
                printf("Pls input the name");
                scanf("%s", name);
                printf("Pls input the average Marks");
                scanf("%lf",& avgMarks);
                fprintf(cfPtr, "%s %s %.2f\n", ID, name, avgMarks);
        }
        fclose(cfPtr);
        return 0;
}
```

Reading data from a sequential – Access file

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

    FILE *cfPtr;
    cftr = fopen("number.dat", "r");

    if ( cfPtr == NULL)
    {
        printf("File could not be pened\n");
        return -1;
    }
    fscanf(cfPtr, "%d", &number);
    printf("Number is : %d \n", number );
    fclose(cfPtr);
     return 0;
}
```



Reading data from a file

```
# include <stdio.h>
int main(void)
{
        FILE *cfPtr;
         char ID[10];
        char name[ 30];
        double avgMarks;
        cfPtr = fopen(“marks.dat", “r");
        if ( cfPtr == NULL)
        {
                printf("File cannot be open");
                return -1;
        }
        fscanf(cfPtr, "%s %s %lf", ID, name, &avgMarks);
        printf "%s %s %lf", ID, name, avgMarks);
        fclose(cfPtr);
        return 0;
}
```

Reading multiple records from a sequential file

```
int main(void)
{
        FILE *cfPtr;
         char ID[10];
        char name[ 30];
        double avgMarks;
        cfPtr = fopen("marks.dat", "r");
        if ( cfPtr == NULL)
        {
                printf("File cannot be open");
                return -1;
        }
        fscanf(cfPtr, "%s %s %lf", ID, name, avgMarks);
        while (!feof(cfPtr))
        {
                printf ("%s %s %lf", ID, name, avgMarks);
                fscanf(cfPtr, "%s %s %lf", ID, name, &avgMarks);
        }
        fclose(cfPtr);
        return 0;
}
```

File Opening Modes

Mode	Description
r	Open an existing file for reading
w	Create a file for writing. If the file already exists, discard the current contents
a	Append; open or create a file for writing at the end of the file
r+	Open an existing fil for update (reading and writing)
w+	Create a file for update. If the file already exists, discard the current contents
a+	Append: open or create a file for update; writing is done at the end of the file.

Summary

- Opening data files for reading and writing
- Reading a data from a file
- Writing data to a file
- File operation modes