PROGRAMING IN C

Module 1

Basics of Computer Hardware and Software

Basics of Computer Architecture: processor, Memory, Input& Output devices

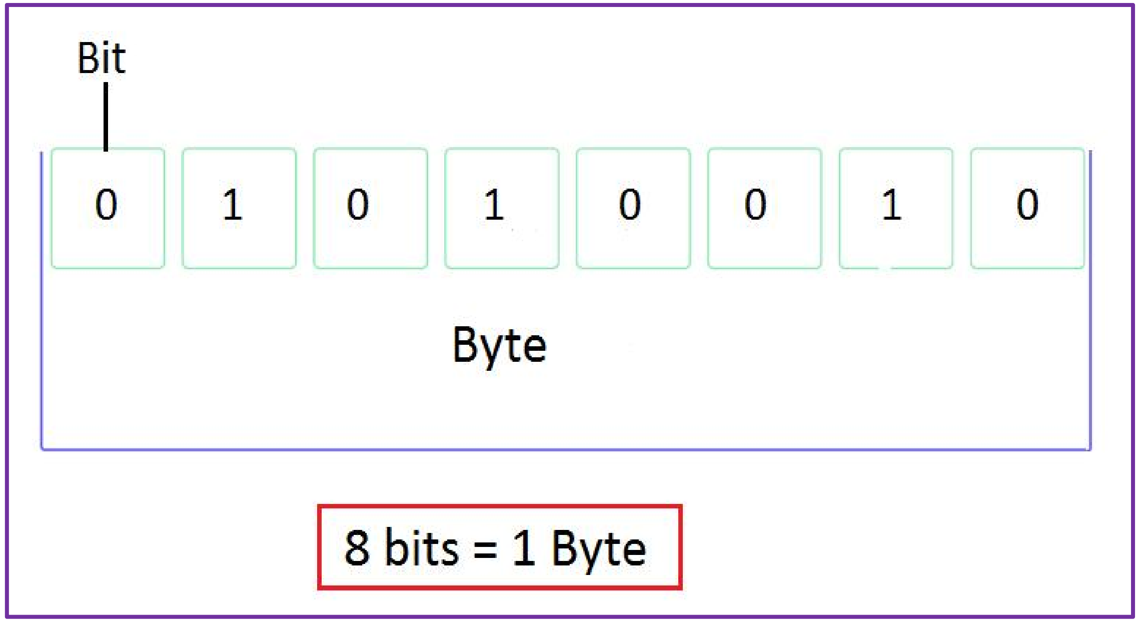
Application Software & System software: Compilers, interpreters, High level and low level languages

Introduction to structured approach to programming, Flow chart Algorithms, Pseudo code (*bubble sort, linear search - algorithms and pseudocode*)

Data processing in computer

Bit Byte Word

- Every information stored in a computer is encoded as a combination of zeros & ones
- These zeros & ones are called **bits** (binary digits)



Data processing in computer

Bit	• Is a single binary digit either a **0** or **1**
Byte	• Eight bits are called a byte • Normally single character occupy 1 byte
Nibble	• Half of a byte or 4 bits
Word	• Group of bits • Varies from machine to machine • Eg: 16-bit, 32-bit, 64-bit words

Data processing in computer

<table>
<tr><td colspan="16">Word</td></tr>
<tr><td colspan="8">Byte</td><td colspan="8">Byte</td></tr>
<tr><td colspan="4">Nibble</td><td colspan="4">Nibble</td><td colspan="4">Nibble</td><td colspan="4">Nibble</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Data processing in computer

Computer bus:

- Is a communication system that transfers data between components
- The size or width of a bus is how many bits it carries
  - Eg: 8 bits, 16 bits, 32 bits, 64 bits etc.

Address bus: Transfer **address of** memory

Data bus: Send and receive **data** to and from memory, CPU, input-output devices

Control bus: Send **control** signals between processor and other components

Computer Software

Software: Is a program/set of programs that instructs the computer what to do

- Includes **programs**, procedures, and routines associated with the operation of a **computer** system

Two types

1. System software
2. Application software

Computer Software

Application software

- Set of one or more programs designed to perform some specific application
- Eg: MS Word, Photoshop, AutoCAD, Web Browsers

Computer Software

System software

- Set of one or more programs designed for computer system management
- It control the operations of computer system
- Acts as an interface between application software & computer hardware
- Eg:
  - Operating system (windows)
  - Language translators (compilers, interpreters)
  - Utility programs (anti-virus, disk cleaners)

Operating System

Operating system (OS) is system software which operates the computer system & manages its resources

- **Acts as an interface** between user and hardware
- **Manages the computer system resources** such as memory, processor, storage, input-output devices and files
- Eg: DOS, windows, LINUX

Operating System – Basic Functions

Memory management

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use
- Allocating memory to running programs & reallocating the memory when programs are terminated

Processor management

- In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**
- Allocates the processor (CPU) to a process & de-allocates processor when no longer required
- Keeps tracks of processor and status of process

Operating System – Basic Functions

Device management

- Operating System manages device communication via their respective drivers
- Keeps tracks of all devices
- Allocate input/output devices to running processes and decides on the needed time period

File management

- Manages the file system
- Opening & closing files, provides access permission to files, keeps track of files, their status & memory locations

Computer Language

- The **computer language** is defined as code or syntax which is used to write programs for specific applications
- The **computer language** is used to communicate with **computers**
- **Computer language** can be classified into three:
    1. Machine language
    2. Assembly language
    3. High-level language

Machine Language

Machine language: Is the language understood by a computer without any translation

- Made up of instructions and data that are **all in binary numbers (0 and 1)**
- Is a **low level language / first generation language**

```
0011000000000000 ; read n -> acc ;
1011000000001010 ; jump to Done if n < 0. ;
0101000000010000 ; add sum to the acc ;
0010000000010000 ; store the new sum ;
1001000000000000 ; go back & read in next number ;
0001000000010000 ; load the final sum ;
0100000000000000 ; output the final sum ;
0000000000000000 ; stop ;
0000000000000000 ; 2-byte location where sum is stored ;
0000000000000000  ;
0000000000000000  ;
```

Machine Language

•Advantage

- Fast program processing since no translator required

Disadvantages

- Very difficult to program
- Programmer has to remember a lot of codes
- Hardware dependent
- Prone to errors while writing code
- Difficult to debug the program

Assembly Language

- A low level language/second generation
- Is almost like the machine code , except that it uses words in place of binary digits
- These words are called mnemonics
- Is machine dependent

```
SUB32   PROC               ; procedure begins here
        CMP   AX,97        ; compare AX to 97
        JL    DONE         ; if less, jump to DONE
        CMP   AX,122       ; compare AX to 122
        JG    DONE         ; if greater, jump to DONE
        SUB   AX,32        ; subtract 32 from AX
DONE:   RET                ; return to main program
SUB32   ENDP               ; procedure ends here
```

Assembly Language

•Advantages:

- Assembly language is easier to understand than machine language
- Easier to correct errors and modify program instructions
- Has the same efficiency of execution as the machine level language

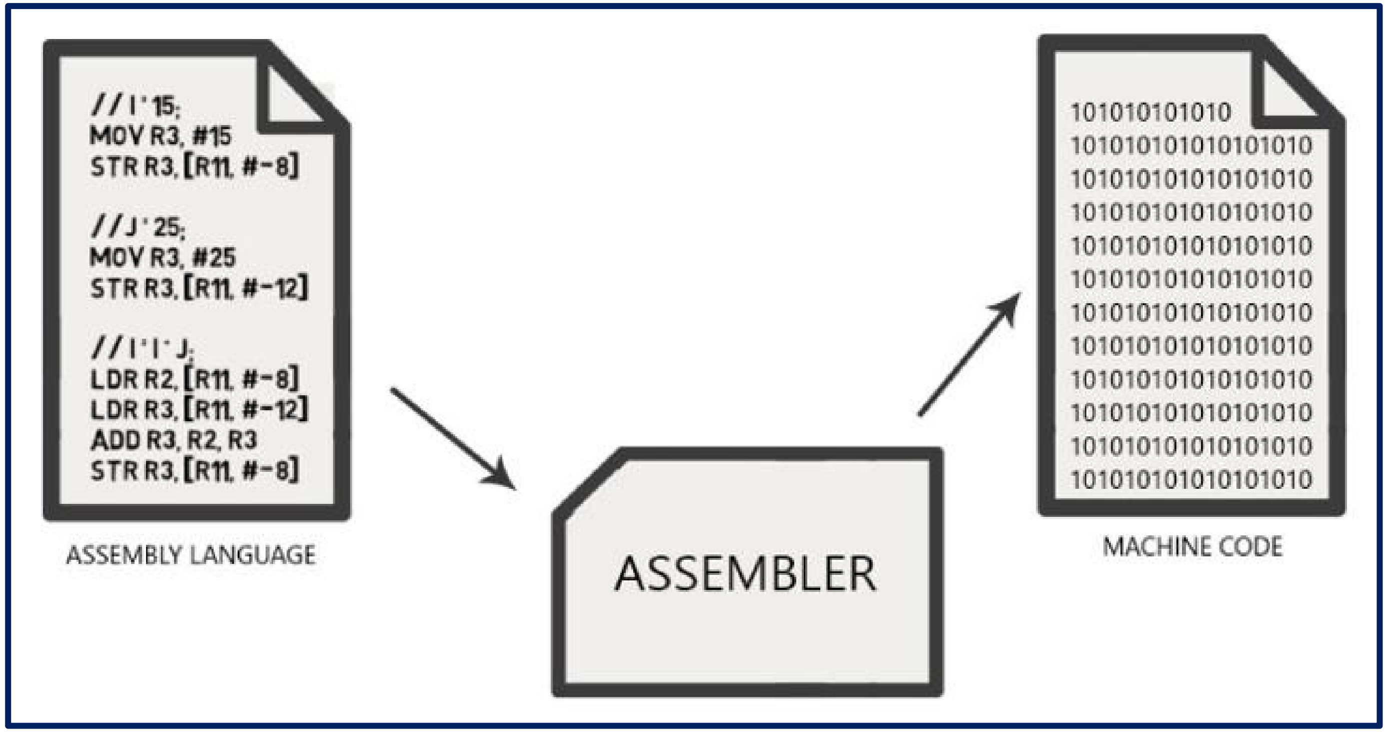
Disadvantages:

- Assembly language is machine dependent

Assembly Language

Assembler:

Translator which converts program in assembly language (source code) to machine language (object code)



PROGRAMING IN C

High Level Language

- High level languages were developed to make programming easier
- The instruction sets are more compatible with human language
- Eg: C, Java
- Programs written in high level languages must also be translated into machine language

High Level Language

Advantages:

- High level languages are programmer friendly
- It is machine independent language & program oriented
- Less prone to errors
- Easy to find and debug errors

Disadvantage:

- Slow Execution

Translator

Translator is a program that converts source code into object code

Generally, there are three types of translators:

1. Compilers
2. Interpreters
3. Assemblers

Compiler

- A compiler is a special program that processes statements written in a programming language (source code) and translates them into machine language (object code)
- C, C++
- Compilers **translate** the **entire program** to machine language **before executing**
- **Steps:**
  - **Lexical analysis:** To recognise the strings in the source code
  - **Parsing**: To analyse the grammatical structure of statement
  - **Code generation**: Generation of object program

Compiler

- **Lexical analysis:** To recognise the strings in the source code like variables, functions, etc.

int value = 100;

Int – keyword, value – identifier, = operator, ; symbol

- **Parsing**: To analyse the grammatical structure of statement
- It checks if the given input is in the correct syntax of the programming language

- **Code generation:** Generation of object program in low level/machine language equivalent to source code

Compiler

- An object code need services from operating systems or utility programs

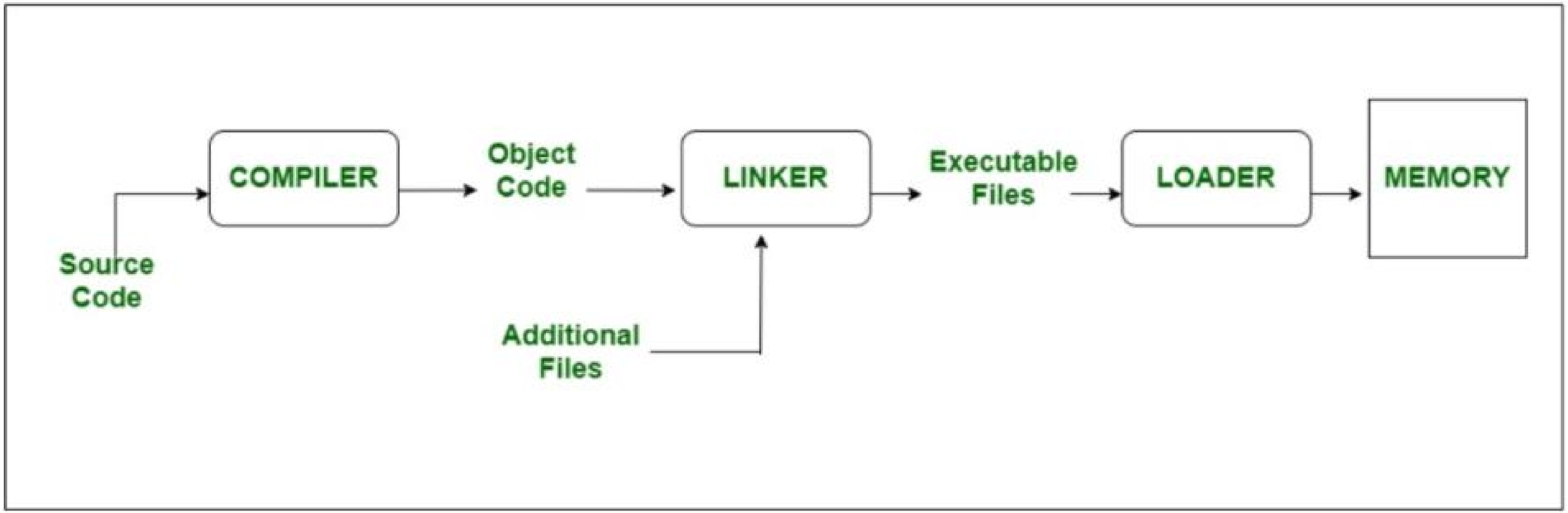
Linker:

- Is special program that combines the object files generated by compiler to create an executable file
- It also merges two or more separate object programs and establishes link among them

Loader:

- Is special program that takes input from linker, loads it to main memory, and prepares this code for execution by computer
- Loader allocates memory space to executable module in main memory

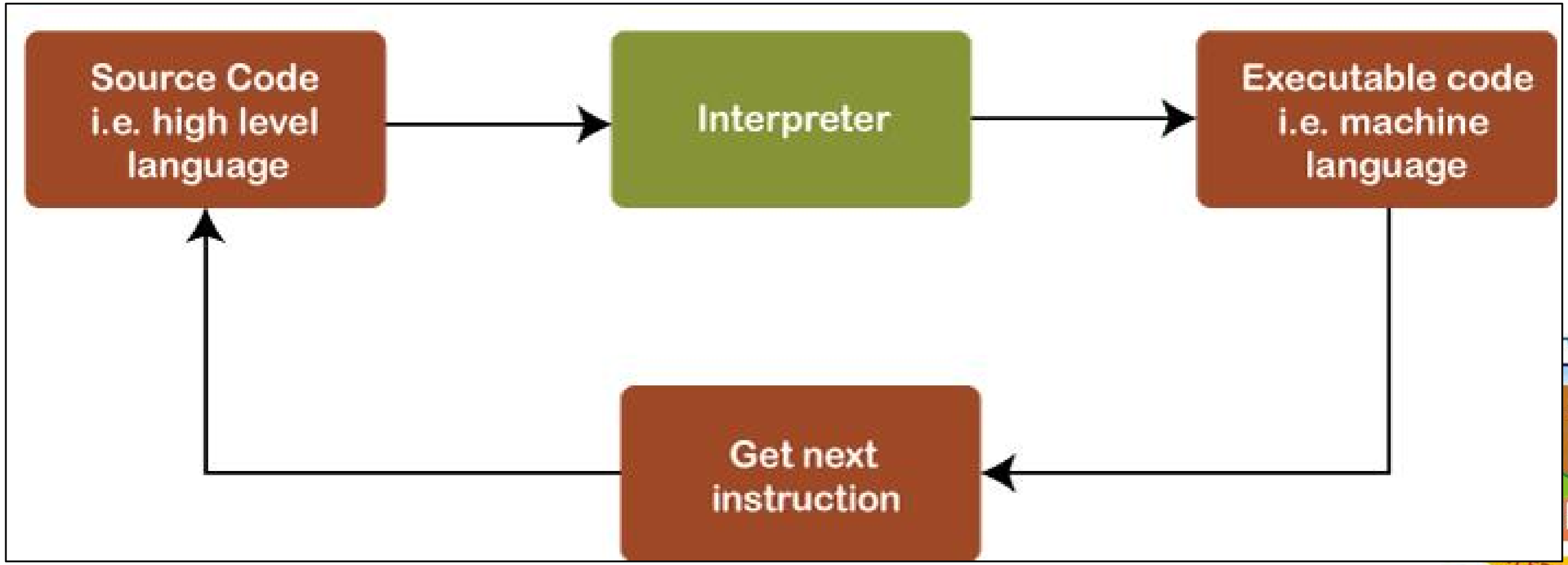
Compiler



- Once compiling & linking is done, the program can be loaded & executed any number of times without any need to return to source code
- If any change is to be made, it must be done in source code and modified program can be complied and linked and loaded

Interpreter

- A special program that processes statements written in a programming language (source code) and translates them into machine language (object code)
- Interpreter translates one statement at a time & executes it
- Translation happening during the execution phase
- Python, MATLab



Compiler	Interpreter
Takes entire program as input	Works on statement by statement
Generates an intermediate code - object code	Does not generate an intermediate code
Memory requirement is more since object code is generated	More memory efficient
Executes control/logical statements like *if, else* faster	Executes control/logical statements slower than compiler
Takes more time to execute source code but overall execution time is comparatively faster	Takes less time to execute source code but overall execution time is comparatively slower
Once compiled can run any time	Programs interpreted line by line every time when run
Errors are reported after entire program is checked	Error is reported at once when an error is encountered
Debugging is difficult	Easier to debug since errors are reported when encountered

Structured Program

- **<u>Structured Programming (modular programming):</u>** Is a programming approach which is a disciplined and ordered approach to develop a program
- Code will execute instruction by instruction one after the other
- The program may be divided into independent group of statements or modules

KNOWLEDGE IS POWER

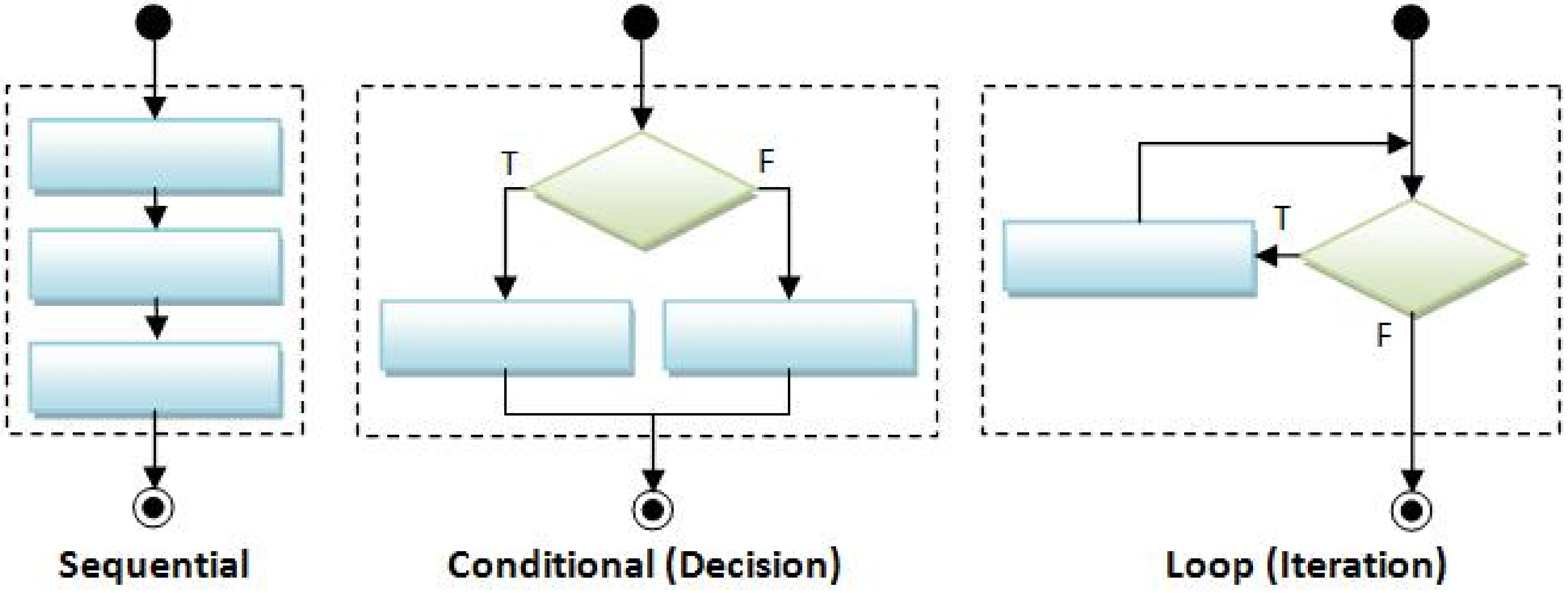
Structured Program

- **Structured Programming (modular programming):** Is a programming approach which is a disciplined and ordered approach to develop a program
- Code will execute the instruction by instruction one after the other
- The program may be divided into independent group of statements or modules
- It doesn't support the possibility of jumping from one instruction to some other with the help of any statement
- The instructions in this approach will be executed in a serial and structured manner
- Structured program uses single-entry and single-exit elements

Structured Program

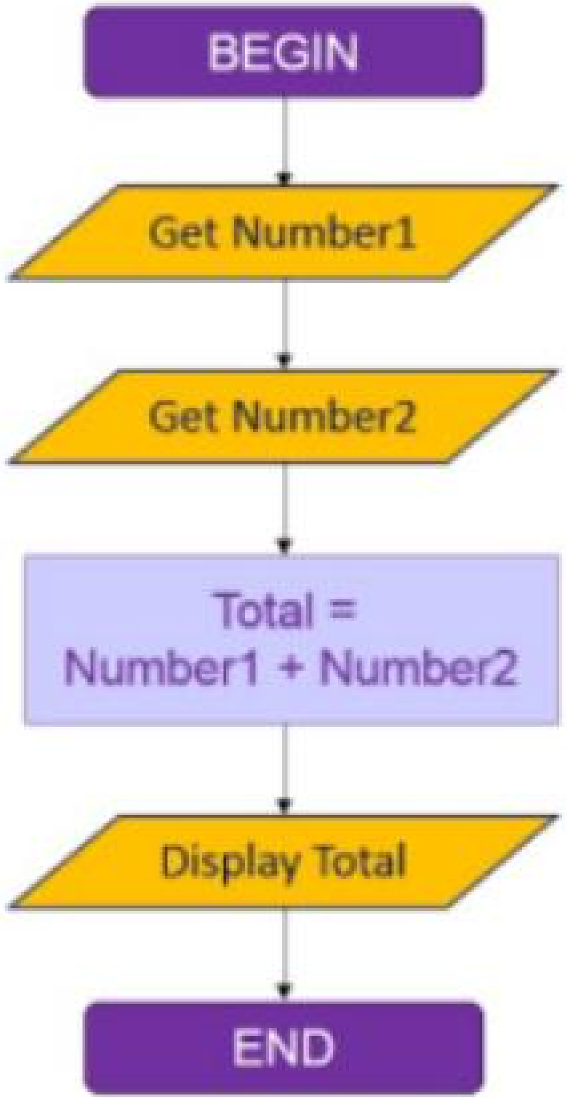
The structured program mainly consists of three types of elements:

- Sequence Statements - order of instructions
- Selection Statements – when there is a decision to execute
- Iteration Statements - process of repeating a set of instructions



Structured Program

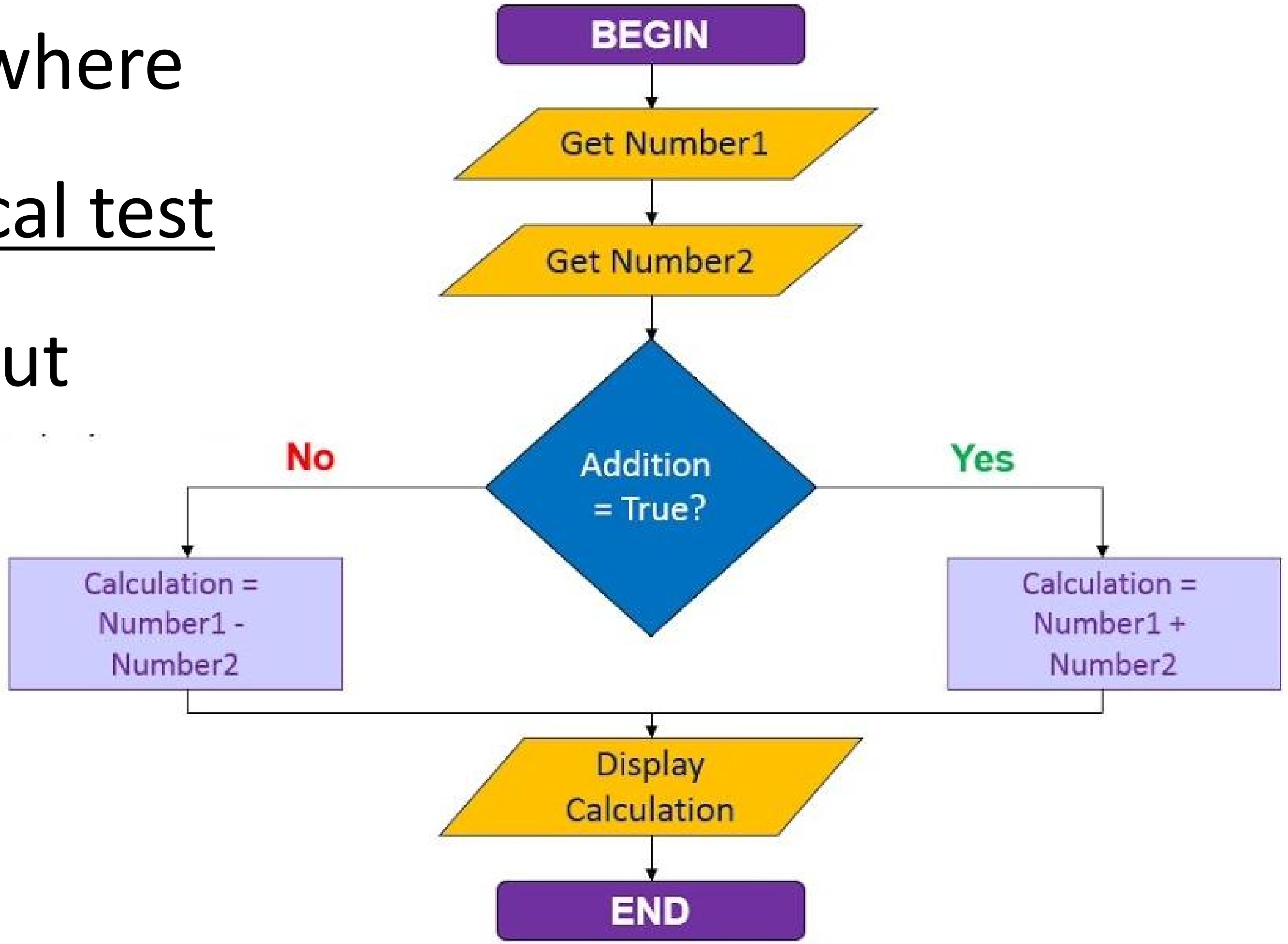
Sequential Structure: Follows a straight line execution mechanism in which sequence of statements are executed in a linear fashion



Structured Program

Selection Structure:

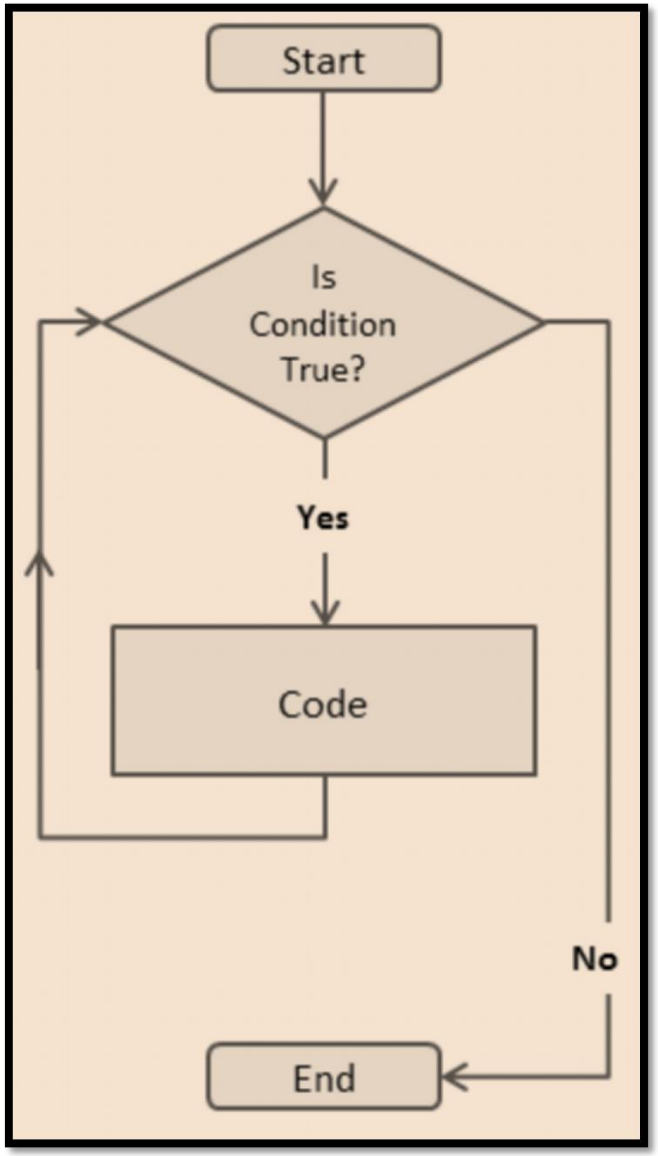
Multiple branching statement where there are many possible & logical test must be performed to get output



Structured Program

Iteration/Looping Structure:

- A condition will be evaluated at the starting of a loop.
- Looping action continues until the expression in loop is met



Structured Program: Advantages

- Easier to read and understand
- User friendly
- Easier to maintain or modify since individual modules can be corrected without changing entire program
- Mainly problem based instead of being machine based
- Development is easier as it requires less effort and time
- Easier to debug

Problem Solving

Problem solving by a computer system involve following steps:

1. Problem definition
2. Analysis & design
3. Coding
4. Running the program
5. Debugging
6. Testing
7. Documentation

Algorithm

- Is a step by step procedure to solve a problem
- Consists of number of statements/instructions designed to perform a specific task
- Statements must be precise & well defined

<u>Algorithm to compute area of a circle</u>

Step 1: Read **Radius**

Step 2: Compute **Area = 3.14 x radius x radius**

Step 3: Print **Area of Circle = Area**

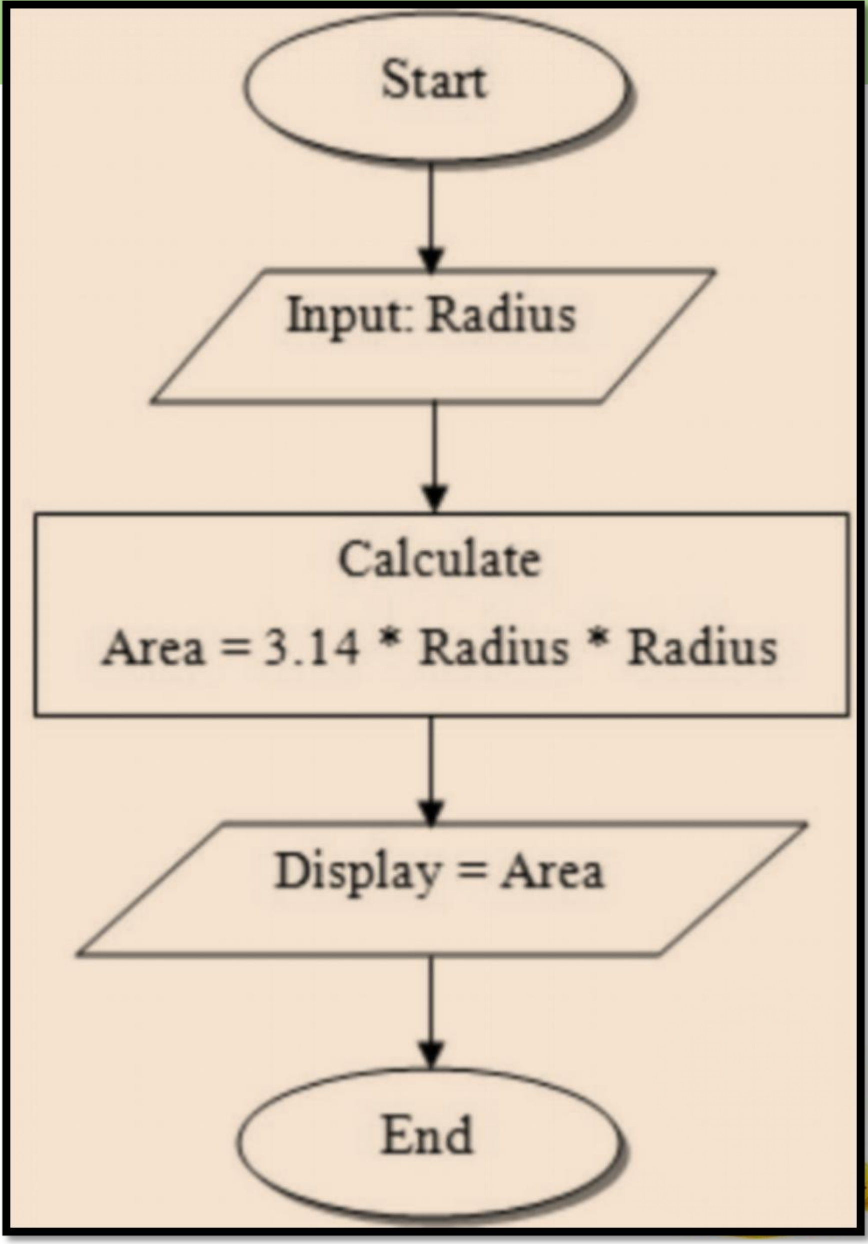
Step 4: Stop

Characteristics of Algorithm

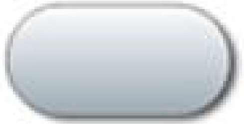
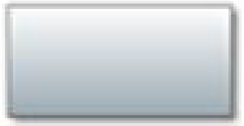
1. **Input:** May accept zero or more inputs
2. **Output:** Should produce at least one output
3. **Definiteness:** Each instruction must be clear, well-defined and precise
4. **Finiteness:** It should end after some time. Should not enter into an infinite loop
5. **Effectiveness:**
   - All the steps required to get to output must be feasible with the available resources
   - It should not contain any unnecessary and redundant steps which could make an algorithm ineffective
6. **Independent**
   - Statements must independent of any programming code

Flowchart

- Diagrammatic representation of an algorithm
- Easy way to understand the problem



Flowchart

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.

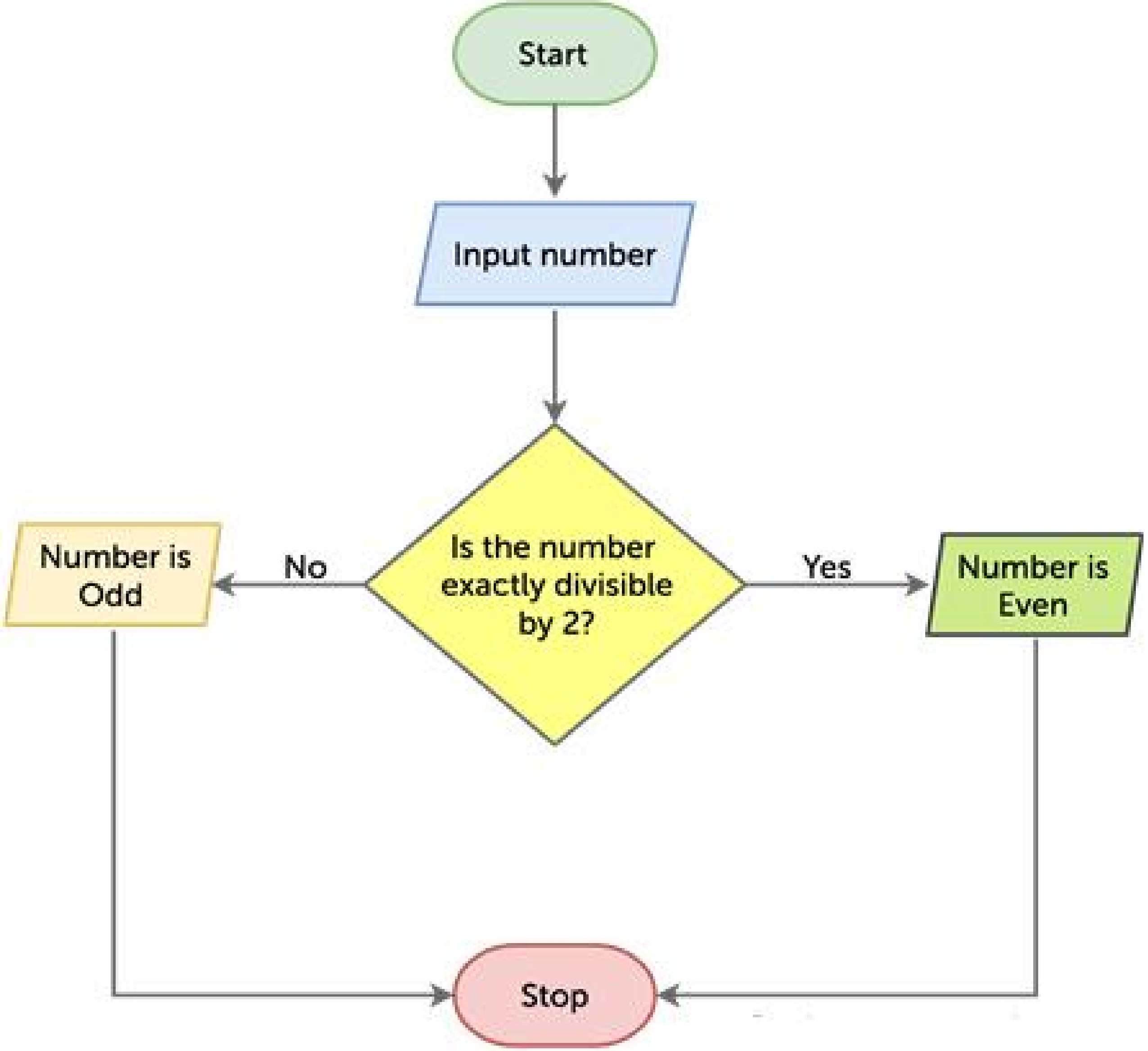
Pseudocode

- Pseudocode is an abstract form of a program
- Uses informal expressions to describe logic of program
- Program is represented in words/phrases, but syntax is not followed

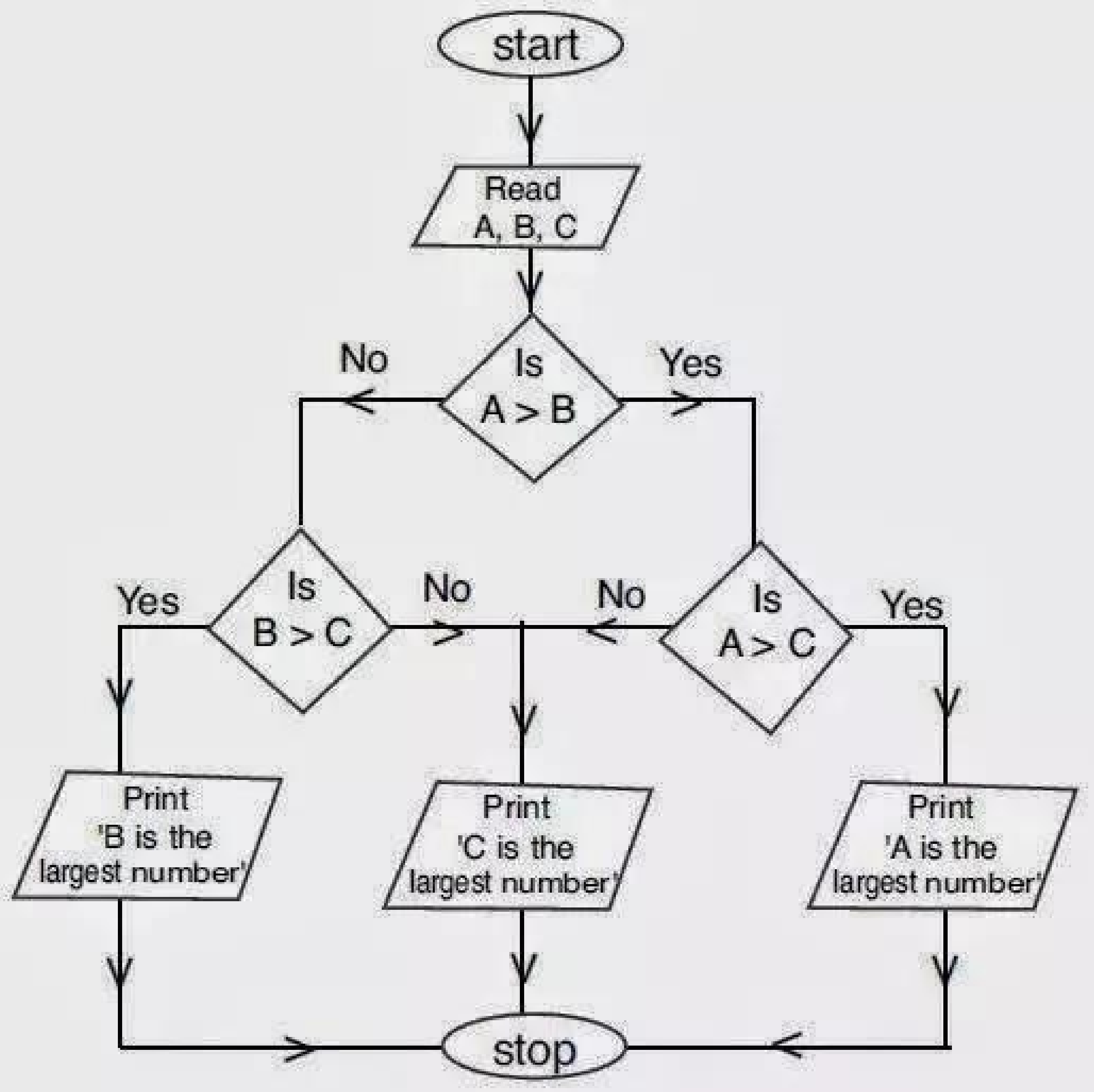
Pseudocode to check number is odd or even

1. Read n1
2. if n1/2 == 0
3. PRINT " Number is Even"
4. Else
5. PRINT " Number is Odd"
6. End if
7. End Program

ODD OR EVEN?



Start
Input number
Is the number exactly divisible by 2?
No
Yes
Number is Odd
Number is Even
Stop


start
Read
A, B, C
Is
A > B
No
Yes
Is
B > C
Yes
No
No
Is
A > C
Yes
Print
'B is the
largest number'
Print
'C is the
largest number'
Print
'A is the
largest number'
stop

ALGORITHM

1. To exchange values of two variables A & B
2. To find the roots of a quadratic equation, $ax^2+bx+c=0$

CHARACTER SET TOKENS

Module 2

Program Basics

Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types , Constants, Console IO Operations, printf and scanf

Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence

Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.(Simple programs covering control flow)

Contents

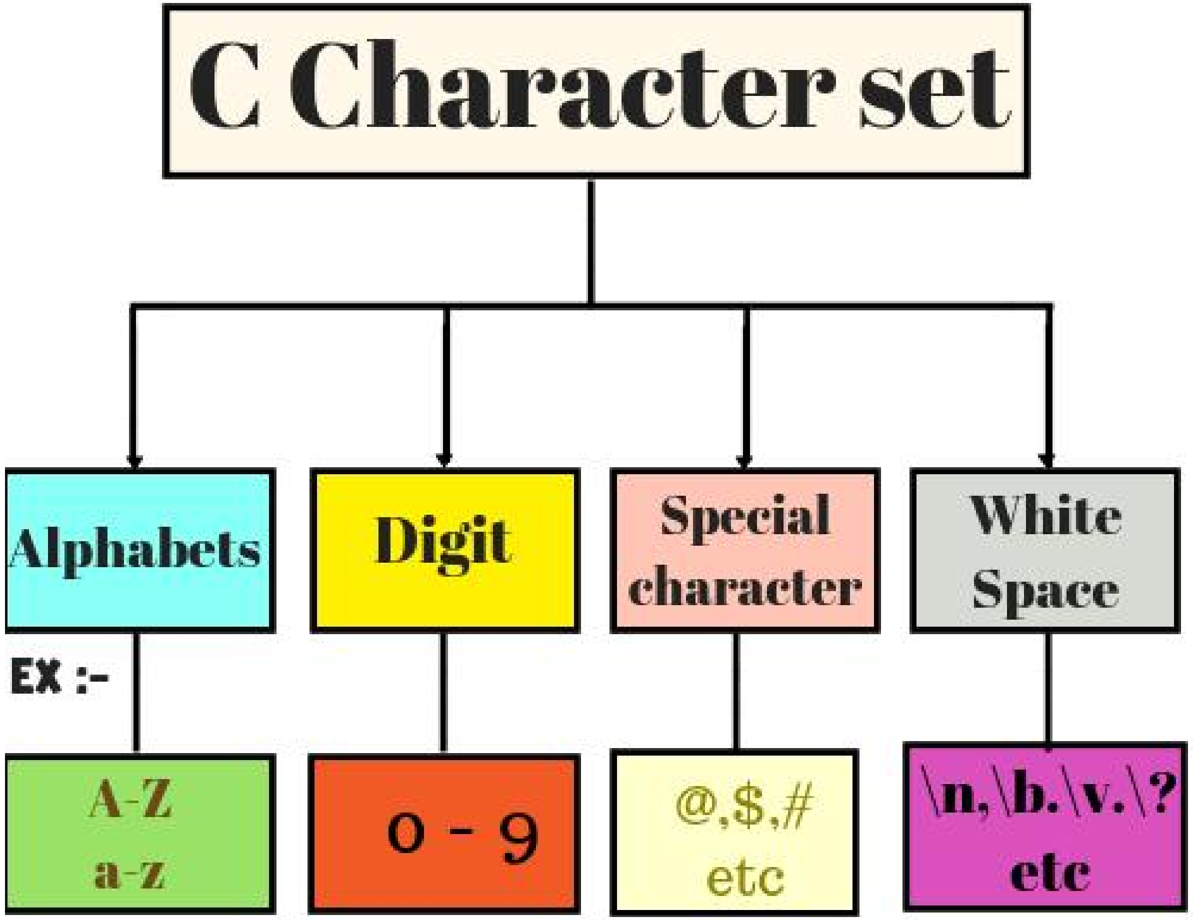
- Character set
- Tokens
- Keywords
- Identifiers
- Constants
- Data-types
- Variables
- Symbolic constants
- Library functions
- Header files

Character Set

Character set defines the **valid characters** that can be used **in source programs** or which can be interpreted when a program is running

Characters in C are grouped into:

1. Letters/Alphabets
2. Digits
3. Special character
4. White spaces



Character Set

<table>
<tr><th>Types</th><th>Character Set</th></tr>
<tr><td>Uppercase Alphabets</td><td>A, B, C, … Y, Z</td></tr>
<tr><td>Lowercase Alphabets</td><td>a, b, c, … y, z</td></tr>
<tr><td>Digits</td><td>0, 1, 2, 3, … 9</td></tr>
<tr><td>Special Symbols</td><td>~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ]
: ; " ' < > , . ? /</td></tr>
<tr><td>White spaces</td><td>Single space, tab, new line.</td></tr>
</table>

Tokens in C

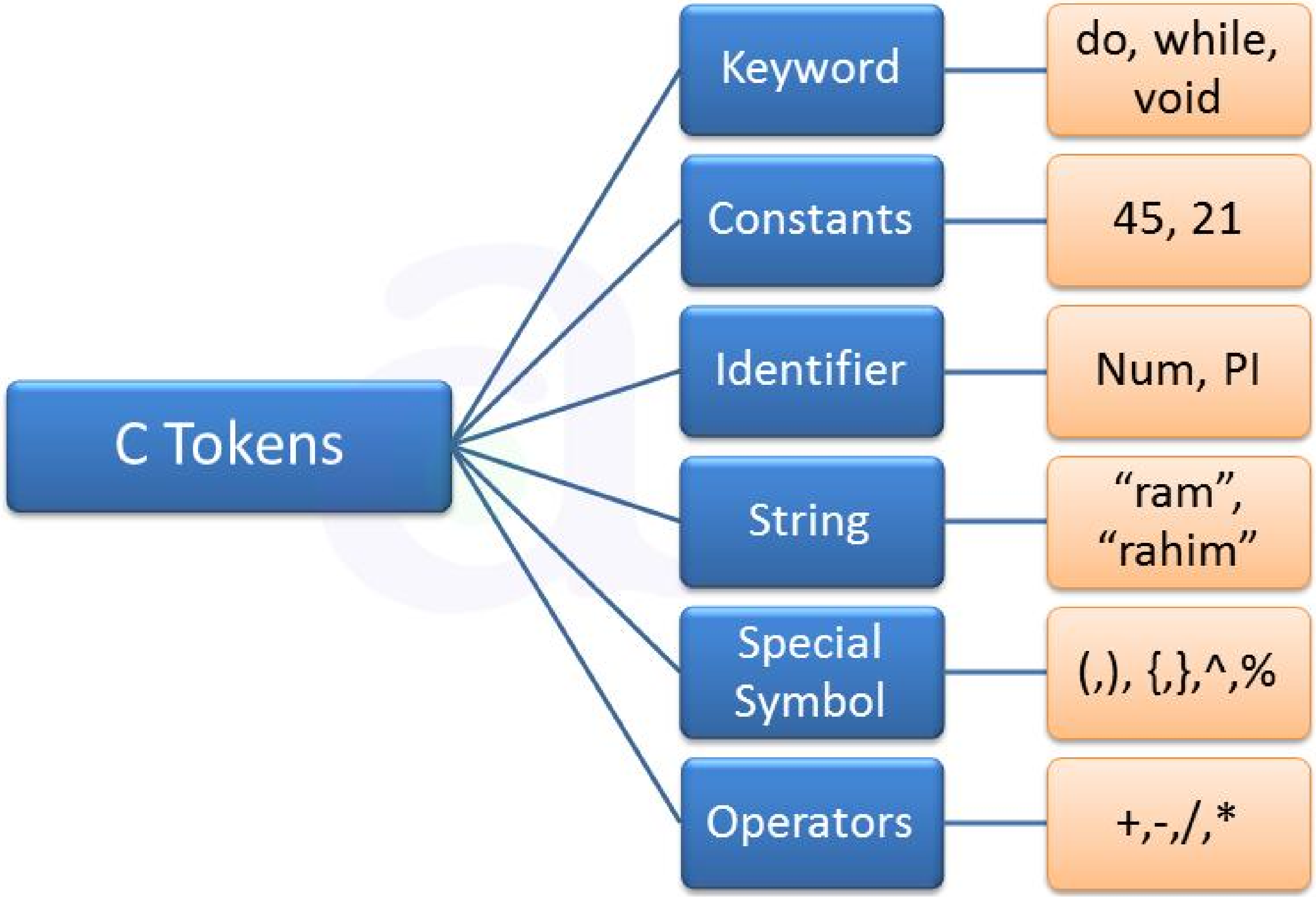
Tokens: The smallest individual units in a program

- It is each and every word and punctuation in C program
- The compiler breaks a program into tokens and proceeds to the various stages of the compilation

A token is divided mainly into six different types:

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

Tokens in C


C Tokens
Keyword
do, while, void
Constants
45, 21
Identifier
Num, PI
String
"ram", "rahim"
Special Symbol
(,), {,},^,%
Operators
+,-,/,*

Keywords & Identifiers

In 'C' every word can be either a keyword or an identifier

Keyword:

- Are reserved words with some predefined meaning
- Eg: int, if, else
- Assigned by compiler designer of the program
- All keywords have fixed meanings and these meanings cannot be changed

Keywords in C

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

Example:

int a = 10;

float b = 10.3;

Keywords & Identifiers

In 'C' every word can be either a keyword or an identifier

Identifier:

- Identifier refers **to name given to elements such as variables, functions, etc**.
- Are user defined and may consists of sequence of letters & digits
- Identifiers must be unique

Example:

float marks;

float average;

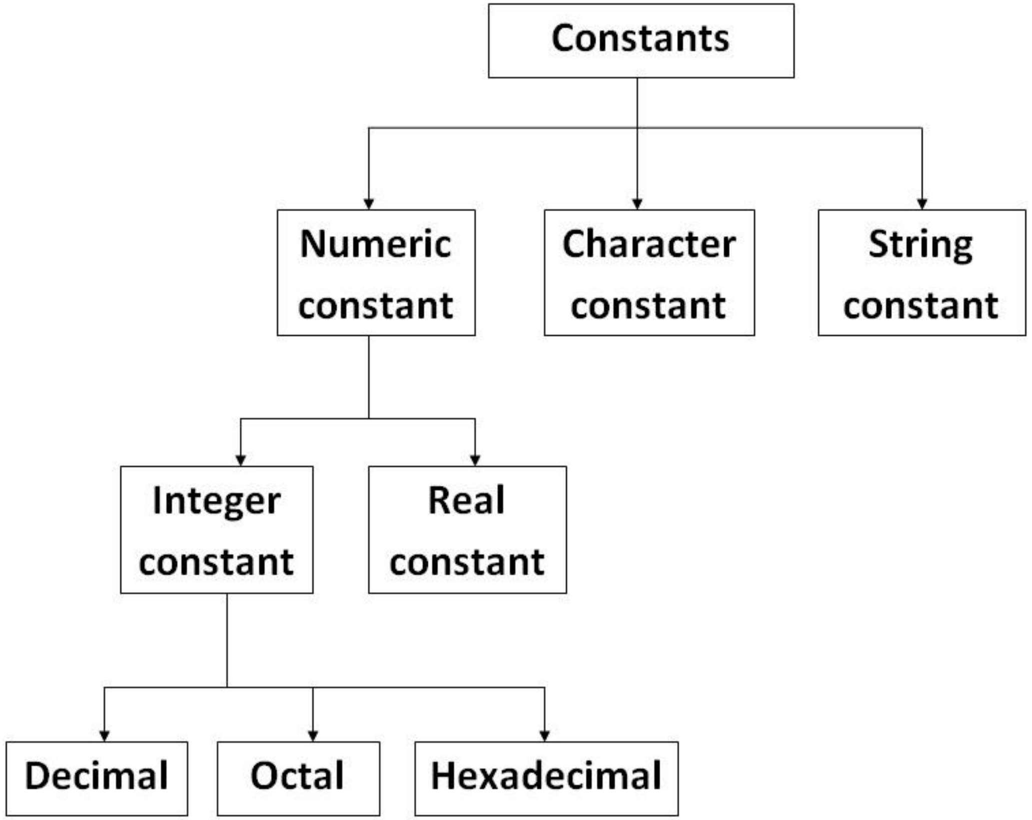
- float – keyword
- marks, average - identifiers

Rules for Identifies

1. Must begin with an alphabet (or underscore)
2. Must consist of only letters, digits or underscore
3. Only first 31 characters are significant
4. Cannot use a keyword
5. Must not contain white space
6. Upper case & lower case letters are distinct

Constants

Constants: Refer to fixed values that do not change during the execution of a program



Integer Constants

Integer Constants:

An *integer* constant refers to a sequence of digits

Three types of integers:

1. Decimal integer
2. Octal integer
3. Hexadecimal integer

Integer Constants

Decimal integers (base 10)

- Consist of a set of digits, 0 through 9, preceded by an optional – or + sign
- Eg: **123, -465**

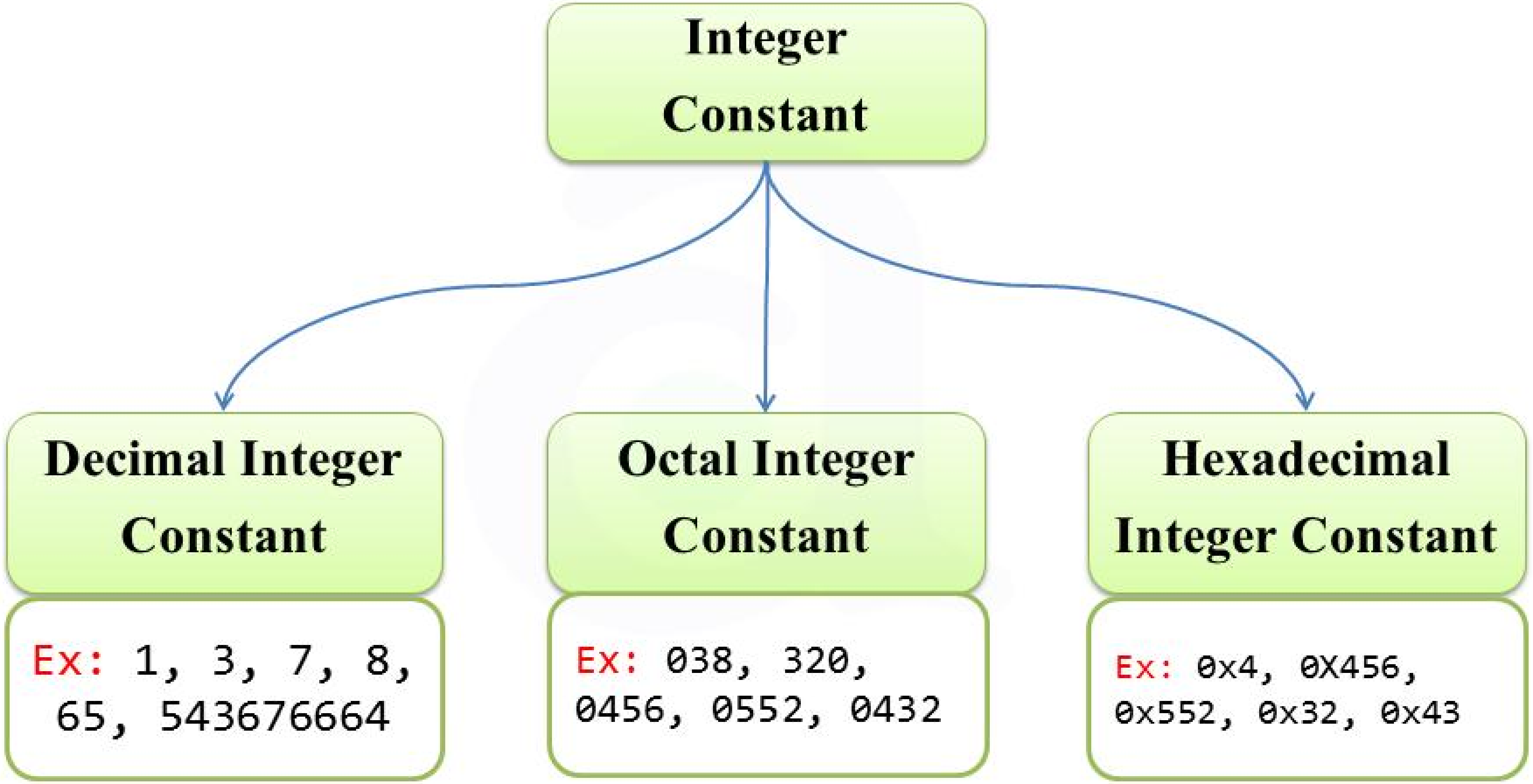
Octal integer (base 8)

- Consists of any combination of digits from the set 0 through 7 with a leading 0
- Eg: **037, 0777, 0435**

Hexadecimal integer (base 16)

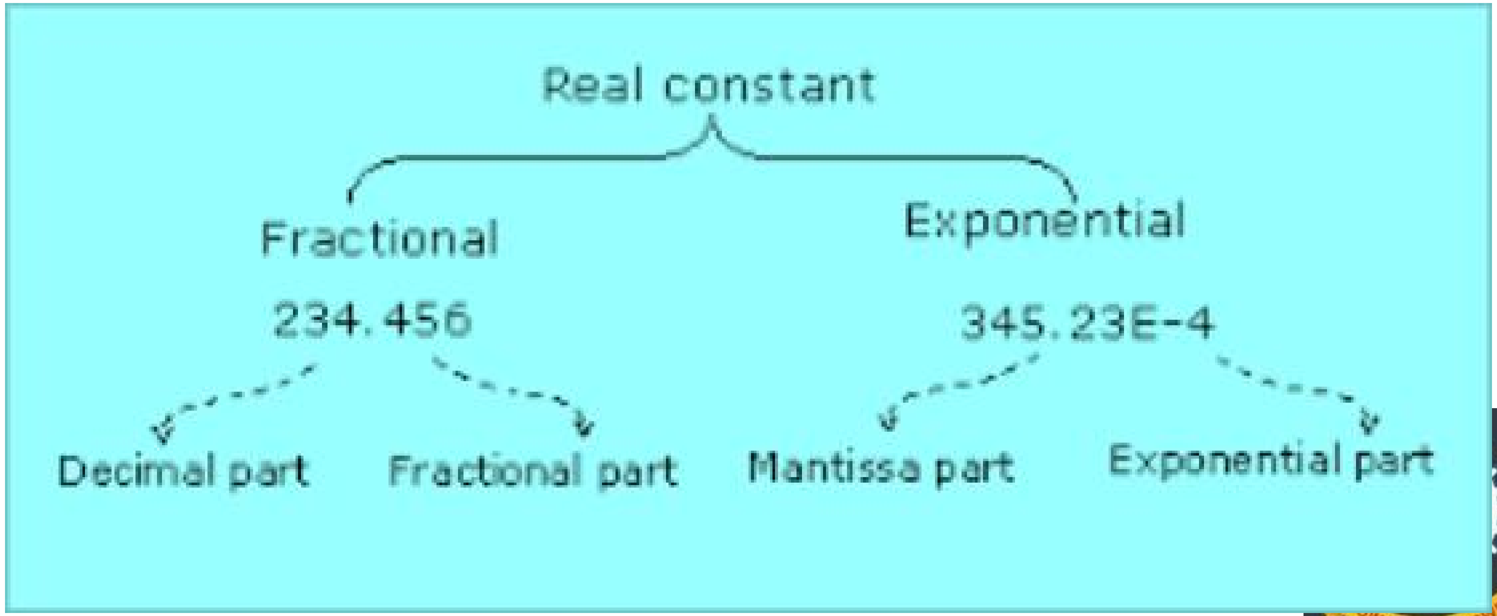
- Consists of sequence of digits preceded by 0x or 0X
- May also include alphabets A through F or a through f
- Eg: **0X3, 0x9f**

Integer Constants



Real/Floating Constants

- Numbers containing **fractional parts are called real or floating point** constant
- Can include integer part, decimal point, fractional part, exponential part
  1. Fractional or Normal form
  2. Exponential or Scientific form
- Eg: 315.3, 1.5e+5



Character Constant

- **Character constant** is a single character enclosed in a ‘single quotes’
- Eg: ‘5’, ‘A’
  - It is to be noted that the character ‘**5**' is not the same as **5**
- Each character constant has a corresponding ASCII value
- Eg: ‘A’ ASCII value 65
- Eg: ‘B’ ASCII value 66
- ASCII value is the numeric code of a particular character

String & Backlash Character Constant

- **String constant** is a set of characters enclosed in “double quotes”
- Eg: "HELLO”, "1234"

Backlash Character Constant

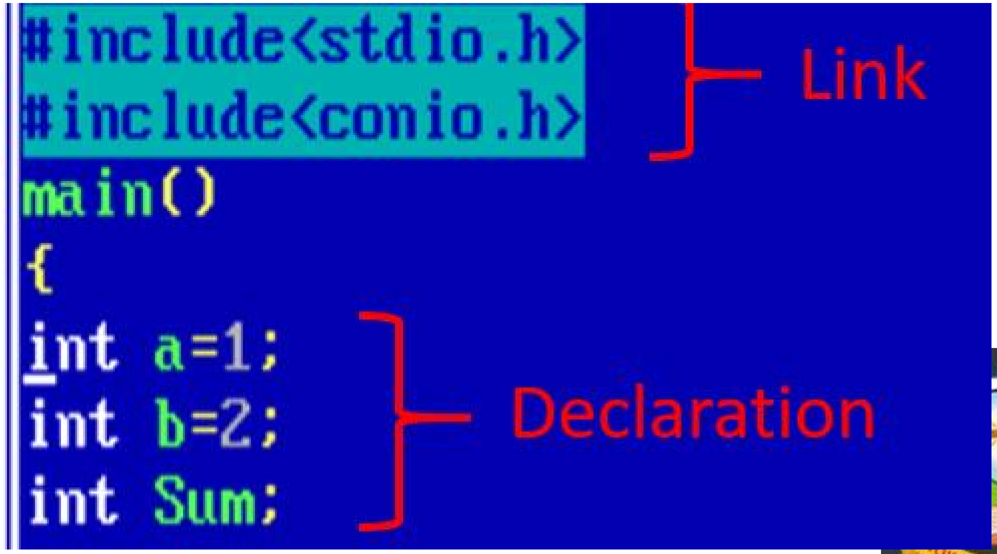
- C supports some character constants having a backslash (\) in front of it
- Backslash characters have a specific meaning which is known to the compiler
- They are also termed as "Escape Sequence“
- Eg:
  - \n – new line
  - \t – horizontal tab
  - \v – vertical tab

Backlash Character Constant

Constant	Meaning
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage Return
\t	Horizontal tab
\v	Vertical tab
\’	Single quote
\”	Double quote
\?	Question mark
\\	Backslash
\0	Null

Datatype

- Are used to store different types of data, Eg: alphabets, digits, strings, etc.
- A system used for declaring variables or functions of different types
- The type of a datatype determines
    - How much space it occupies in storage
    - How the bit pattern stored is interpreted
    - Which specific operations can be performed over it



Datatype

1.1. Primary data types

- Integer (int)
- Character (char)
- Floating Point (float)
- Double precision floating point (double)
- Void (void)

2. Derived data type

- Array
- Function
- Pointer
- Reference

3. User-defined data type

- Structure
- Union
- Enumeration

Primary Datatype

char:

It holds a single character and requires a single byte of memory in almost all compilers

int:

Holds integer quantities that do not contain a fractional part

float:

Holds decimal numbers with a fractional components

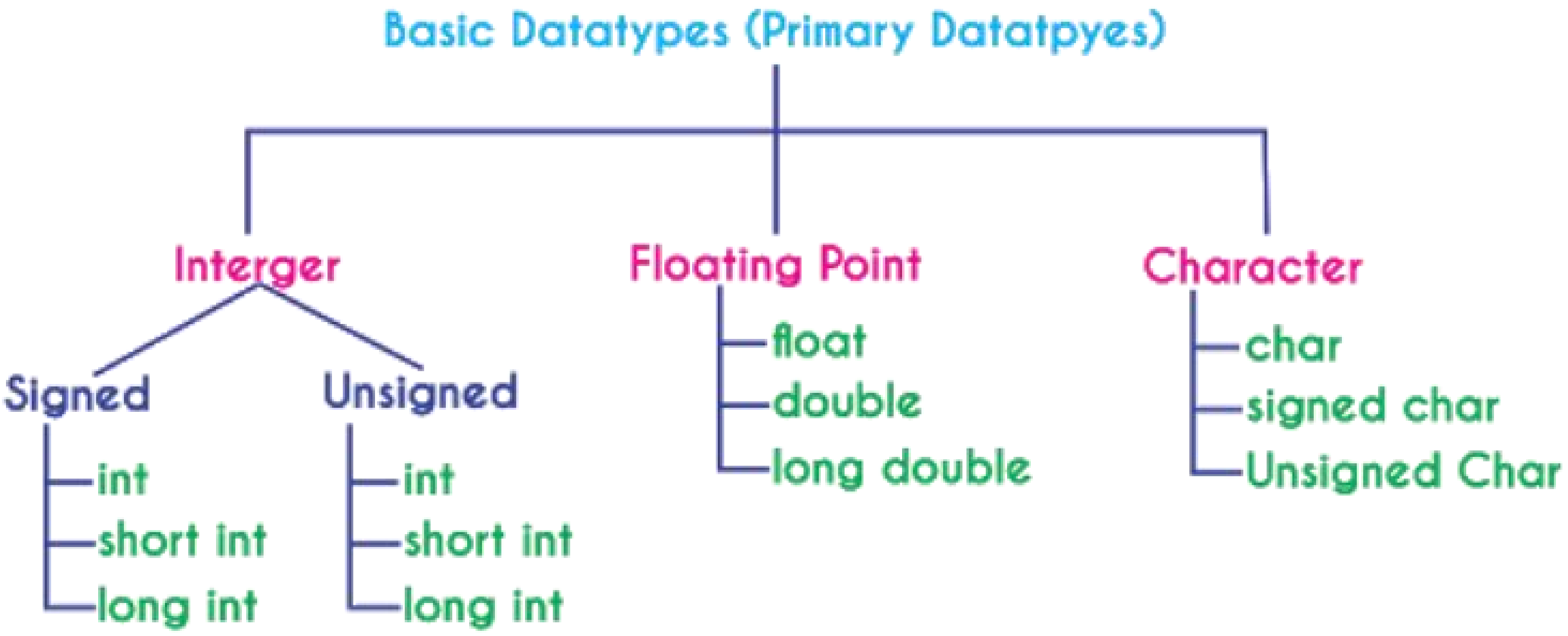
double:

It is used to store decimal numbers (floating point value) with double precision

void:

Used along with functions that do not return any value to the call

Primary Datatype



Primary Datatype

Data type	Size(bytes)	Range	Format String
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short	2	-32,768 to 32,767	%d
unsigned short	2	0 to 65535	%u
int	2	32,768 to 32,767	%d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to +2147483647	%ld
Unsinged long	4	0 to 4294967295	%lu
float	4	-3.4e-38 to +3.4e-38	%f
double	8	1.7 e-308 to 1.7 e+308	% lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf

Variable

- Variable is a valid identifier used for naming & declaring data items such as, integers, arrays, functions etc.
- Eg: age, height
- A variable may take different values at different times during execution
- Any variable must be declared properly before it is used in the program

Syntax for variable declaration:

data-type variable name;

OR

data-type variable name = value;

Example

int age, height;

int c = 3;

float average;

Variable Declaration

<u>Integer variable declaration</u>

```
int a;
```

<u>Character variable declaration</u>

```
char c;
```

<u>Floating variable declaration</u>

```
float f;
```

The declaration does two things

1. Tells the compiler the variable name
2. Specifies what type of data the variable will hold

Symbolic Constants

Symbolic constant:

- Is a way of defining a variable constant whose value cannot be changed
- If a numeric constant need to be used many times in a program, it will be difficult to modify or update its value
- Instead a symbolic constant can be defined to hold this constant

 # define ***symbolic-constant*** ***value of constant***

 # define PI 3.14

 # define NAME ANU

- Once the symbolic constant is initiated in the program it cannot be changed

Library Functions

- The standard library functions are built-in functions in C programming
- Each library function in C performs specific operation
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs
- A library function is accessed by simply writing the function name, followed by a list of arguments enclosed in paranthesis, which represent the information being passed to the function
- Eg: printf(…), sqrt (a)

Header files

- Header files are files with .h extensions and include several C function definitions
- Before using any predefined function in C, we have to include the header file in which that function resides
- Eg: stdio.h header file before using printf
- Header files contain function prototypes, datatype definitions for standard library functions
- Header files are included in the program with a #include pre-processor directive and file name in <> brackets
- stdio.h, math.h, ctype.h, string.h

CONTROL STATEMENTS

CONTROL STATEMENTS

- Statements in a program are executed in the order they appear in program
- This kind of execution is termed sequential execution
- In most programs it is necessary to manipulate this order to:
  - Select a set of statements from several alternatives
  - Skip certain statements based on some conditions and continue from another point
  - Repeat a set of statements until a specified condition is fulfilled
- For such processes CONTROL STATEMENTS must be used

CONTROL STATEMENTS

Control statements enable us to specify or direct the flow of program

i.e, The order in which the instructions in a program must be executed

There are three types of control statements in C:

1. Conditional control statements (if, switch etc.)
2. Loop control statements (while, for, etc.)
3. Jump statements (break, goto, continue)

Conditional Control Statements

- Used when required to check a condition
- Involves performing a logical test – results either TRUE or FALSE
- Depending on this the statements to be executed are determined
- Called **CONDITIONAL EXECUTION**
- Include statements such as:
  - *If*
  - *If-else*
  - *Switch*

if STATEMENTS

- The *if* statement is used to control the flow of execution of statements
- The different forms by which if statement can be implemented is:
  - Simple *If*
  - *If-else*
  - *Nested if*
  - *Else if ladder*

Simple *if* statement

- The general form of a simple *if* statement is:

```
If (condition to be checked)
{
Statement OR set of Statements;
}
```

- If the result of condition is true – statement immediately following if will be executed
- If the result of condition is false – control transfers to the next executable statement outside body of *if*

Simple *if* statement

```
/*Program to check whether even number*/
main()
{
        int n;
        scanf(“%d”, &n);                /* Enter a number*/
        if ((n%2)==0);
                printf(“even number”)
}
```

If – else Statements

- The *if…else* statement is an extension of the simple *if* statement
- The general form of a simple *if* statement is:

```
if (condition)
        {
        Statement 1;
        }
else
        {
        Statement 2;
        }
```

- If the result of condition is true – statement 1
- If the result of condition is false – statement 2

If – else Statements

- *If* Statement Or *else* Statement will be executed, not both
- In both cases, the control is transferred subsequently to the next *statement*

If – else Statements

```
/* program to check whether two numbers are equal*/
main()
{
        int M,N;
        printf(" Enter values of M and N \n");
        scanf("%d %d", &M, &N);
        If (M==N)
                printf("M and N are equal \n");
        else
                printf("M and N are not equal \n");
}
```

Nested *if statement*

- Used when a series of decisions are involved – <u>will have to select more than two alternatives</u>
- Hence, will have to use more than one *if- else* statement

Nested *if statement*

```
if (condition 1)
{
        if (condition 2)
        {
                statement 1;
        }
        else
        {
                statement 2
        }
}
else
{
statement 3
}
```

Nested *if statement*

```
/* program to find largest of 3 numbers*/
If (A>B)
  {
          If (A>C)
                   printf("A is the largest \n");
          else
                    printf("C is the largest \n");
   }
else
  {
          If ( B> C)
                   printf("B is the largest");
          else
                    printf("C is the largest");
  }
```

else if ladder

- The *else if* ladder is used when multipath decisions are involved
- A multipath decision is a chain of *if*s in which the statement associated with each *else* is an *if*

else if ladder

```
if (condition 1)
        statement 1;
else if (condition 2)
        statement 2;
else if (condition 3)
        statement 3;
……….
else
        default-statement;
Statement -x
```

- Conditions are evaluated from top to bottom
- When a true condition is found
  - Statement associated with it will be evaluated
  - Then control will be transferred to statement – x (skipping the rest)
- When all conditions are false
- Final else with default statement will be executed

Switch statement

- Switch statement is a multi branch decision structure
- Switch statement causes a particular group of statements to be chosen from several available groups
- The selection is based upon the current value of an expression which is included within the switch statement

Switch statement

```
switch ( expression )
{
   case value-1:
            statement-1;
            break;
   case value-2:
            statement-2;
            break;
………
………
default :
            default statement;
            break;
}
statement-x;
```

- Expression is an integer expression or characters
- Value 1, value 2 etc. are case labels and should be unique
- When switch is executed the value of expression will be compared with Value 1, value 2 etc.
- If a case whose value matches with value of expression is found, statements of that case is executed
- Break statement indicates end of each case
    - Transfer control to outside body of switch
- Default is optional
- Will be executed if value of expression does not match with any case values

switch statement

```
switch ( expression )
{
   case value-1:
          statement-1;
          break;
   case value-2:
          statement-2;
          break;
………
………
default :
          default statement;
          break;
}
statement-x;
```

```
char colour;
colour = getchar();
switch (colour)
{
case ‘R’:
          printf(“Colour is RED”);
          break;
case ‘G’:
          printf(“Colour is GREEN”);
          break;
case ‘Y’:
          printf(“Colour is YELLOW”);
          break;
Default:
          printf(“No colour given”);
}
```

PROGRAMMIN G IN C

CONTROL STATEMENTS

Control statements enable us to specify or direct the flow of program

i.e, The order in which the instructions in a program must be executed

There are three types of control statements in C:

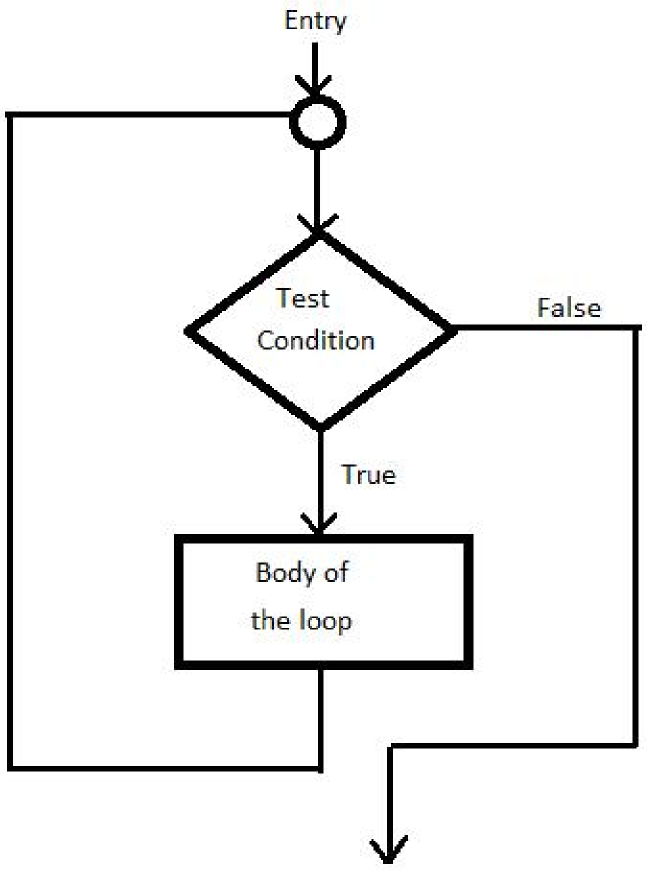
1. Conditional control statements (if, switch etc.)
2. Loop control statements (while, for, etc.)
3. Jump statements (break, goto, continue)

Decision Making & Looping

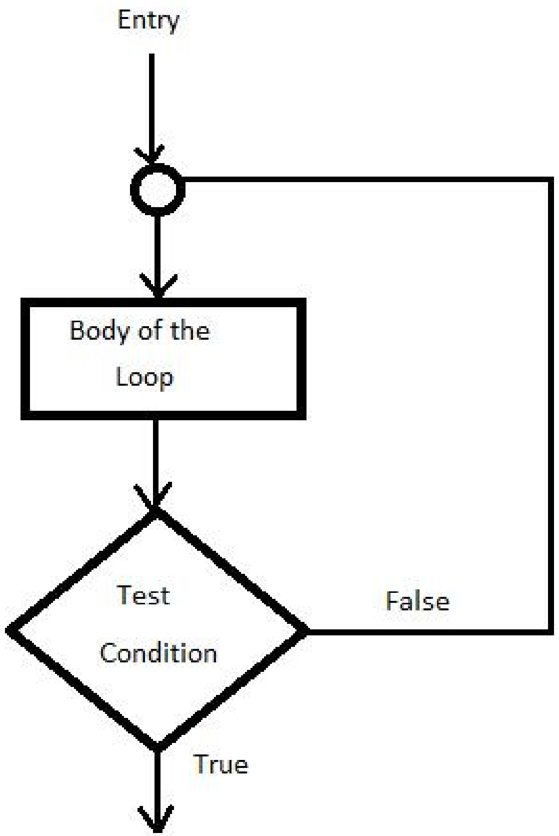
Loop Control Structures

- Looping is the programing technique in which group of statements will be executed repeatedly, until certain specified condition is met
- Also called ***repetitive* or *iterative*** control mechanism
- A *loop* consists of 2 parts
  - *Body* of loop
  - *Control statement*
- Control statements – perform a logical test resulting in a true or false result
- Body of loop – if logical test result is true, statements in the body of loop will be executed. Otherwise loop will be terminated

Loop Control Structures



Entry Controlled Loop



Exit Controlled Loop

Loop Control Structures

The control statements can either be placed before or after the body of loop

- **Entry controlled loop** or **pre-test loop**
  - Control statements placed before body of loop
  - Conditions are tested before the start of the loop execution
- **Exit controlled loop** or **post-test loop**
  - Control statements placed after body of loop
  - The test is performed at the end of the body of the loop and therefore the statements are executed unconditionally for the first time

Loop Control Structures

Steps of efficient loop control systems:

for (n = 1; n <= 10; n ++)

1. **Initialisation**
   - To set the initial value for loop counter
   - May be an increment loop counter or decrement loop counter
2. **Decision**
   - An appropriate test condition to determine whether to execute loop or not
3. **Updation**
   - Incrementing or decrementing the counter value

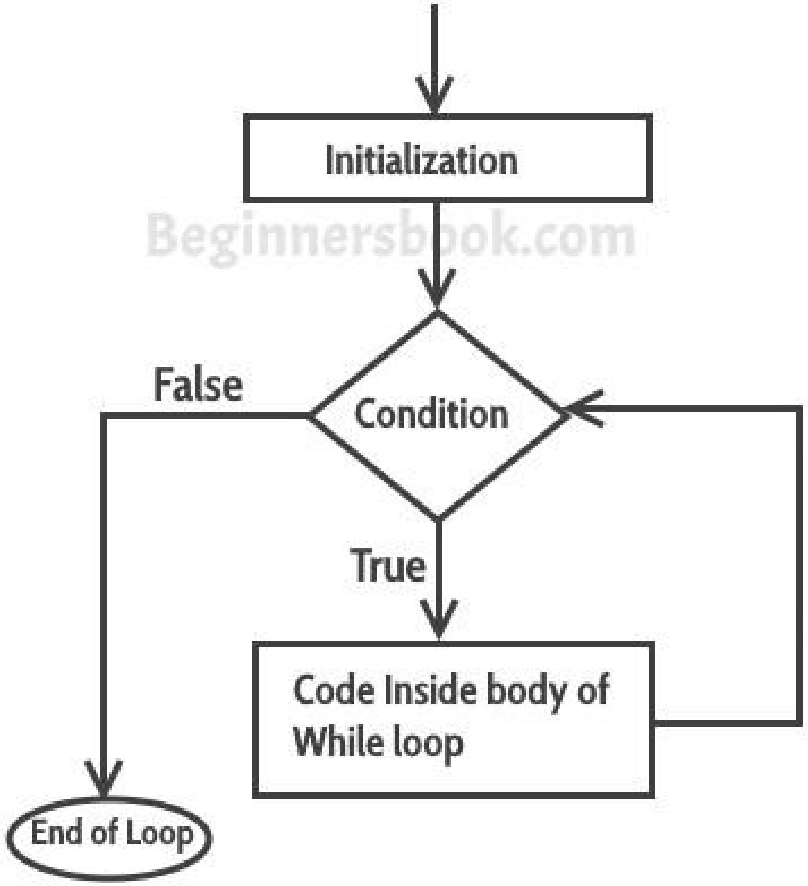
while loop statement

Syntax:

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

- *while* is an entry-controlled loop statement
- The test-condition is evaluated first
- If the condition is true - then the associated statements will be executed
- If the condition is false – control comes out of the loop and continues with the next executable statement
- After every repetition of loop, condition is checkd to decide whether to continue the loop or not

while loop statement




```
//Program to print sum of first 5 numbers
void main ()
{
int n= 1; int sum = 0;
while (n<=5)
{
   sum = sum + n;
   n=n + 1;
}        // end of while
printf("sum = %d", sum);
}        // end of main
```

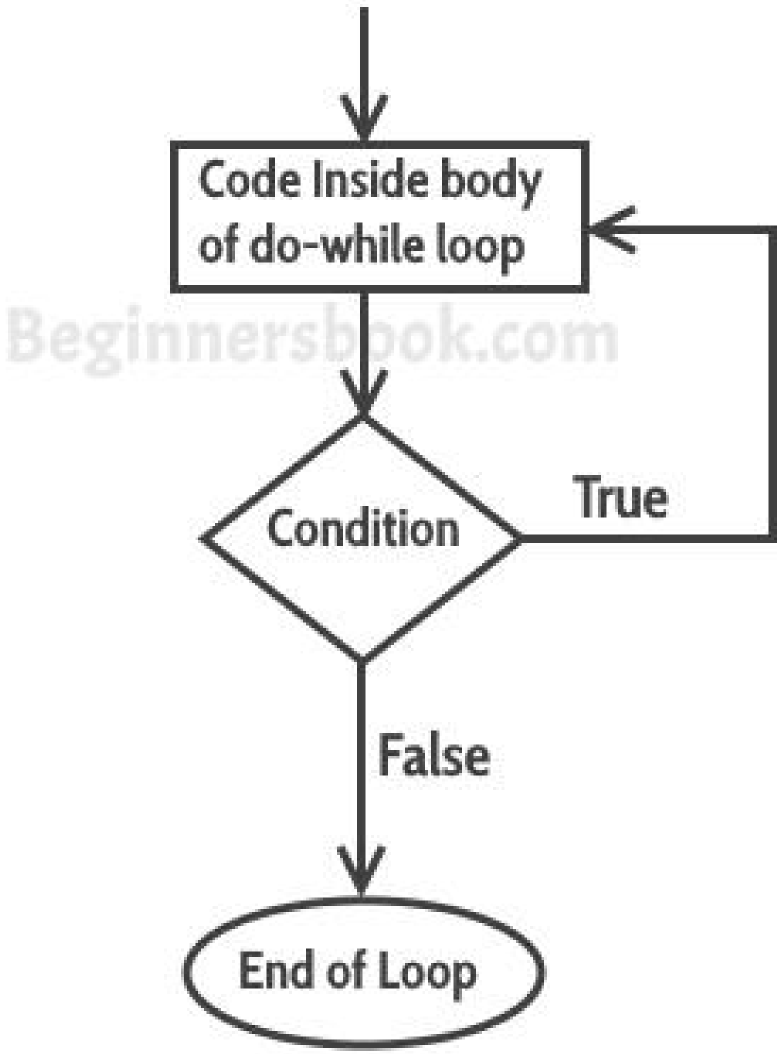
do-while statement

Syntax:

```
do
{

    statements;

} while (condition)
```

- *do-while* is an exit controlled loop statement
- Test for repetition is made at the end of each pass
- The process continues as long as the condition is true
- When the condition becomes false, the loop will be terminated and the control goes to the next statement outside while
- The statements of the loop is always executed at least once

do-while statement



do-while statement

// Program to print sum of first 5 numbers

```
void main ()
{
int n= 1; int sum = 0;
do
{
    sum = sum + n;
    n++;
} while (n<=5)         // end of while
printf("sum = %d", sum);
}                      // end of main
```

for loop statement

Syntax:

```
for ( initialization ; test-condition ; increment )
{
  statements;
}
```

- The *for* loop is an entry-controlled loop

for loop

```
for ( n = 1; n <= 10; n ++)
{
  statements;
}
```

<u>The execution of for loop statement is as follows:</u>

1. Initialization of the control variables is done first, using assignment statements
2. The value of the control variable is tested using the test condition
   - If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop
3. When the body of the loop is executed, the control is transferred back to the *for* statement
4. Now the control variable is incremented /decremented and the new value of the control variable is again tested to see whether it satisfies the loop condition
5. Process continues till the value of control variable fails the test-condition

for loop statement

```
// Program to print sum of first 5 numbers
void main ()
{
int n, sum = 0;
for (n=1; n<=5; n++)
        {
          sum = sum + n;
        }
printf(“sum = %d”, sum);
}                          // end of main
```

Jump Statements

goto Statement

- **goto:** Is an unconditional control statement, to transfer control of one program from one point to another
- Is a branching statement and require a *label*
  - To identify the place where control must be transferred
- Label can be any variable name and to be followed by a colon

syntax

```
goto label;
```

Example:

```
goto END;
..................
..................
..................
END:
```

goto Statement

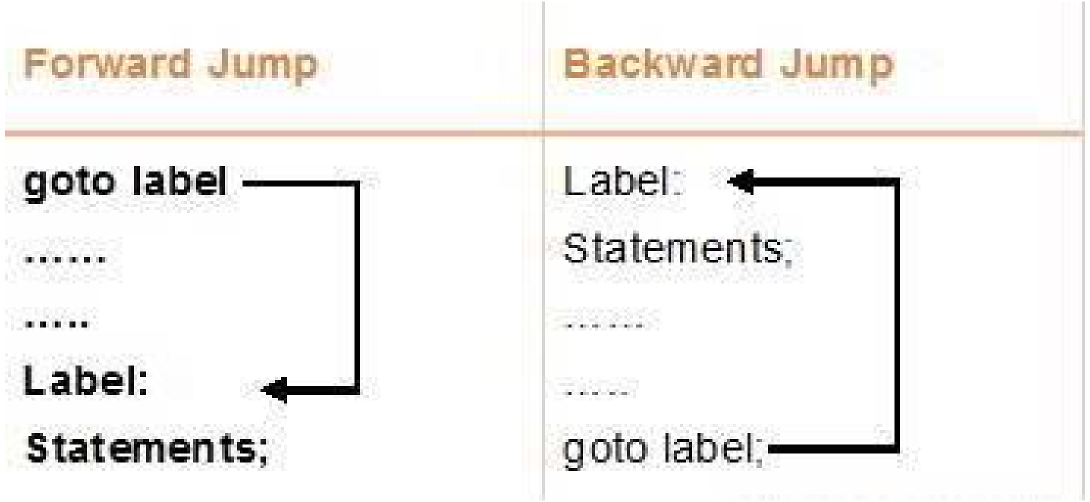
- Label can be anywhere in the program either before OR after *goto* statement

Forward jump

- If the label is placed after *goto* statement, some statement will be skipped

Backward jump

- If the label is placed before *goto* statement, a loop will be formed and some statements will be executed repeatedly



break statement

Syntax: **break;**

- break statement is a jumping statement which allows control of program to shift to another location
- break statement can be used to terminate a loop and exit from a particular switch case label
- When used with any looping statements, control comes out of the corresponding loop and continues with the next statement
- When used with a nested loop, control comes out of that loop only not from the complete nesting

continue statement

Syntax: **continue;**

- continue statement is a jumping statement which allows control of program to shift to another location
- Is used to skip certain statements inside a loop and to start next iteration of loop
- i.e. The control does not come out of the loop, instead skips the remaining statements and is transferred to the beginning of the loop

Nested loops

Nested loops

- When one loop is placed inside the other, it is termed nested loop
- Inner and outer loops need not be of same type
- One loop must be completely inside the other, with no overlapping
- Each loop must have different index variables

Nested loops

Nested *for* loop

```
For (n=1,n<10;n++)
{
   for (m=1;m<10;m++)
         {
            statements;
         }
}
```

Nested loops

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

*

* *

* * *

* * * *

* * * * *

- Outer loop – number of rows

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

- Outer loop – number of rows

i					
1	*				
2	*	*			
3	*	*	*		
4	*	*	*	*	
5	*	*	*	*	*

- Outer loop – number of rows
- Inner loop – number of ‘*’ in one row

i					
1	*				
2	*	*			
3	*	*	*		
4	*	*	*	*	
5	*	*	*	*	*

- Outer loop – number of rows
- Inner loop – number of ‘*’ in one row

j

i	1	2	3	4	5
1	*				
2	*	*			
3	*	*	*		
4	*	*	*	*	
5	*	*	*	*	*

- Outer loop – number of rows
- Inner loop – number of ‘*’ in one row

```
void main ()
{
int i,j;
for (i=1; i<=5; i++)
{
  for (j=1; j<=i; j++)
   {
        printf("*");
   }
printf("\n");
}
```

i \ j	1	2	3	4	5
1	*				
2	*	*			
3	*	*	*		
4	*	*	*	*	
5	*	*	*	*	*

i =1 , *j* loop – one *

i =2 , *j* loop – two **

i =3 , *j* loop – three ***

Nested loops

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

- Outer loop – number of rows
- Inner loop – number of ‘*’ in one row

```
void main ()
{
int i,j;
for (i=1; i<=5; i++)
{
  for (j=1; j<=i; j++)
   {
        printf(“%d”,j);
   }
printf(“\n”);
}
```

j

i	1	2	3	4	5
1	**1**				
2	**1**	**2**			
3	**1**	**2**	**3**		
4	**1**	**2**	**3**	**4**	
5	**1**	**2**	**3**	**4**	**5**

i =1 , *j* loop – one *

i =2 , *j* loop – two **

i =3 , *j* loop – three ***

INPUT OUTPUT FUNCTIONS

Input Output Functions

Input/Output function in C

- Various input/output functions are available in C language
- They are classified into two broad categories

1. **Console Input/Output Functions** – These functions receive input from keyboard and write them on the VDU (Visual Display Unit) - Monitor
2. **File Input/Output Functions** – These functions perform input/output operations on a hard disk or other storage devices

Input/Output function in C

Console I/O functions further classified into

1. Formatted Input/Output Functions
2. Unformatted Input/Output Functions

Type	Input function	Output function
Formatted	scanf()	printf()
Unformatted	getchar()	putchar()
	gets()	puts()

Formatted I/O functions enable the user to specify the type of data, and the way it should be read in or written out. This user specification is not possible with unformatted I/O functions

Input/Output function in C

Input function	Output function	Purpose
scanf()	printf()	Input & Output of single characters, strings, numerical values
getchar()	putchar()	Input & Output of single characters
gets()	puts()	Input & Output of strings

getchar() & putchar()

getchar() function:

Reads a character from keyboard

Header file – <stdio.h>

Example:

```
main()
{
        char letter;
        letter = getchar ()
}
```

char letter:

Declares that ‘letter’ is a character-type variable

letter = getchar ():

Reads a single character entered from the keyboard and assign to ‘letter’

getchar() & putchar()

putchar() function:

Prints a character on the screen

Header file – <stdio.h>

Example:

```
main()
{
        char letter;
        letter = getchar ();
        putchar (letter);
}
```

getchar() & putchar()

- The *getchar* function can also be used to read multi-character strings, by reading one character at a time within a loop
- The *putchar* function can be used with loops to output a string.

gets() & puts()

- The **gets()** function reads strings from keyboard until ENTER key is pressed
- The **puts ()** function prints a string of characters on the screen

```
#include<stdio.h>
void main()
{
        char message [20];
        printf(" Enter message:\n");
        gets(message);
        puts(message);
}
```

printf() & scanf() functions

scanf() function

- **scanf()** function can be used to enter any combination of numerical values or characters or strings

Syntax

```
scanf("control string", arguments list);
scanf("control string", arg1, arg2, arg3);
```

Control string:

- Specifies the field format in which the data is to be entered
- It is a sequence of one or more character groups
- Starts with a % and followed by a conversion character indicating the type of data

Arguments list:

- Specify the address of locations where the data is stored

scanf() function

```
scanf("control string", arg1, arg2, arg3);
```

control string

- Starts with a %
- Conversion character indicating the type of data

% s - string

% d - integer

% f - floating point

```
scanf ("%d", &number);
```

scanf() function

Example

```
main()
{
        char a[20];
        int b;
        float c;
        scanf (“ %s  % d  %f  ”, a, &b, &c );
}
```

% s - string

% d - integer

% f - floating point

- Every variable name other than an array must be preceded by an ampersand (&)
- It represents the memory address

Common Format Specifiers

Format Specifier	Data Type
%d	int
%f	float
%c	char
%u	short unsigned
%lu	long unsigned
%ld	long signed
%lf	double

Common Format Specifiers

Data Type		Format
Integer	Integer	%d
	Short	%d
	Short unsigned	%u
	Long	%ld
	Long assigned	%lu
	Hexadecimal	%x
	Long hexadecimal	%lx
	Octal	%O (letter 0)
	long octal	%lo
Real	float,double	%f, %lf, %g
Character		%c
String		%s

scanf() function

Field width can be specified

scanf("%3d", &number);

- Can input integer with 3 digits

scanf("%3d, %5d", &A, &B);

- First 3 consecutive digits will be assigned to A
- Next 5 consecutive digits will be assigned to B

printf() function

- printf() function can be used to output numerical values, single characters and strings
- printf () function moves data from the computer's memory to the output device

Syntax

printf("control string", arguments list);

printf() function

Syntax

printf("control string", arguments list);

Example:

printf ("Hello World");

printf ("%d", number);

printf ("%f %d ", x1, x2); when x1 is float type and x2 is interger type

printf ("Sum of two numbers = %d", sum);

printf() function

a = 123.456	printf ("%f", a);	123.456000
a = 123.456	printf ("%.3f", a);	123.456
a = 123.456	printf ("%.1f", a);	123.5
b = 678	printf ("%d", b);	678
b = 678	printf ("%5d", b);	678
b = 678	printf ("%05d", b);	00678

OPERATORS EXPRESSIONS

Contents

- Operators & types
- Expressions
- Statements
- Precedence of operators

OPERATORS & EXPRESSIONS

OPERATORS

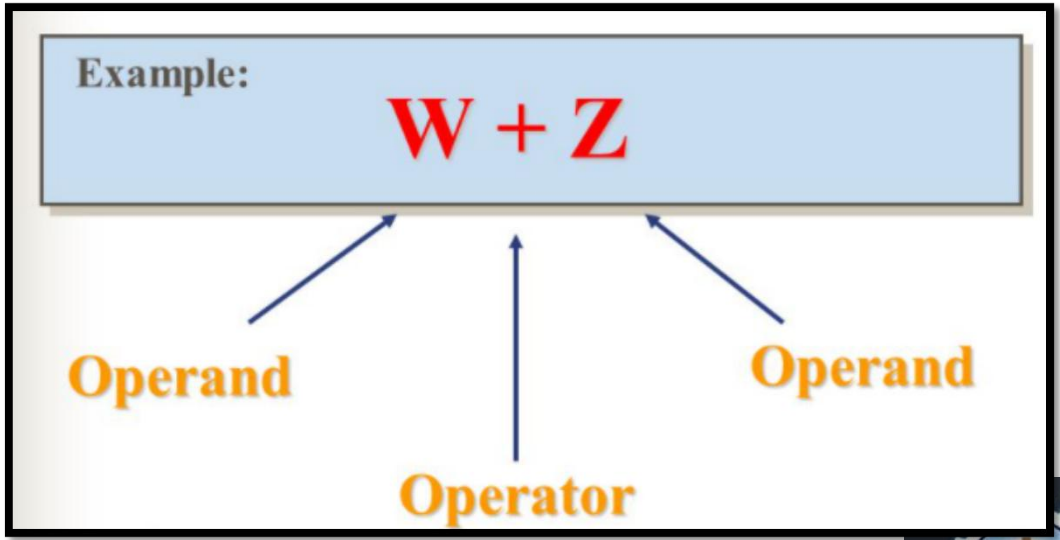
Operator: is a symbol that tells the compiler to perform specific mathematical or logical functions

- Arithmetic Operators
- Unary operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

- An arithmetic operator performs mathematical operations on numerical values
- The operands must be numerical values

Operator	Meaning of Operator
+	addition
-	subtraction
*	Multiplication
/	Division
%	Module operator



UNARY OPERATORS

- Operators that act upon a single operand to produce a new value

1. ++ : Increment operator
   - Increments the value of operand by 1
   - A++ means A=A+1

2. -- : Decrement operator
   - Decrements the value of operand by 1
   - A-- means A=A-1

UNARY OPERATORS

Increment/Decrement Operators — Let us assume X is a variable

Operator	Expression	Description
++	++X	Pre-increment
	X++	Post-increment
--	--X	Pre-decrement
	X--	Post-decrement

UNARY OPERATORS

- Pre-fix increment ++A
- Pre-fix decrement - -A

First the increment/decrement action will be done and then the value of operand (A) will be used

- Post-fix increment A++
- Post-fix decrement A - -

First the value of operand (A) will be used and then the increment/decrement action will be done

Relational Operators

Relational Operators		Suppose X and Y are two variables
Operator	**Expression**	**Description**
<	X < Y	X is less than Y
<=	X <= Y	X is less than or equal to Y
>	X > Y	X is greater than Y
>=	X >= Y	X is greater than or equal to Y
==	X == Y	X is equal to Y
!=	X != Y	X is not equal to Y

Equality Operator (== and !=)

Logical Operators

Logical Operators

Suppose X and Y are two variables

	Operator	Expression	Description
1	&&	X && Y	Logical AND Operator
2	||	X || Y	Logical OR Operator
3	!	!X	Logical NOT Operator

Relational & Logical Operators

- Statements which use <u>relational or logical operations yield either true or false as outcome</u>
- Relational or logical operations return
  - 0 for false
  - 1 for true

int a=7 **int b=5** **float f = 6.5** **char k ='w'**	**Expression**	**Interpretation**	**Value**
	a < b	false	0
	f > 4	true	1
	k ==119	true	1
	k != 's'	true	1
	(a+f) <= 10	false	0

Bitwise Operator

- All data items are stored in a computer memory as a sequence of bits (0 and 1)
- There are several applications which need manipulation of these bits
- To perform this C provides six bitwise operations
- They work with int & char type
- Cannot work with float

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

Bitwise Operator - Results

Bitwise **AND** **&** :

- 1 if both bits are 1
- Otherwise 0

Bitwise **OR** **|** is:

- 1 if one of bits is 1
- Otherwise 0

Bitwise **XOR** **^**:

- 1 if both bits are different
- Otherwise 0

Bitwise **complement** **~**:

- Reverses state of each bit

Bitwise Operator

- Assume A = 60 and B = 13 in binary format, they will be as follows –
- A = 0011 1100
- B = 0000 1101

A	0	0	1	1	1	1	0	0
B	0	0	0	0	1	1	0	1
A&B	0	0	0	0	1	1	0	0
A|B	0	0	1	1	1	1	0	1
A^B	0	0	1	1	0	0	0	1

Bitwise Operator

Bitwise complement ~ operator

A	0	0	1	1	1	1	0	0
~A	1	1	0	0	0	0	1	1

Bitwise Operator

Left shift operator:

- Represented by <<
- Two operands, Eg: A<<2
- Bits of first operand will be shifted to left by number specified by second operand

A	0	0	1	1	1	1	0	0
A<<2	1	1	1	1	0	0		
	1	**1**	**1**	**1**	**0**	**0**	**0**	**0**

Bitwise Operator

<u>Right shift operator:</u>

- Represented by >>
- Two operands, Eg: A>>2
- Bits of <u>first operand will be shifted</u> to right <u>by number specified by second operand</u>

A	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**
A>>2			0	0	1	1	1	1
	0	**0**	**0**	**0**	**1**	**1**	**1**	**1**

Assignment Operator

- Assignment operators are used to assign value of an expression to one or more identifiers
- Eg: x = y + 3;
- = Assignment operator

Assignment Operator

Shorthand Operator	Example	Equivalent expression
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/ =	x /= 2;	x = x / 2;
% =	x %= 2;	x = x % 2;
<<=	x <<= 2;	x = x << 2;
>>=	x >>= 2;	x = x >> 2;
&=	x &= 2;	x = x & 2;
|=	x |= 2;	x = x | 2;
^=	x ^= 2;	x = x ^ 2;

Conditional Operator

- **Conditional operators** are used to test the relationship between variables
- Used along with relational operators
- An expression that makes use of the conditional operator is called a conditional expression
- Takes 3 inputs as operands. Evaluate 2nd or 3rd expression based on the first conditional expression

<expression 1> ? < expression 2 > : < expression 3>

- **Expression 1** is evaluated first
- If expression 1 is true, then expression 2 is evaluated
- If expression 1 is false, then expression 3 is evaluated

Conditional Operator

Example:

a = 3

b=5

c = a<b ? < a > : < b>

If a<b, c= a

If a>b, c=b

C = a = 3

sizeof Operator

- This operator returns the size of operand in number of bytes
- x = sizeof (sum)

Expressions

Expression:

- Is a sequence of operands and operators that reduces to a single value
- It can be any valid combination of constants, variables or other data elements
- x=y
- C=a+b

Statements

Statement:

- Is an instruction to computer to carryout some action
- Three different statements in C

1. Expression statements
2. Compound statements
3. Control statements

Statements

1. Expression Statement:

- Consists of an expression followed by a semicolon
- Eg:
  - a=b+c;
  - printf("value of sum = %f", sum);
- Include one or more operands and operator
- While executing the expression statement, the expression is evaluated

Statements

2. Compound Statement:

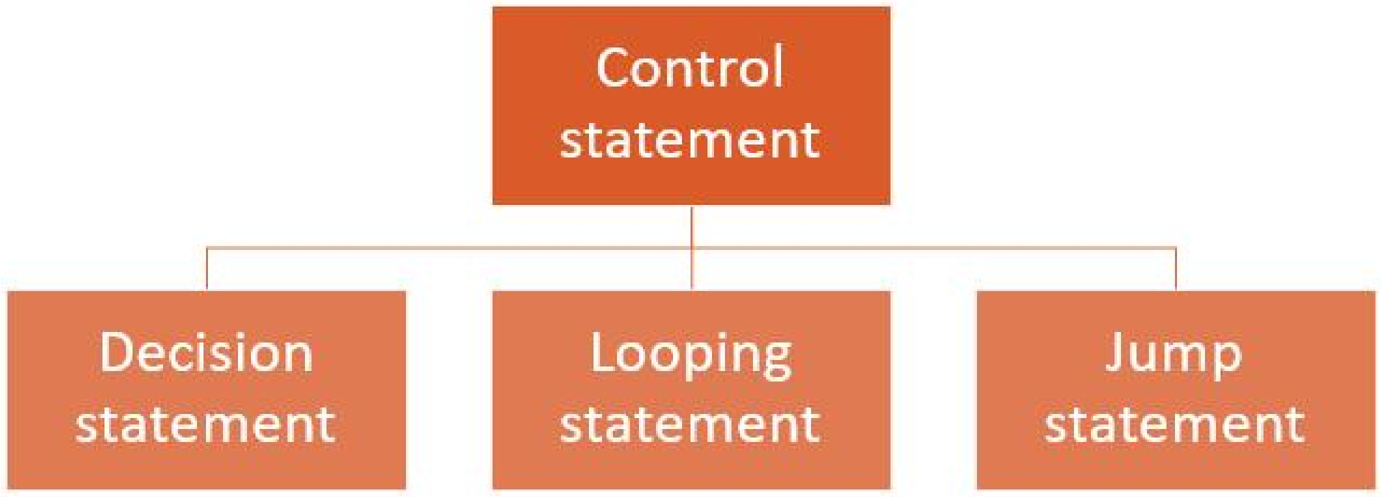
- Consists of several individual statements enclosed within a pair of curly braces {…}
- Eg:

```
{
int a=5;
float b=4.5;
float sum  = a+b;
}
```

Statements

3. Control statements

- Are used to control the flow of execution of a program
- Logical tests, loops & branches are the main features provided by control statements



Precedence of Operator

Precedence

- The operators in C are grouped in hierarchically as per their precedence (order of evaluation)
- Operations with higher precedence are carried out before other operations with lower precedence

Associativity

- The order in which consecutive operations within the same precedence group are carried out is known as associativity

Precedence & Associativity

Operator category	Operators	Precedence	Associativity
Parentheses, braces	( ), [ ]	1	L to R
Unary operators	-, ++, --, !, ~, &	2	R to L
Multiplicative operators	*, /, %	3	L to R
Additive operators	+, -	4	L to R
Shift operators	<<, >>	5	L to R
Relational operators	<, <=, >, <=	6	L to R
Equality operators	==, !=	7	L to R
Bitwise operators	&, ^, |	8	L to R
Logical operators	&&, ||	9	L to R
Conditional operators	?, :	10	R to L
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	11	R to L
Comma operator	,	12	L to R

STRUCTURE OF C

Module 2

Program Basics

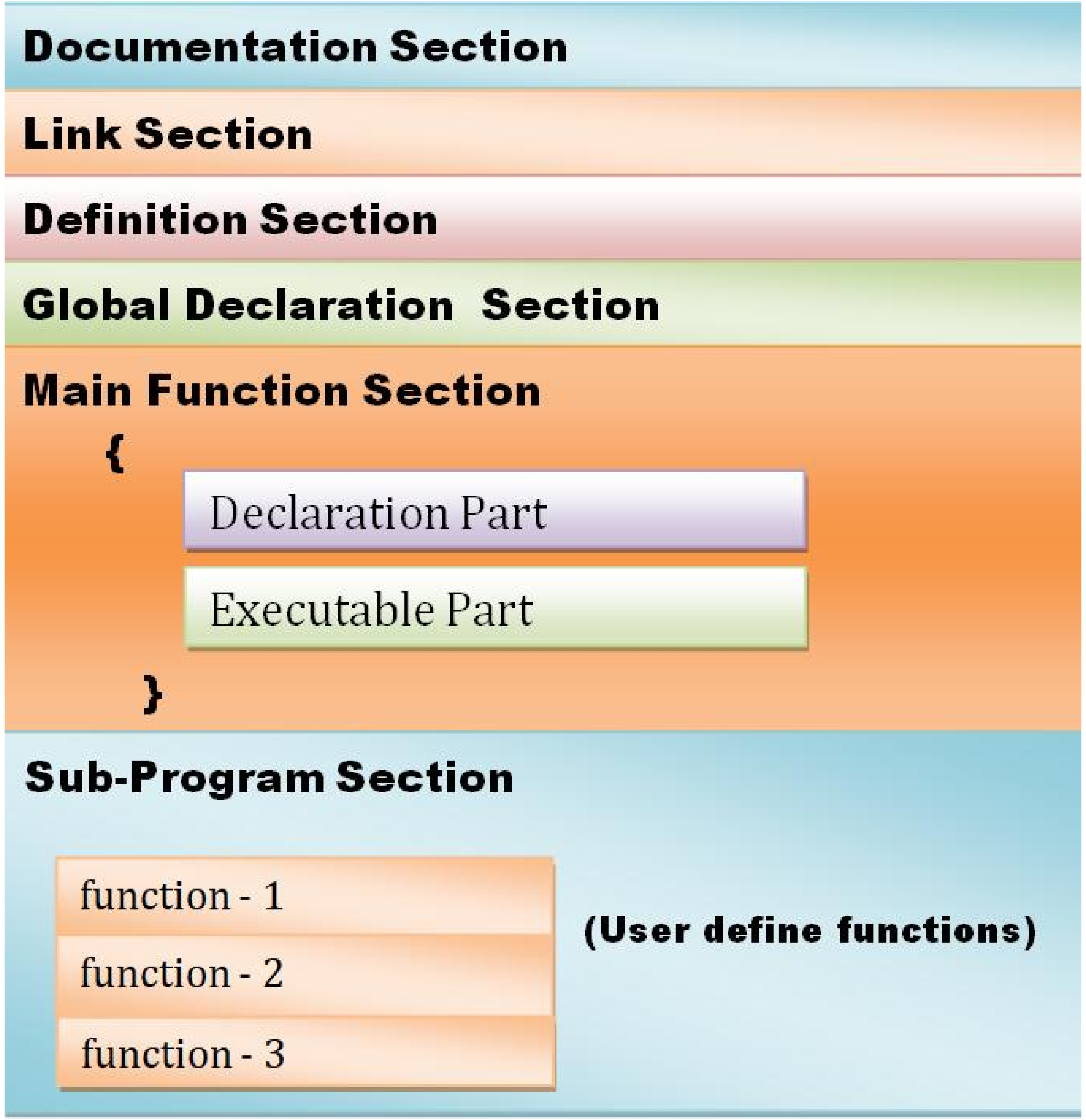
Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types , Constants, Console IO Operations, printf and scanf

Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence

Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.(Simple programs covering control flow)

Structure of C Program

The basic format of a C program:

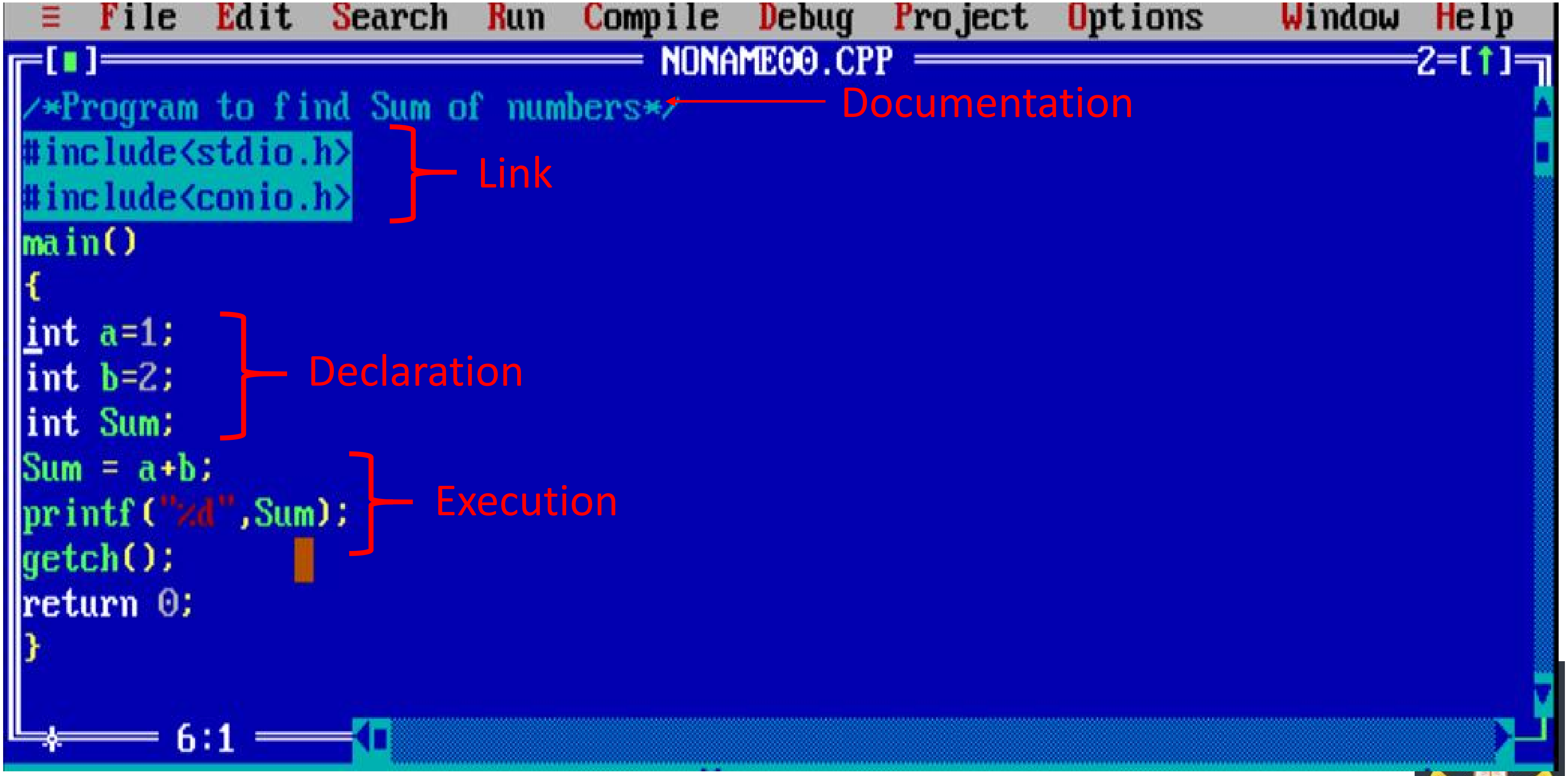


Structure of C Program

Documentation section:

- Is the part of the program where the programmer gives the details associated with the program
- Usually the name of the program, the details of the author, time of coding etc.
- These comment lines include the details the user would like to use later

```
/*File name: print hello
Date: 09/06/2021
Description: A program to display hello*/
```


≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.CPP
/*Program to find Sum of numbers*/
Documentation
#include<stdio.h>
#include<conio.h>
Link
main()
{
int a=1;
int b=2;
int Sum;
Declaration
Sum = a+b;
printf("%d",Sum);
getch();
Execution
return 0;
}
6:1

Structure of C Program

Link Section

- This part of the code is used to **declare all the header files** that will be used in the program
- Link section provides instructions to **compiler to link functions** to system library
- Eg: **#include<stdio.h>** : For standard input & output functions

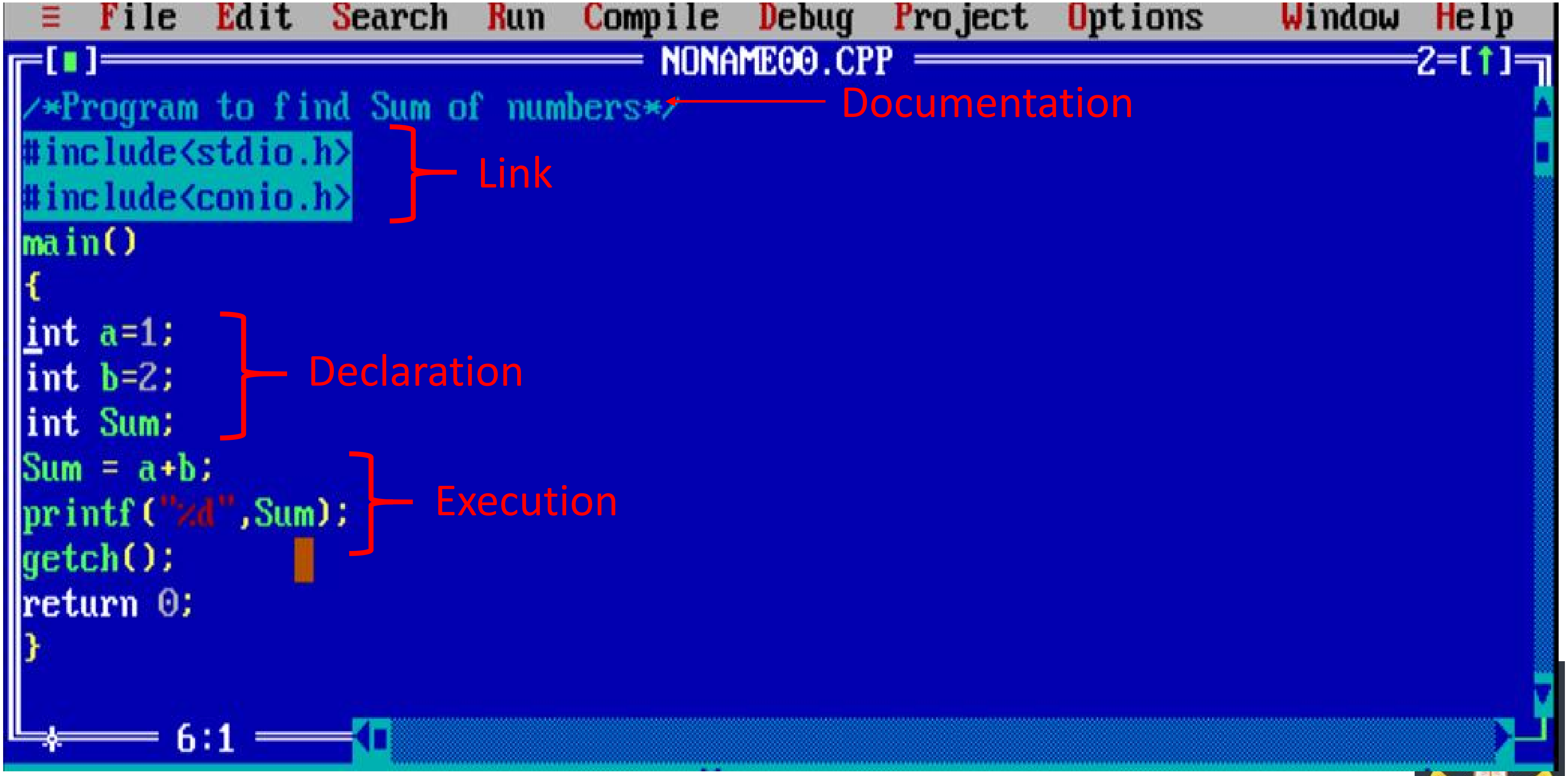
```
/* Program to print 'Hello'*/

main ()
{
      printf (" Hello");
}
```

Structure of C Program

Link Section

- This part of the code is used to **declare all the header files** that will be used in the program
- Link section provides instructions to **compiler to link functions** to system library
- Eg: **#include<stdio.h>** : For standard input & output functions

```
/* Program to print ‘Hello’*/
#include<stdio.h>
main ()
{
      printf (“ Hello”);
}
```


≡ File Edit Search Run Compile Debug Project Options Window Help
[■] NONAME00.CPP 2 [↑]
/*Program to find Sum of numbers*/
Documentation
#include<stdio.h>
#include<conio.h>
Link
main()
{
int a=1;
int b=2;
int Sum;
Declaration
Sum = a+b;
printf("%d",Sum);
getch();
Execution
return 0;
}
6:1

Structure of C Program

Definition Section

Section which defines values of all symbolic constants

Eg: **#define PI=3.14, #define X = 20**

Structure of C Program

Global declaration Section

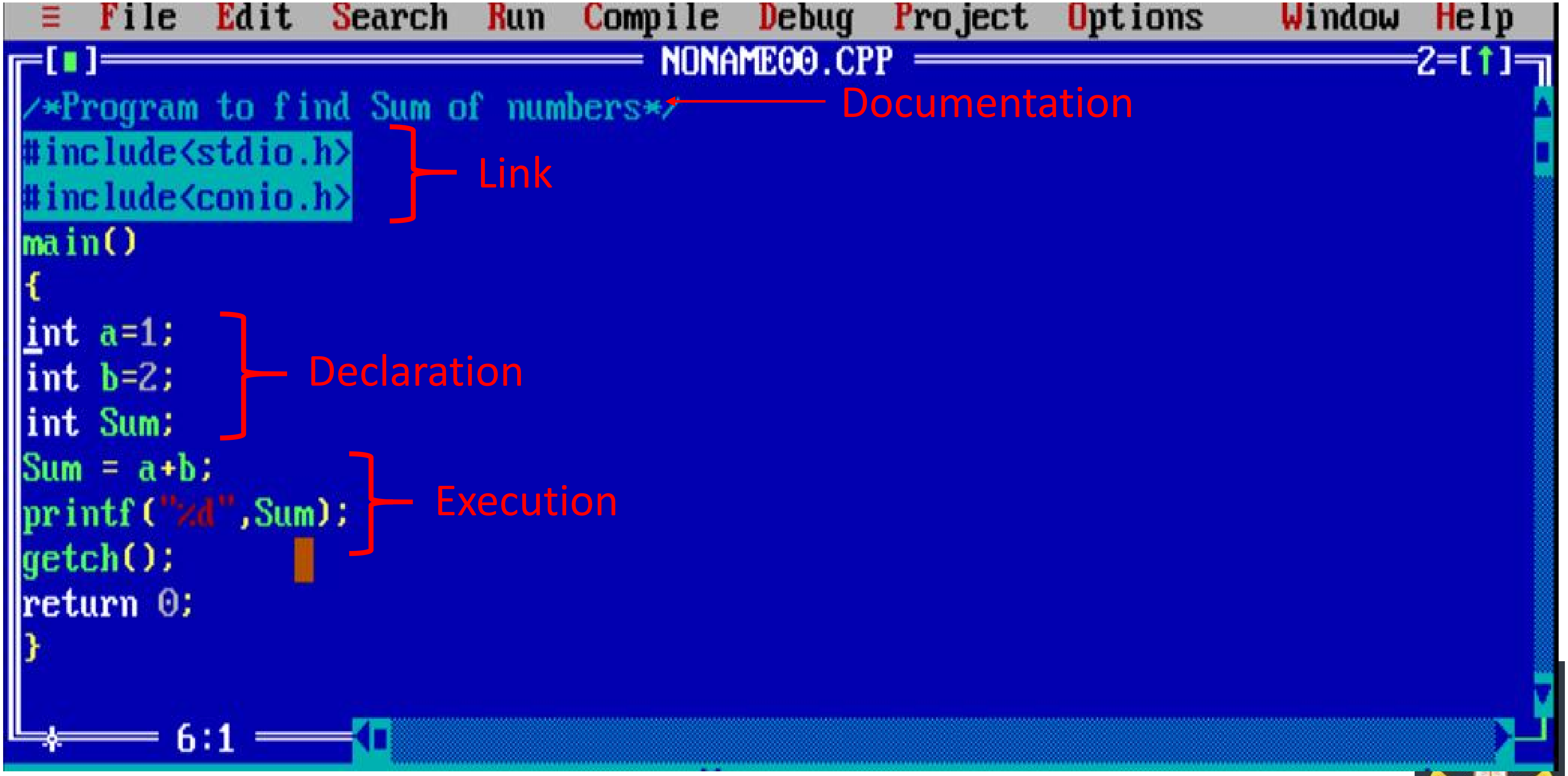
- There are some variables that are used in more than one function
- Such variables are called global variables
- These are declared in the global declaration section that is outside of all the functions
- **Global variables** hold their values throughout the program and they can be accessed inside any of the functions defined for the program
- This section also declares all the user-defined functions
- Eg: int age;

Structure of C Program

Main Function Section

- Every C-programs must have one **main ()** function
- Execution of C program starts with **main ()**
- Contains two parts,
  1. Declaration part
  2. Executable part
- These two parts comes between set of opening & closing curly braces
- Closing curly brace indicate end of function

```
main()
{
int a=10;              ← Declaration
printf (" %d", a);     ← Executable
}
```


≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.CPP
2
/*Program to find Sum of numbers*/
Documentation
#include<stdio.h>
#include<conio.h>
Link
main()
{
int a=1;
int b=2;
int Sum;
Declaration
Sum = a+b;
printf("%d",Sum);
getch();
Execution
return 0;
}
6:1

Structure of C Program

Sub Program Sections

- All the user-defined functions are defined in this section of the program
- User defined functions are generally placed immediately after main function, although they may appear in any order

Basic Structure of C Programs
Documentation Section
Link Section
Definition Section
Global Declaration Section

main() Function Section
{
Declaration Part
Executable Part
}

Subprogram Section Function 1 Function 2 Function 3 - - - Function n

Structure of C Program

Documentation Section

Link Section

Definition Section

Global Declaration Section

Main Function Section

{

Declaration Part

Executable Part

}

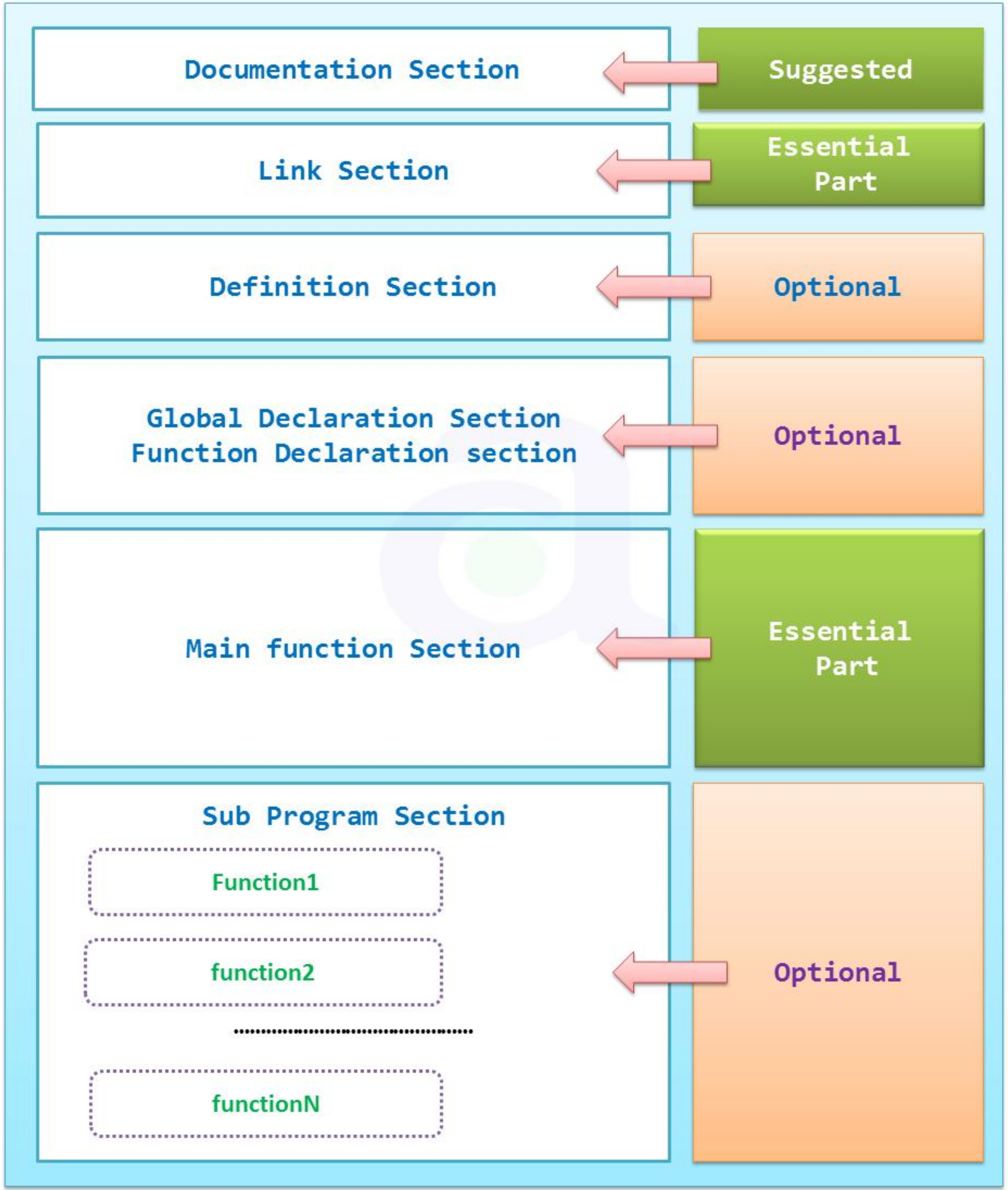
Sub-Program Section

function - 1

function - 2

function - 3

(User define functions)

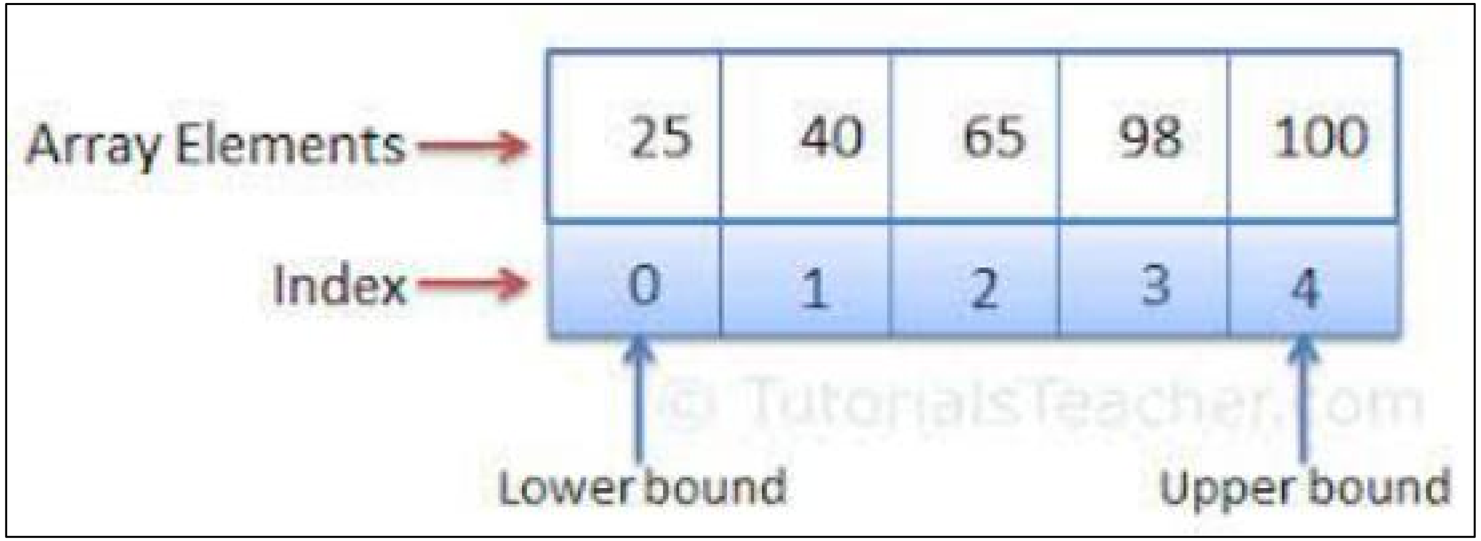

Documentation Section
Suggested
Link Section
Essential Part
Definition Section
Optional
Global Declaration Section
Function Declaration section
Optional
Main function Section
Essential Part
Sub Program Section
Function1
function2
functionN
Optional

ARRAY

MODULE 3

ARRAY

Array is an ordered list of homogeneous data elements that share a common name



ARRAY

Array is an ordered list of homogeneous data elements that share a common name

- Eg: int **number** [5]
- **number** represents an array which can hold 5 elements
- All elements must be integer data-type (in this example)
- Computer reserves five storage locations:

number [0]
number [1]
number [2]
number [3]
number [4]

number [0] -------- Indicate 1^{st} element in the array
number [1] -------- Indicate 2^{nd} element in the array
number [2] -------- Indicate 3^{rd} element in the array
number [3] -------- Indicate 4^{th} element in the array
number [4] -------- Indicate 5^{th} element in the array

ARRAY

- Each element in an array is referenced by a *subscript OR index* enclosed in a pair of square brackets
- This subscript indicates the position of an individual element in an array

Example: To store marks of 5 subjects in an array (21.5,22.0,23.5,24.5,25.0)

```
float mark [5];
mark [0] = 21.5
mark [1] = 22.0
mark [2] = 23.5
mark [3] = 24.5
mark [4] = 25.0
```

Classification of Array

Arrays are classified into two types:

1. **One dimensional arrays**
2. **Multi dimensional arrays**
   - Two-dimensional, Three-dimensional, ……… n-dimensional array

- The dimensionality of an array is determined by the number of subscripts present in an array
- If there is only one subscript it is called one dimensional array
  - ARRAY [ ]
- If there are two subscripts, then it is called two dimensional array
  - ARRAY [ ][ ]

Declaration of Array

- An array must be declared before they are used in the program

Syntax for declaring one-dimensional array:

datatype Array_name[size];

Datatype: Type of element contained in array - int, float, char etc.

Size: Number of elements that can be stored in an array

Eg1: **int age [20];**

  - Declares ***age*** as an array to contain maximum of 20 integers

Declaration of Array

Eg2: **char name [10];**

- Declares ***name*** as a character array (string) to contain maximum of 10 characters
- Reading string “WELL DONE” to array ***name***
- Each character of the string is treated as an element of array and stored as:

‘W’
‘E’
‘L’
‘L’
‘ ’
‘D’
‘O’
‘N’
‘E’
‘\0’

- A character string will be terminated with an additional null character ‘\0’
- Hence while declaring character arrays, we must allow one extra element space for null terminator

Declaration of Array

Rules for subscript:

1. Each subscript must be an unsigned positive integer or expression
2. Subscript of subscript is not allowed
3. C does not perform bounds checking. Therefore the maximum subscript appearing should not exceed the declared
4. In C, subscript value ranges from 0 to (size - 1). i.e, if the size is 10, first subscript is 0, second is 1 and last is 9.

```
float mark [5];
mark [0] = 21.5
mark [1] = 22.0
mark [2] = 23.5
mark [3] = 24.5
mark [4] = 25.0
```

Initialising One Dimensional Array

Initialising is the process of assigning value to a variable

- After array is declared, its elements can be initialised
- An array can be initialised at:
  - Compile time & Run time

Compile time initialisation

datatype Array_name[size] = { List of values};

Eg: int number [5] = { 11, 12, 13, 14, 15};

- Number of elements initialised may be less than declared size.
- In such cases, remaining elements will be initialised to zero

Run time initialisation

Can use read function scanf to initialise an array

One Dimensional Array

How to accept *n* integers and store them in an array

```
int n, i, number[20];
printf ("Enter the size of Array\n");
scanf("%d",&n);
printf ("Enter the elements one by one\n");
for(i=0, i<n; i++)
  {
     scanf("%d", &number[i]);
  }
```

```
float mark [5];
mark [0] = 21.5
mark [1] = 22.0
mark [2] = 23.5
mark [3] = 24.5
mark [4] = 25.0
```

One Dimensional Array

How to print *n* integers stored in an array

```
printf ("The elements in array is\n");
for(i=0, i<n; i++)
 {
    printf(" %d \n", number[i]);
 }
```

11
12
13
14
15

Two Dimensional Array

Two Dimensional Array

	Mark 1	Mark 2	Mark 3
Student 1	21	34	74
Student 2	38	87	74
Student 3	89	85	39
Student 4	44	53	85

- Two-dimensional array can be defined as an array of arrays
- The 2D array is organized as matrices which can be represented as the collection of rows and columns
- Example above: marklist [4] [3]

2D Array Declaration

Two-dimensional arrays are declared as:

datatype arrayname [row_size] [column_size];

Example: **int x [20][10];**

2D arrays are stored in memory as shown:

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

2D Array Initialisation

- Two dimensional arrays can be initialised after declaration
- With list of initial values enclosed in braces
- The initialization is done row by row

Example:

```
int table [2][3]={0,0,0,1,1,1};

int table [2][3]={{0,0,0},{1,1,1}};
```

0	0	0
1	1	1

2D Array Initialisation

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

matrix[0][0] = 1 matrix[0][1] = 2 matrix[0][2] = 3

matrix[1][0] = 4 matrix[1][1] = 5 matrix[1][2] = 6

matrix[2][0] = 7 matrix[2][1] = 8 matrix[2][2] = 9

1	2	3
4	5	6
7	8	9

If any values are missing during initialisation process, they will be automatically set to zero

/* Program to read elements to a matrix*/

```
int mat[10][10];
int row, col, i, j;
printf(“Enter order of matrix\n”);
scanf(“%d%d”, &row, &col);
printf(“Enter elements of matrix row-wise\n
for (i=0; i<row; i++)
{
   for (j=0; j<col; j++)
   {
      scanf(“%d”, & mat[i][j]);
   }
}
```

	0	1	2
0	**21**	**34**	**74**
1	**38**	**87**	**74**
2	**89**	**85**	**39**
3	**44**	**53**	**85**
row = 4, col = 3			

/* Program to print elements of a matrix*/

```
int mat[10][10];
int row, col, i, j;
printf(“Enter order of matrix\n”);
scanf(“%d%d”, &row, &col);
printf(“The matrix is \n”);
for (i=0; i<row; i++)
{
   for (j=0; j<col; j++)
   {
      printf(“%4d”,  mat[i][j]);
   }
printf(“\n”);
}
```

	0	1	2
0	21	34	74
1	38	87	74
2	89	85	39
3	44	53	85
row = 4, col = 3			

Programs to write:

1. Program to read n numbers to an array and display the same array.
2. Program to accept two arrays and find sum of their corresponding elements.
3. Program to input n numbers to an array and display the largest number.

4. Program to read and display a matrix.
5. Program to read a square matrix and to display its trance and transpose
6. Program to read two matrices and find their sum.

STRING

STRINGS

- String is a sequence of characters that is treated as a single data item
- One dimensional array of characters
- Eg: “computer” is a string stored as:

c	o	m	p	u	t	e	r	\0

- Each one dimensional character array ends with a null character **\0**
- printf(“WELL DONE”);
- Ouput string – WELL DONE

Declaration

- General form of declaration of a string variable:

char string_name [size];

- Size determines the number of characters in the string
- Example: char name[20];
- When compiler assigns a character string to an array it assigns a null character \0 at end
- Hence the size should be equal to maximum required number of the string plus one

Initialisation

- Character arrays may be initialised when they are declared
- Can be initialised in the following forms:

char msg [10] = “WELL DONE”;

char msg [10] = {‘W’, ‘E’, ‘L’, ‘L’, ‘ ’, ‘D’, ‘O’, ‘N’, E’, ‘\0’};

- 10 elements long: 9 characters + null terminator

Initialisation

- C also permits to initialise a character array without specifying number of elements
- In such cases, size of array will be automatically determined based on number of elements initialised
- Eg: char string[ ] = {‘G’, ’O’, ’O’, ’D’, ’\0’}; - declares as a five element array
- Can also declare a size larger than string size
- Eg: char string [10] = “ GOOD”; – creates an array of size 10, places “GOOD”, adds \0, initialise other elements as NULL

G	O	O	D	\0	\0	\0	\0	\0	\0

Reading & Writing Strings

Strings can be entered and displayed using following functions:

- scanf()
- gets()
- getchar()

- printf()
- puts()
- putchar()

Reading String

scanf function

scanf function can be used with %s format specifier

Example: char message[10];
scanf("%s", message);

- While using *scanf* for strings & is not required before variable name
- *scanf* function terminates its input on the first white space (blank, tab, newline etc.)
- Eg: If we enter "WELL DONE", only "WELL" will be read to the array address

Reading String

getchar function

- Can be used to read a single character
- Can be used repeatedly to read successive single characters from input and place them in a character array
- Thus an entire line of text can be read and stored in an array
- No parameter required

```
char ch;
ch = getchar();
```

Reading String

getchar function

To enter a line of text

```
char line[30], ch;
 int i = 0;
printf("Enter line: ");
while(ch != '\n') // terminates entry with newline
{
ch = getchar();
line[i] = ch;
i++;
}
line[i] = '\0'; // inserting null character at end
printf("Line is: %s", line);
```

Reading String

gets function

- Another convenient method to input string with white spaces
- Reads characters from keyboard until a newline is encountered and then adds a null character

```
char line [10];
gets (line);
```

Writing String

printf function

- %s format specifier to be used
- printf(“%s”, name);

putchar function

- To output values of character variables
- putchar (ch) ----------- requires one parameter

puts function

- char line [10];
- gets (line);
- puts (line);

/*Program to enter a string and count number of characters*/

```
void main ()
{
char text[20];
int count =0, i;
printf("Enter the text\n");
gets(text);

while (text[count]!='\0')    // Counting from index 0 till character before \0
{
count++;                     // increment count number if character is not \0
}
printf("%d",count);
```

Index	ch	count
0	G	1
1	O	2
2	O	3
3	D	4
4	\0	

Arithmetic Operations on Characters

- Each character constant has an ASCII value
- Whenever a character is used in an expression, its ASCII value is accessed

Example:

ASCII value of character A is 65

- Can print the integer representation of a given character constant using *printf*

```
char ch = 'A';
printf(" The character is: %c\n",ch); --------- A
printf(" The ASCII value is: %d\n",ch); ------- 65
```

Arithmetic Operations on Characters

```
int y, x = 10;
y = x+'a';            //10+97
printf("y = %d\n",y);
```

<u>To check upper case or lower case</u>

```
char ch;
ch>='A' && ch<='Z'      ch>=65 && ch<=90
ch>='a' && ch<='z'      ch>=97 && ch<=122
```

String Handling Functions

String Handling Functions

- String handling functions to manipulate the strings according to the need of problem
- Are defined under string.h header file
- The commonly used string handling functions are:

Function	Action
strcat()	Concatenates (join) two strings
strcmp()	Compares two strings
strcpy()	Copies one string over another
strlen()	Finds length of a string

strcat() Function

strcat function joins two strings together

Syntax: **strcat (string1, string2);**

- When this function is executed, string2 will be added to string1
- Done by removing the null character at the end of string1 and placing string2 from that location
- Text in string2 will remain unchanged
- Size of string1 must be large enough to accommodate final string

strcat() Function

strcat function joins two strings together

Syntax: strcat (string1, string2);

- Function can also join a string constant to a string variable
  - strcat(string1, “GOOD”);
- Nesting of **strcat** functions is also possible
  - strcat((strcat(string1, string2), string3)

strcmp() Function

- **strcmp function compares the strings in arguments**
- **Stntax: strcmp(string1, string2);**
  - If are equal give value 0
  - If not equal give the numeric difference between first non-matching character
- string1 & string2 may represent string variables or string constants
  - strcmp(text1, text2);
  - strcmp(text1, “good”);
  - strcmp(“well”, “done”);

strcmp() Function

- **strcmp function compares the strings in arguments**
- **Stntax: strcmp(string1, string2);**
- <u>Example:</u>
- strcmp(“there”, “there”); ------- return value 0
- strcmp (“their”, “there”); -------return -9, ASCII value difference between “i” and”r”
- If the return value is negative, string1 is alphabetically above string2

strcpy() Function

- strcpy is a string assignment operator
- Stntax: strcpy (string1, string2);
  - Assigns contents of string2 to string1
  - string1 & string2 may represent string variables or string constants
  - Sixe of string1 should be large enough to copy contents of string2

strlen() Function

- **strlen function counts and returns number of characters in a string**
- N= strlen (string);
- N is an integer variable, will be assigned with length of string
- Counting ends at the first null character
- Example: strlen (“well done”) ---------- 9

strstr() Function

- **strstr function locates a substring in a string**
- **Syntax: strstr (s1,s2);**
- Searches string s1 to check whether s2 is contained in it
- If yes, will return location of first occurrence
- If no, will return null pointer
- Example:

```
if(strstr(s1,s2) == null)
    printf("substring is not found");
else
    printf("s2 is a substring of s1");
```

Other string handling functions

- strncat (s1, s2, n) joins first *n* characters of s2 to the end of s1
- strncpy (s1, s2, n) ---- copies first *n* characters of s2 into s1
  - Null character(\0) should be supplied at the end
- strncmp (s1, s2, n) compares leftmost *n* characters of s1 to s2
  - Return 0 if equal, -ve if s1 substring is less than s2

FUNCTION IN C

MODULE 4

Contents

1. Modular programming
2. Functions
3. Recursion
4. Structure
5. Union
6. Storage classes

Modular Programming

Modular Programming is the process of subdividing a computer program into separate sub-programs

- Organising a large program into small & independent program segments called modules
- Each module is separately named and are individual units

Functions

Function is a **self-contained program segment** that **carries out some specific, well-defined task**

C functions classifications

1. Library functions
2. User-defined functions

Library Functions

- Each library function performs a predefined operation
- Example: strcat, sqrt, printf, scanf, getchar, strcat, sqrt
- These library functions are created at the time of designing the compilers

User-defined functions

- These are written by the programmer for carrying out certain tasks

Elements of user-defined function

1. Function declaration
2. Function call
3. Function definition

Need of Functions

- Large and difficult program can be divided in to sub programs and solved
- Task which has to be performed repeatedly at different locations in a program
- Function avoids this repetition or rewriting over and over

Function Elements

In 'C' programming functions are divided into three components :

1. Function definition
2. Function call
3. Function declaration

1. Function Declaration

- Function declaration means: we specify the name of a function (the sub-program) that we are going to use in the program - like a variable declaration
- All functions in a C program must be declared, before they are called
- A function declaration is also called "**Function Prototype**"

1. Function Declaration

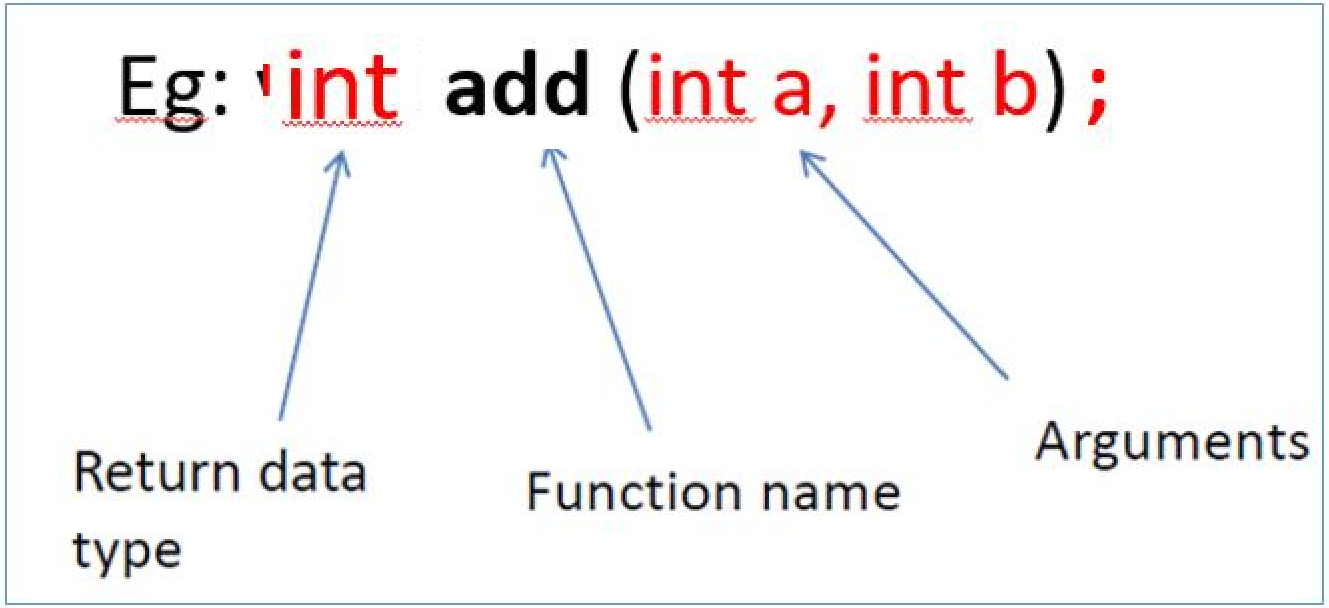
Format of function declaration:

Return-datatype Function-name (Parameter list) ;

Function declaration (function prototype) consists of four parts:

1. Function type (return type): data type of the value function returned back to the calling function
2. Function name
3. Parameter list
4. Terminating semicolon

1. Function Declaration



This means you have declared a function named add with arguments of the function as integer a and b.

1. Function Declaration

Examples of function declaration

- int **add** (int a, int b);
- int **add** (int, int);
- **add** (int, int);
- void add (int, int);

1. Function Declaration

Parameter list:

1. Parameter list must be separated by commas
2. The type of parameter must be same in the function definition, in number and order
3. If the function has no formal parameters, the list is written as void
4. The return type must be void if no value is returned
5. The return type is optional, when the function returns **int** type data

2. Function Call

- A function call means calling a function when it is required in a program
- When a function is called, it performs an operation for which it was designed
- Example: `add(4,5);`

2. Function Call

- A function is called by using the *function name* followed by list of parameters (or arguments)

Example:

```
main ()
{
int s;
s = add (3,5); ---------- function call (sends two values 3 & 5 to function)
printf(" sum = %d", s);
}
```

3. Function Definition

- Writing the body of a function
- A body of a function consists of statements which perform the specific task
- A function body consists of a single or a block of statements

3. Function Definition

- Function definition is the independent program module written to implement one particular task
- Also called function implementation
- **Include components:**

1. Function name
2. Function type
3. List of parameters
4. Local variable declaration
5. Function statements
6. A return statement

Items 1–3: Function header

Items 4–6: Function body

3. Function Definition

<u>Example: function to find sum of 2 numbers</u>

int add (int a, int b) ------------ Header

```
{
int c;
c = a + b;
return (c);
}
```

Function body

3. Function Definition

Function Header

1. Function name - Any valid C identifier
2. Function type
   - Specifies the data-type of the value the function is expected to return
   - If not specified, compiler will assume integer data type
   - If not returning any value, specify return type as void

3. Function Definition

Function Header

3. Parameter list

- Declares the variables which receive the data sent by the calling program
- List contains declaration of variables separated by commas
- Example: int add (int a, int b)
  - No semi-colon after parenthesis
  - Cannot combine datatype (int a, b) -----wrong

3. Function Definition

Function Body

Function body contains all the declarations and statements for performing the required task

3 components:

1. Local variable declaration ---- specifies variables needed
2. Executable statements ------- to perform task
3. Return statement -------- returns value evaluated by function
   - Function may nor may not return value to calling function. If it does, value is returned through return statement

3. Function Definition

Return statement

- Returns value evaluated by function
- Function may nor may not return value to calling function
- If it does, value is returned through return statement
- Return statement can be in following forms:

1. return; ----------- does not return any value, the control is immediately passed back to the calling function
2. return (expression); ---------- returns the value of the expression

```
#include <stdio.h>
#include <conio.h>
void add (int a, int b); //function declaration
void main()
{
   int a = 10, b = 20;
   add(10,20);          //function call
}
void add (int a, int b) //function
{
int c;
c = a + b;
printf (“SUM = %d\n",c);
}
```

Function - Example

Main function

```
main ()
{
  int s;
  s = sum (3, 5);
  .....
}
```

User defined function

```
int sum (int x, int y)
{
   int result;              /* local variable
  result = x + y;        /* x= 3, y=5
  return (result);
  }
```

Parameters

Actual Parameter :

- The **actual** value/expression that is passed into the function by a caller
- The variable or expression corresponding to a formal parameter that appears in the function call

Formal Parameter :

- A variable and its type as they appear in the function definition or prototype
- The identifier used **in a** function to stand for the value that is passed into the function by a caller

Main function

```
main ()
{
  int s;
  s = sum (3, 5);
  .....
}
```

Actual Parameter: (3, 5)

Formal Parameter: (int x, int y)

```
int sum (int x, int y)
{
   int result;          /* local variable
  result = x + y;       /* x= 3, y=5
  return (result);
  }
```

User defined function

Category of Functions

Functions can be categorised into:

1. With no arguments and no return value

2. With arguments but no return value

3. With no arguments but a return value

4. Function with arguments and return values

Category of Functions

1. Functions with no arguments and no return value

- Example: void name ( ); OR void name (void)
- The called function does not receive any data from the calling function
- Function does not return any value back to the calling function
- Hence no data transfer between calling function and called function
- The function can be defined with empty parameter list
- An empty parameter list may be specified by writing either void or nothing in the parenthesis of function definition.

/* function with no argument and no return value*/

```
#include<stdio.h>
#include<conio.h>
void printline();    // function declaration
void main ()
{ printf("Function is easy to learn");
   printline();                          // function call
     getch();     }
void printline()
{
int i;
printf("\n");
for ( i=0;i<30;i++)
{
   printf(" – ");    }
}
```

Function is easy to learn

Category of Functions

2. Functions with arguments but no return value

- Function receives data from calling function
- Called function does not return any value back to the calling function
- One-way data communication between calling function and called function

```
/* function with argument and no return value*/
#include<stdio.h>
void main()
{
   void max (int, int);
   int x,y;
   printf("Enter the value of x and y\n");
   scanf("%d%d", &x,&y);
   max(x,y);      ------------------------- Function call
}
void max (int p,int q)              --------------- Function
{
  if(p>q)
     printf("max = %d",p);
  else
     printf("max = %d",q);
}
```

OUTPUT

Enter the value of x and y

3 5

max = 5

Category of Functions

3. Functions with no arguments but a return value

- Function does not receive any data from calling function
- Uses local data to perform task
- After processing the computed data will be returned to calling function
- One-way data communication between calling function and called function

```
/* function with no argument but return value*/
#include<stdio.h>
void main()
{
   float add (); -------------Function declaration
   float sum;
   sum = add (); ----------- Function call
   printf("Sum = %.2f", sum);
}
float add ()   -------------------------- Function
{
    float x, y, s;
    x=20.8;
    y=50.9;
    s=x+y;
return (s);
}
```

OUTPUT

Sum = 71.7

Category of Functions

4. Functions with arguments and return value

- Function receives data from the calling function through arguments
- Function returns value to calling function

```
/* Function to add two numbers*/
#include<stdio.h>
void main()
{
  int a, b, sum;
  int add (int, int);
  printf("Enter the numbers\n");
  scanf("%d%d",&a, &b);
  sum = add(a,b);          ---------------------------- Function call
  printf("Sum = %d",sum);
}
int add (int n1, int n2)    -------------------------- Function
{
   int result;
   result = n1+n2;
   return(result);
}
```

OUTPUT

```
Enter the  numbers
 3 5
Sum = 8
```

Parameter Passing Methods

- While using functions, arguments are passed to the functions and their values are copied to the function
- **This is information exchange between** *calling function* and *called function*
- **This process of transmitting values from one function to other is called *parameter passing***
- There are 2 methods of parameter passing:

1. Pass by value (Call by value)
2. Pass by reference (Call by reference)

Parameter Passing Methods

	CALL BY VALUE	CALL BY REFERENCE
1	While calling a function, values of actual parameters are passed to function	While calling a function, instead of passing the values of variables, address of variables(location of variables) are passed to function
2	The value of each variable in calling function is copied into corresponding dummy variables of the called function	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function
3	The changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function	Any modifications made to the values in the called function, original values will get changed with in calling function

/* pass by value*/

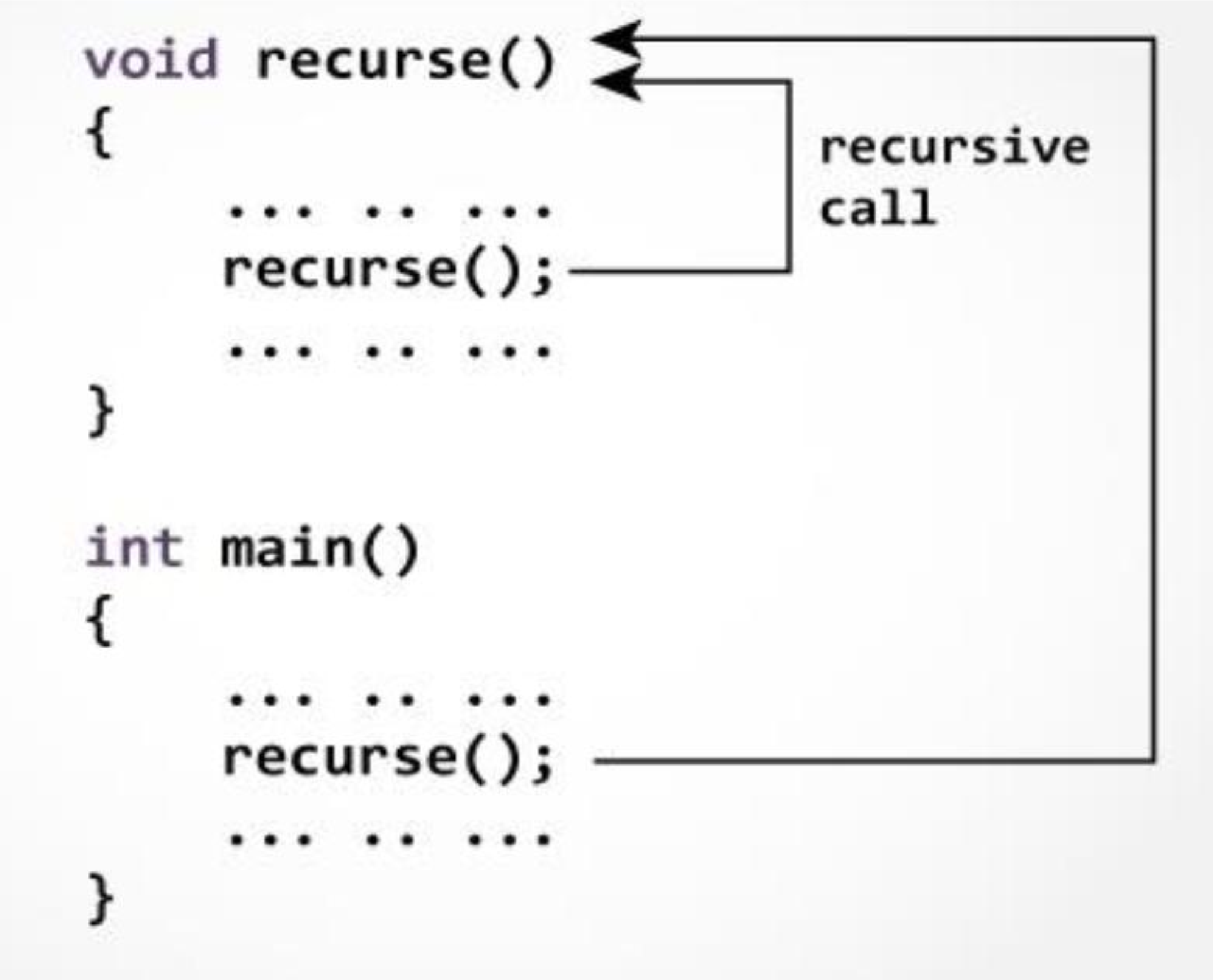
```
void main()
{
   void pass_value (int, int);
   int n1, n2;
   n1 = 5;
   n2 = 10;
   pass_value(n1, n2);
   printf(" n1=%d, n2=%d\n",n1, n2);
}
void pass_value(int a, int b)
{
   a=a+2;
   b = b+2;
   printf(" a=%d, b=%d\n", a,b);
}
```

```
a = 7, b = 12

n1 = 5, n2 = 10
```

Recursion

- In ‘C’ Recursion is the programming technique by which a function calls itself
- i.e., the function is repeated
- Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem
- While using recursion, **programmers need to be careful to define an exit condition from the function**, otherwise it will go in infinite loop

Recursion



Recursion - Example

/* Function to find factorial of a number*/

```
int factorial (int N)
{
    int fact;
    if ( N==1)
        return (1);
    else
       fact = N x factorial (n-1);
    return (fact);
}
```

To find factorial of 3, N=3

fact = 3 * factorial (2)

= 3 * 2 * factorial (1)

= 3 * 2 * 1

= 6

Passing Arrays to Functions
1 D Arrays

Passing Arrays to Functions

To pass one dimensional array:

Example: Function to find average of array ***list*** with size ***N***

: Assume parameters as array and its size

Function declaration	float average (float [ ], int);
Function call	average (list, N);
Function definition	float average (float arr[ ], int N)

Passing Arrays to Functions

- When an array is passed to a function, the values of the array elements are not passed to the function. Rather, the array name is interpreted as the address of the first array element
- This address is assigned to the corresponding formal argument when the function is called
- Arguments that are passed in this manner are said to be passed by reference
- Hence, any change in the array in the called function will result in change in the original array

Passing Arrays to Functions Example

```
main ()
{
float largest (float list [ ], int N); --------------- declaration
float list [4] = {2, 3, 4, 5};
printf("largest number = %f ", largest(list, 4))
}
float largest (float list [ ], int N) ---------------- definition
{
………..
…………
}
```

Passing Arrays to Functions 2 D Arrays

Passing 2D Arrays to Functions

- Function can be called by passing only the array name
- In the function definition, should indicate that array has two dimensions, by including two sets of brackets
- The size of second dimension must be specified
- Prototype declaration should be similar to function header

Passing 2D Arrays to Functions

To pass two dimensional array:

Example:

Function declaration	int average (int [ ] [N], int, int);
Function call	average (mat, M, N);
Function definition	int average (int mat[ ] [N], int M, int N)

STRUCTURES IN C

STRUCTURE

- Variables are used to store a single value of any data type
- Arrays are used to store a group of homogeneous elements
- To store and process a group of heterogeneous elements, C provide a datatype called **structure**
- Example:
  - **Student details**: Name, Roll no., Mark
  - Name is a character array, roll no. is an integer and mark is a float

STRUCTURE

- A structure is a collection of data items of different types
- Structure collects all the elements under a single name
- Each item in the structure is called as member
- '**struct**' keyword is used to create a structure
- All the members can be grouped together in one structure using keyword **struct**

STRUCTURE: DECLARATION

Syntax:

```
struct name
{
   datatype1 member1;
   datatype2 member2;
    …
    …
};
```

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
};
```

```
struct employee
{
   char name [20];
   int employee_code;
   float salary;
};
```

STRUCTURE: DECLARATION

Syntax:

```
struct name
{
   datatype1 member1;
   datatype2 member2;
    …
    …
} ;
```

In defining a structure note the following:

- Is terminated with a semicolon
- While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template

DECLARING STRUCTURE VARIABLES

- Like ordinary variables, structure variables can be defined and declared
- Structure variable declaration includes the following elements:

1. The keyword **struct**
2. The structure name
3. List of variable names separated by commas
4. Terminating semicolon

DECLARING STRUCTURE VARIABLES

Method 1 :

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
};
void main()
{
struct student st1, st2, s[10];
}
```

```
int a, b, c;
struct student st1, st2, s[10];
```

Here, st1 & st2 are structure variables having four fields for student

Similarly, s[0], s[1],…s[9]

DECLARING STRUCTURE VARIABLES

Method 2:

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
} st1, st2, s[10];
```

Here, st1 & st2 are structure variables having four fields for student
Similarly, s[0], s[1],…s[9]

DECLARING STRUCTURE VARIABLES

Method 1 :

```
struct student
{
  char name [20];
  int roll_no.;
  char branch [10];
  float total_mark;
};
void main()
{
struct student st1, st2;
}
```

Method 2:

```
struct student
{
  char name [20];
  int roll_no.;
  char branch [10];
  float total_mark;
} st1, st2;
```

Here, st1 & st2 are structure variables having four fields for student

STRUCTURE – INITIALISATION

- A structure variable can also be initialized in the way as any other data type in C

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
} st1 = {“abc”, 10, “civil”, 400};
```

OR struct student st1 = { “abc”, 10, “Civil”, 400} ;

Structure: Student 1	
name	abc
roll_no	10
branch	Civil
total_mark	400

ACCESSING A STRUCTURE

- Each member of the structure is accessed with the ***dot operator (.)***
- To access a particular member, dot operator must be placed between the name of structure and name of structure member

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
} st1, st2;
```

Example:

st1.name

st2.name

st1.roll_no.

st2.roll_no

```c
void main()
{
  struct book
  {
    int num;
    char author[20];
    float price;
};
  struct book b1;
  printf("Enter the book number\n");
  scanf("%d",&b1.num);
  printf("Enter the name of author\n");
  scanf("%s",b1.author);
  printf("Enter the price of book\n");
  scanf("%f",&b1.price);
  printf("\n Book no.  :%d",b1.num);
  printf("\n Author    :%s\n",b1.author);
  printf("\nPrice     :%f\n",b1.price);
}
```

```
Book no.    :111
Author      :abc
Price       :100.000000
```

ARRAY OF STRUCTURES

- An array of structures in C can be defined as the collection of multiple structure variables where each variable contains information about different entities
- The array of structures can be considered as a **collection of structures**
- Consider the case, where we need to store the data of 10 students

ARRAY OF STRUCTURES

```
struct student
{
   char name [20];
   int roll_no.;
   char branch [10];
   float total_mark;
} s[10];
```

PASSING STRUCTURES TO FUNCTIONS

- Structures can be passed as an argument to a function

Method 1

- Individual elements of structures can be passed to functions as actual arguments

Method 2

- Entire structure can be passed by passing a copy to the function

Method 3

- Concept of pointers can be used to pass the structure

```
/* structure passed to function: method 1, passing individual elements*/
void main()
{
   void output (int, char[], float);  // function declaration
   struct book                  // structure declaration
  {
    int number;
    char author [10];
     float price;
   } b1 = {11, “abc",100}; // structure variable declaration
output (b1.number, b1.author, b1.price);   //passing variables
}
void output (int n, char a[],float p )
{   printf("%d   %s   %f", n, a, p);
}
```

Method 2

```
void main()
{
struct emp                                  // structure declaration
 {
   char name[10];
   int id;
   float salary;
 }e1;
 void display (struct emp);          // function declaration
 printf("Enter name, id and salary of employee1\n");
 scanf("%s %d %f",e1.name, &e1.id, &e1.salary);
 display (e1);
}
void display(struct emp e1)
{
 printf("Employee name: %s", e1.name);
 printf("Employee id: %d", e1.id);
 printf("Employee salary: %f",e1.salary);
}
```

UNION

UNION

- Union is also a collection of heterogeneous items
- Union is a concept similar to structure and follow the same syntax as structure
- But differ from structure in terms of storage
- **In structures, each member has its own storage location, whereas all the members of a union use the same location**
- A union can store only one member at a time
- Unions, hence can be used to save memory, as only one of its members can be accessed at a time

UNION

Syntax:

```
union union_name
{
   datatype1 member1;
   datatype2 member2;
    …
    …
};
```

Example:

```
union data
{
  int i;
  float f;
  char str[20];
};
void main ()
{
union data data1, data2;
}
```

UNION

- When a union is created, the compiler allocates a storage space that is large enough to hold the largest variable type in the union

```
union data
{
  int i;
  float f;
  char c;
};
```

- In this case, float requires more space than others (4 bytes) Hence four bytes is allocated to union data.
- We can store either an integer, a float or a character at any one time, but not all of them together.

- This declares a variable code of type union data. The union contains three members, each with different data type
- However, we can use only one of them at a time

```
#include<stdio.h>
#include<conio.h>
void main()
{
union number // union declaration
  {
   int n;
   char c;
   float f;
  };
  union number x;   // variable declaration
  x.n = 11;
  x.c = 's';
  x.f = 1.1;
printf("Value of n = %d\n",x.n);
printf("value of c = %c\n",x.c);
printf("value of f = %.1f\n",x.f);
}
```

```
Value of n = -13107
value of c = =
value of f = 1.1
```

STORAGE CLASS

STORAGE CLASS

In C language, each variable has a storage class which decides the following things:

- **Scope** – over what region of a program the value of variable would be available
- **Default initial value** - if we do not explicitly initialize that variable, what will be its default initial value
- **Lifetime of that variable** - for how long will that variable exist

STORAGE CLASS

The following variable storage classes are most relevant to functions:

1. Automatic Variables
2. External or Global Variables
3. Static Variables
4. Register Variables

AUTOMATIC VARIABLES

- Automatic variables are declared inside a function in which they are to be utilized
- They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic
- Automatic variables are therefore private or local to the function in which they are declared
- Since these are local to a function, are also referred to as local or internal variables

AUTOMATIC VARIABLES

- We may also use keyword ***auto*** to declare automatic variables
- Eg: auto int number
- A variable declared inside a function without storage class specification is, by default, an automatic variable

Scope: Variable defined with auto storage class are local to the function block inside which they are defined

Life time: Till the end of the function block where the variable is defined

Example: Automatic variable

```
void main()

{

int a=10, b=5;

int s = a + b;

}  // a,b,s are variables declared in main function – automatic variables, local variables in main function
```

EXTERNAL VARIABLES

- External variables are those which are both alive and active throughout the entire program
- They are also known as global variables
- Global variables can be accessed by any function in the program
- External variables are declared outside a function

Example: External variable

```
#include<stdio.h>
int a=10;
void main()
{
  void display (void);
  clrscr();
  int s = a+10;
  printf("s = %d\n",s);
  display();
}
void display(void)
{
printf("a from function named display = %d", a);
}
```

- *a* is a global variable available to all functions
- *s* is a variable local to main function

```
s = 20
a from function named display = 10
```

STATIC VARIABLES

- The value of static variables persists until the end of the program
- A variable can be declared static using the keyword **static**
- Eg: static int x; static float y;
- A static variable can either be internal or external depending upon the place of declaration.
  - Scope of internal static variable remains inside the function in which it is defined
  - Scope of external static variables remain restricted to the scope of file in which they are declared

Example: Static variable

```
void add(void)
main ()
{   int i;
    for (i=1;i<=3;i++)
     add();
}
void add()
{
static int x = 1;
printf (“%d\n”,x);
x++;
}
```

output

x=1

x=2

x=3

REGISTER VARIABLES

- Register variables inform the compiler to store the variable in CPU register instead of memory
- Register variables have faster accessibility than a normal variable
- If frequently used variables are kept in register – will lead to faster execution
- But only a few variables can be placed inside registers
- One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time

Syntax :

- register int number;

REGISTER VARIABLES

- Since only a few variables can be places in the register, it is important to carefully select register variables
- However, C will automatically convert register variables into non register variable once limit is reached

Table 9.1 *Scope and Lifetime of Variables*

Storage Class	*Where declared*	*Visibility (Active)*	*Lifetime (Alive)*
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with **extern**	Entire program (Global)
extern	Before all functions in a file (cannot be initialized) **extern** and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or **auto**	Inside a function (or a block)	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

FILE OPERATIONS

FILE

A **file** is a space in a memory where data is stored.

1. Text files
2. Binary files

FILE management in C

- All files should be declared as FILE datatype
- To process a file, it should be opened
- Once opened – the data in the file can be read, added, or fresh data can be written
- While opening the file ---- the purpose also must be specified
  - For reading, writing or appending

OPENING A FILE

General format for declaring and opening a file:

```
FILE *fp;

fp = fopen("filename", "mode");
```

- 1st statement: declares variable *fp* as a pointer to datatype FILE
- 2nd statement:
  - Opens a file named *filename* and assigns to FILE type pointer *fp*
  - Specifies the purpose of opening this file

OPENING A FILE

General format for declaring and opening a file:

```
FILE *fp;

fp = fopen(“filename”, “mode”);
```

```
FILE *p1, *p2;
p1 = fopen(“input”, “w”);
p2 = fopen(“output”, “r”);
```

OPENING A FILE – File mode

r	▪ Open an existing file for reading only. ▪ If the file does not exists, an error occurs. ▪ If the file exists, it will be opened with current contents safe.
w	▪ Open file for writing only. ▪ If the file does not exists, a file with the specified name will be created. ▪ If the file exists, the contents will be deleted
a	▪ Open file for adding data to it. ▪ If the file does not exist, a file with the specified name will be created. ▪ If the file exists, it will be opened with current contents safe.

OPENING A FILE – File mode

r+	▪ Open an existing file for reading and writing.
w+	▪ Creates a file for reading and writing. Erases file if existing
a+	▪ Opens an existing file for reading and appending. Creates file, if does not exist
rb	▪ Open a binary file for reading
wb	▪ Create a binary file for Writing
ab	▪ Append to a binary file

CLOSING A FILE

- A file must be closed when all operations on it are completed
- Format of closing a file:

fclose(file pointer);

Example:

```
FILE *p;
p = fopen("output", "r");
fclose (p);
```

CLOSING A FILE

- Failure to close files may result in corrupted files and loss of data
- **fcloseall**() ------To close a number of open file at a time

Input – Output operations on files

Input – Output Operations

Function name	*Operation*
fopen()	* Creates a new file for use. * Opens an existing file for use.
fclose()	* Closes a file which has been opened for use.
getc()	* Reads a character from a file.
putc()	* Writes a character to a file.
fprintf()	* Writes a set of data values to a file.
fscanf()	* Reads a set of data values from a file.
getw()	* Reads an integer from a file.
putw()	* Writes an integer to a file.
fseek()	* Sets the position to a desired point in the file.
ftell()	* Gives the current position in the file (in terms of bytes from the start).
rewind()	* Sets the position to the beginning of the file.

Input – Output Operations

Different classes of read-write systems:

1. One character at a time
2. One integer at a time
3. One line of statement at a time
4. One block of statements at a time
5. Assorted types of data at a time

getc – putc functions

- Can handle only one character at a time

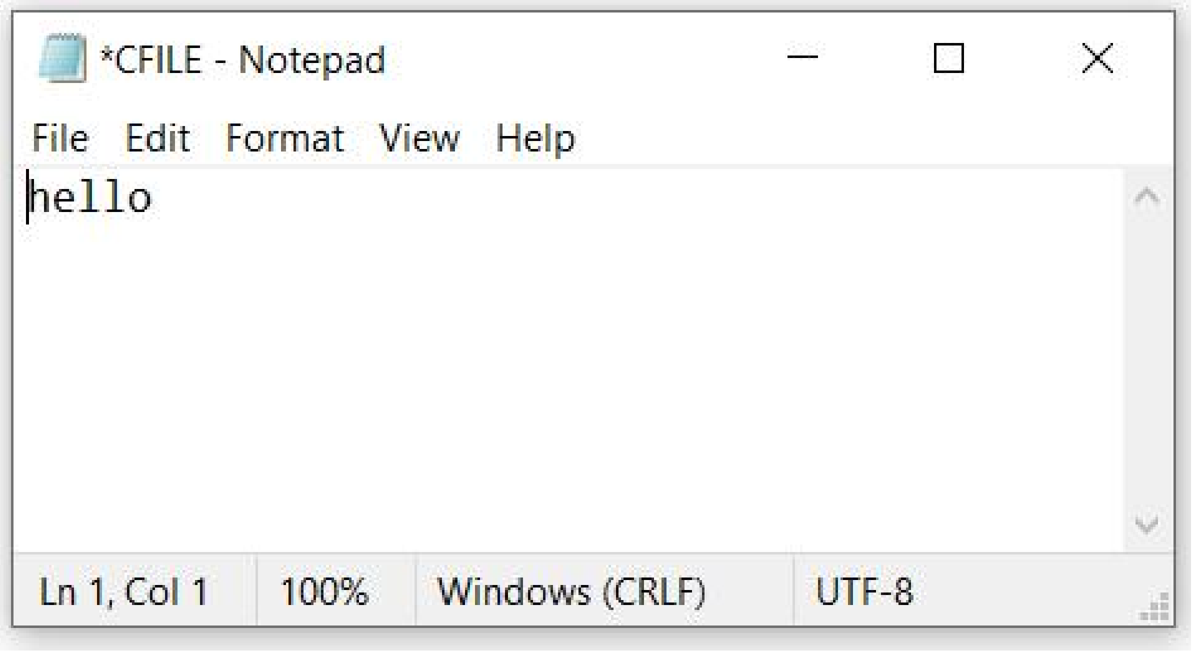
Format:

ch = getc(fx) -------- read a character from file pointed by *fx* and assign to *ch*

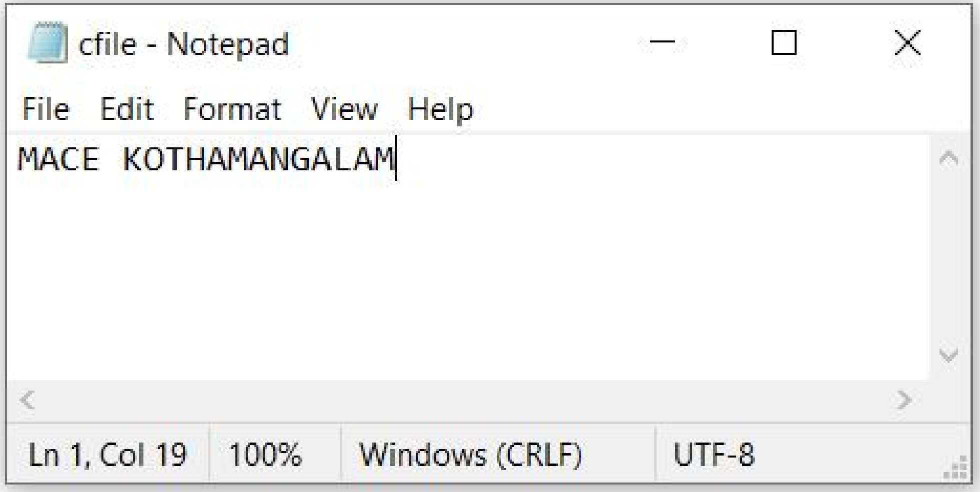
putc (ch, fy) --------- writes the character stored in variable *ch* to file pointed by *fy*

Program to read from file using **getc** and write it to console screen

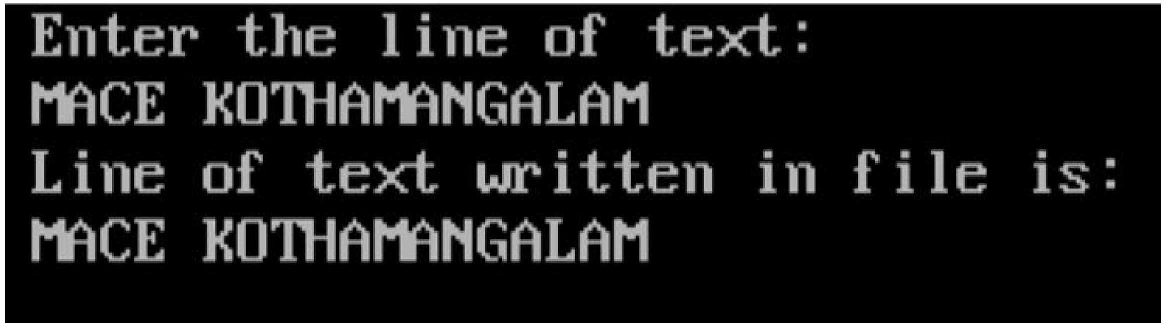
```
void main()
{
char ch;
 FILE *fx;
fx = fopen("cfile.txt", "r");
printf("Line of text written in file is:\n");
while ((ch = getc(fx))!=EOF)
printf("%c", ch);
fclose(fx);
}
```

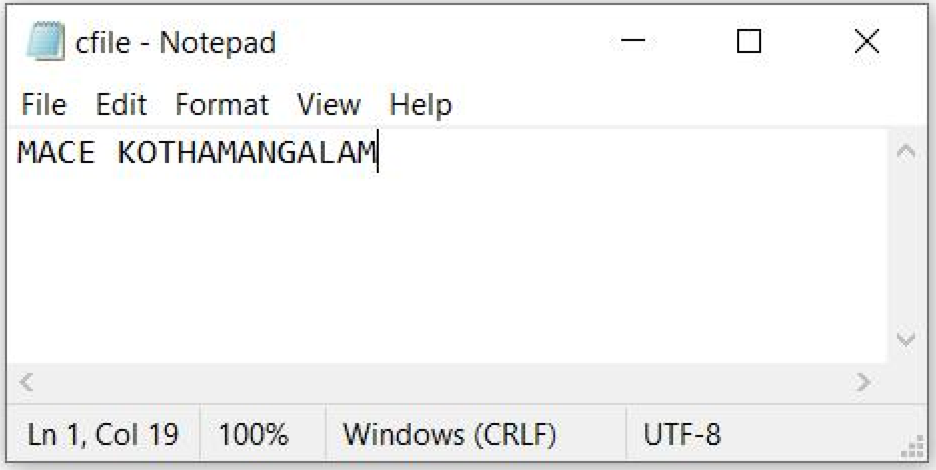


```
void main()
{
  char ch;
  FILE *fx;
  fx = fopen("cfile.txt", "w");  //opening file
    printf("Enter the line of text: \n"); // Entering string through console
    while (  (ch = getchar()) != '\n')  // input character until new line
  putc (ch, fx);   // adding the character in to the file
  fclose(fx);
}
```



```
void main()
{
  char ch;
  FILE *fx;
  fx = fopen("cfile.txt", "w");  //opening file
    printf("Enter the line of text: \n"); // Entering string through console
    while (  (ch = getchar()) != '\n')  // input character until new line
  putc (ch, fx);   // adding the character in to the file
  fclose(fx);
fx = fopen("cfile.txt", "r");
printf("Line of text written in file is:\n");
while  ( (s = getc(fx))!= EOF)
printf("%c", s);   // getting from file
fclose(fx);
}
```





getw – putw functions

- To read and write integer values

<u>Format:</u>

n = getw(fx) ------- reads an integer from file pointed by *fx* and assign to *n*

putw (n, fy) -------- writes the integer stored in variable *n* to file pointed by *fy*

fgets – fputs function

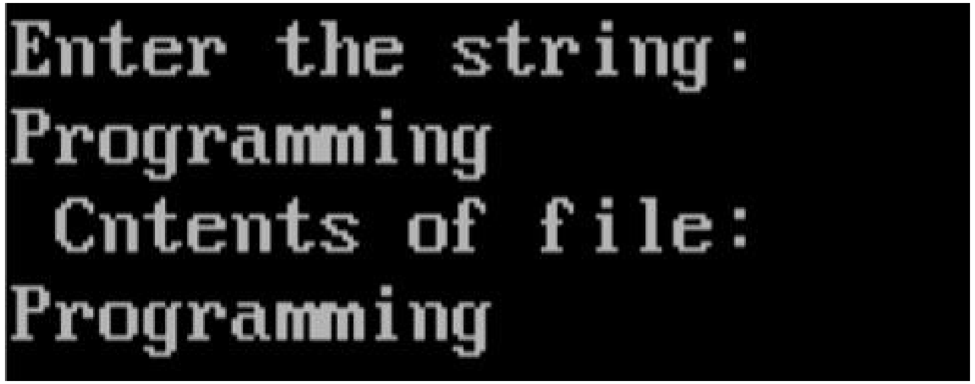
To read and write strings to files

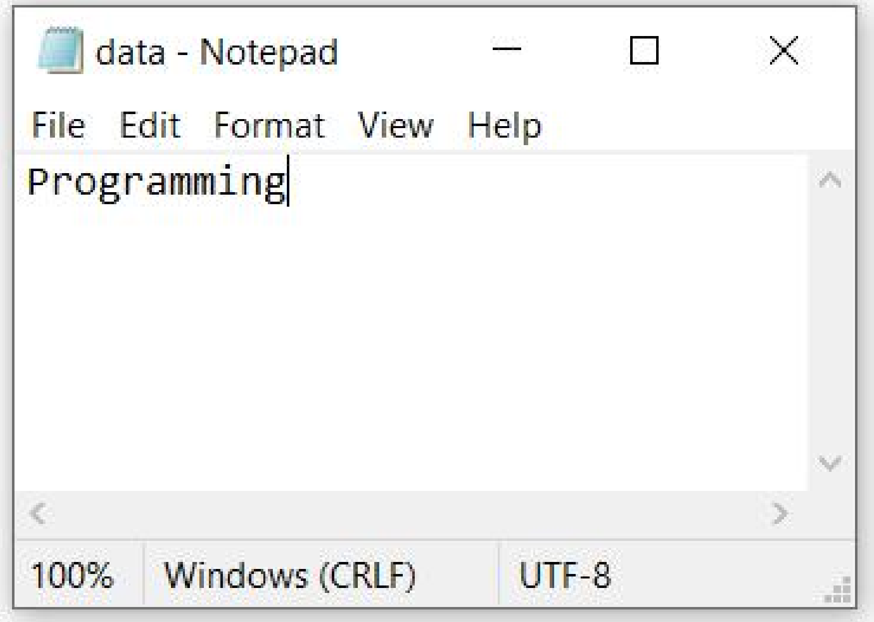
<u>Format:</u>

fgets (str, size, fx) ------- **read** characters up to length 'size' from file pointed by *fx* and assign to string variable *str*

fputs (str, fy) -------- writes the contents of string variable *str* to file pointed by *fy*

```
void main()
{
 char str[50], strnew[50];
 FILE *fx;
fx = fopen(“data.txt", "w");
  printf("Enter the string: \n");  // Entering string through console
  gets (str);  // reading string
fputs(str,fx);   // adding string to file
fclose(fx);   */
   fx = fopen(“data.txt", "r");
   printf(" Cntents of file:\n");
   fgets(strnew, 20, fx);    //  reading string from file
   fclose(fx);
   puts(strnew);
}
```





fprintf – fscanf files

- Can handle a group of data

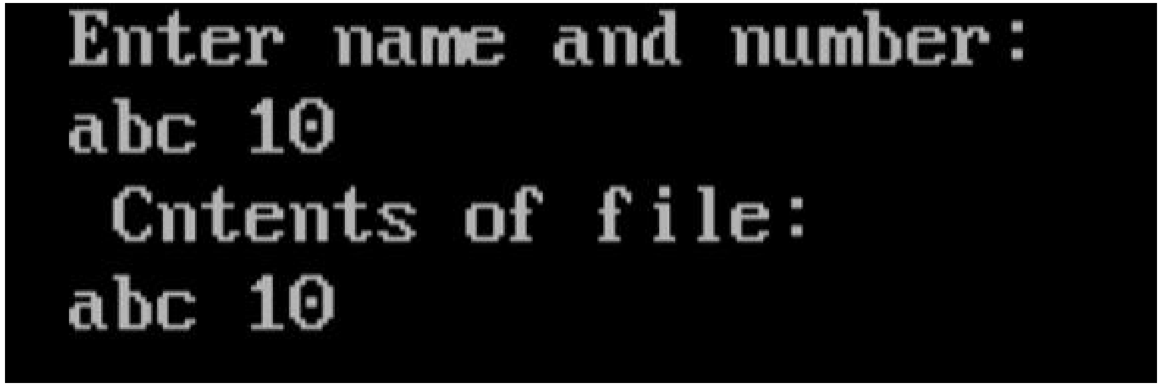
fprintf (*fp, "control string", variable list*);

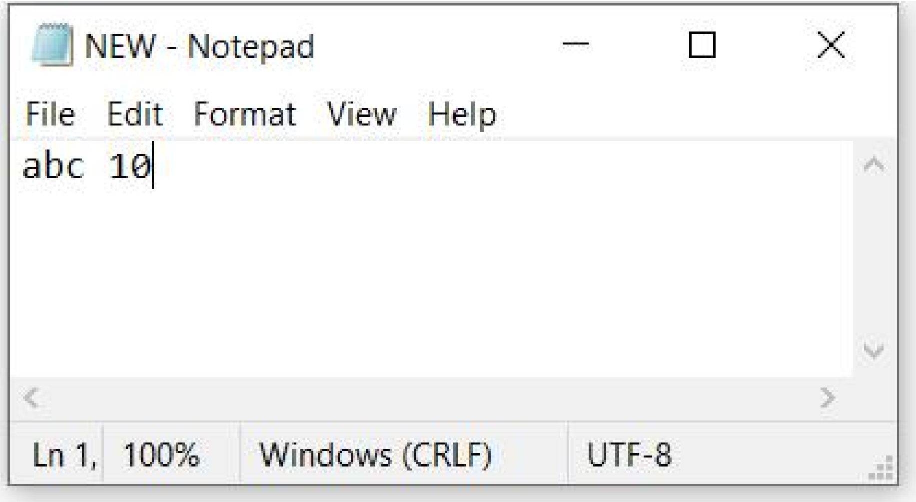
fscanf (*fp, "control string", & variable list*);

```
fprintf(fx, " %s  %d  %f ", name, rollno., 10.5);
```

- ***fp:*** The file pointer for the file opened for writing
- ***Control string***: Format specifiers for items in the list
- ***List:*** may include variables, constants and strings

```
void main()
{
char name[10];
 int num;
 FILE *fx;
 fx = fopen("new.txt", "w");
printf("Enter name and number: \n");
scanf("%s %d", name, &num);
fprintf (fx, "%s %d", name, num);  // writing to file
fclose(fx); */
  fx = fopen("new.txt", "r");
  printf(" Cntents of file:\n");
  fscanf(fx,"%s %d",name, &num);  // reading from file
  printf("%s %d", name, num);
  fclose(fx);
}
```





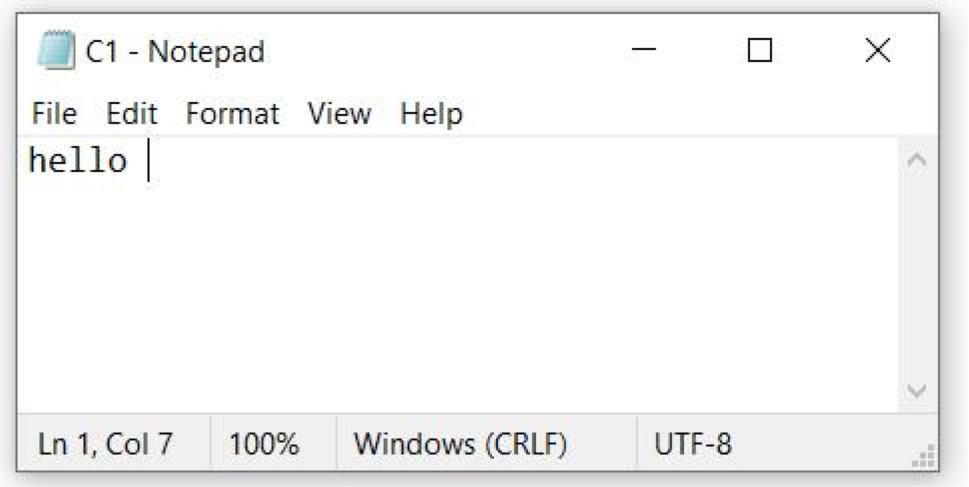
Appending to a file

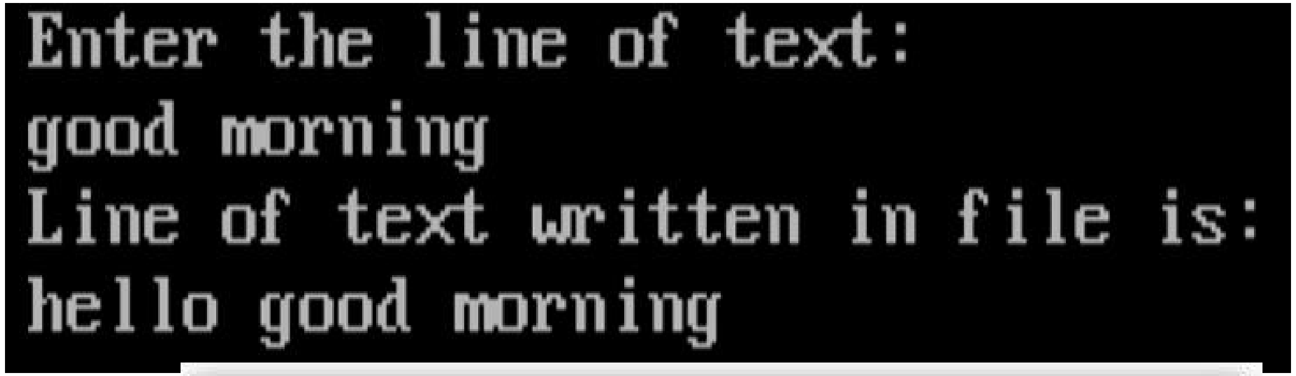
- Process of adding new data to a file
- Appending mode:

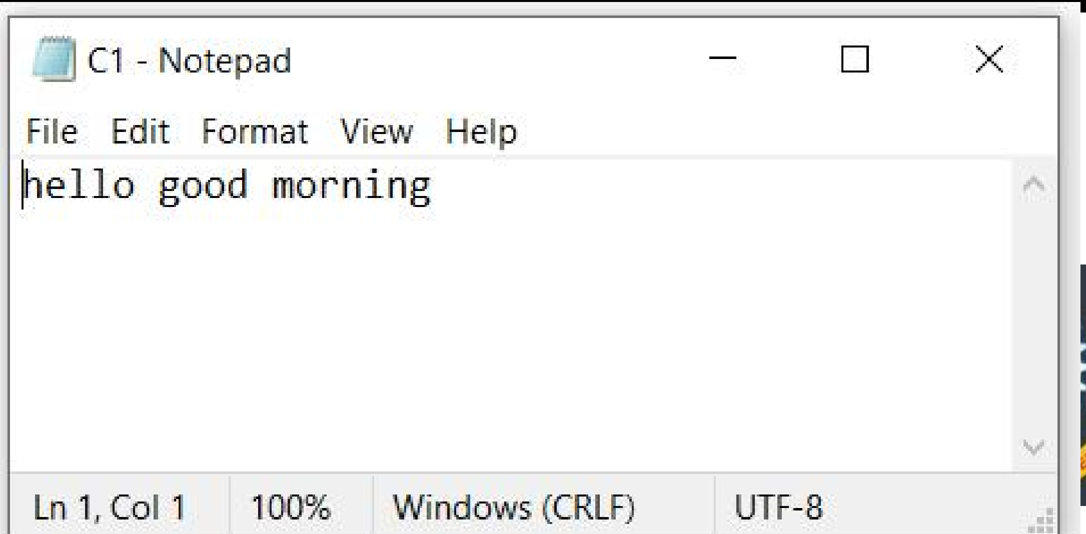
a	▪ Open file for adding data to it. ▪ If the file does not exists, a file with the specified name will be created. ▪ If the file exists, it will be opened with current contents safe.
a+	▪ Opens an existing file for reading and appending. Creates file, if does not exist.

- When a file is opened in Append(a) mode, <u>the cursor is positioned at the end of the present data</u> in the file

```
void main()
{
char ch;
 FILE *fx;
 fx = fopen("c1.txt", "a");
 printf("Enter the line of text: \n");

 while ( (ch = getchar())!='\n')
    putc (ch, fx);
 fclose(fx);

fx = fopen("c1.txt", "r");
printf("Line of text written in file is:\n");
while ((c = fgetc(fx))!=EOF)
printf("%c", c);
fclose(fx);
}
```







Sequential and Random Access of Files

Sequential And Random Access

Sequential access: computer system reads or write information sequentially, starting from beginning of the file and proceeding step by step towards the end

Random access: computer system can read or write information anywhere in the data file. Can be accessed with:

1. rewind()
2. ftell()
3. fseek()

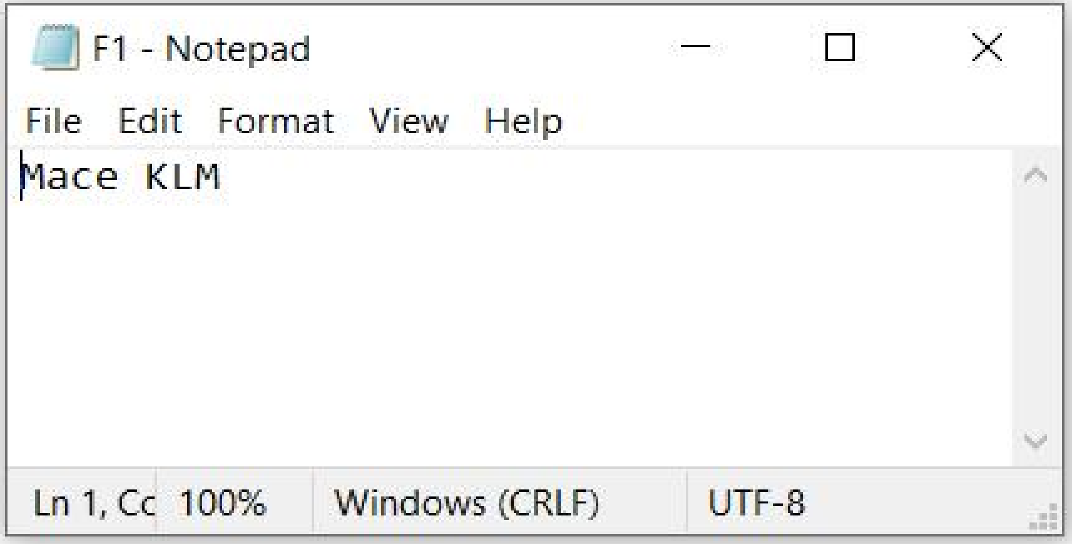
rewind()

rewind () function: Resets the file position indicator to the beginning of the file

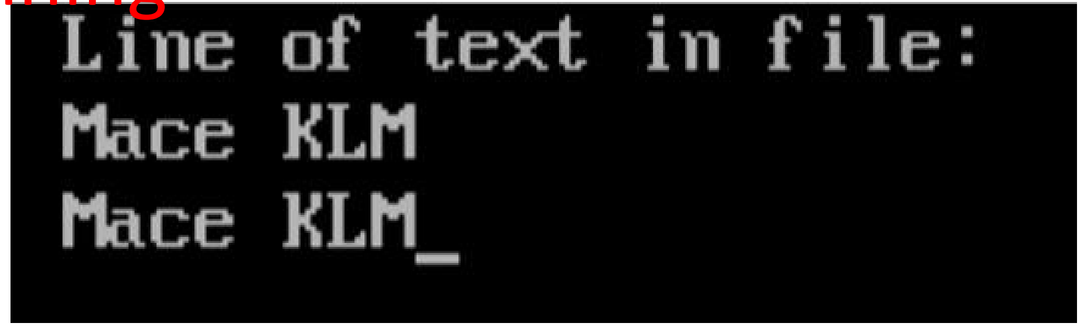
Format: rewind (fx) ----fx is the file pointer

Whenever a file is opened for reading or writing rewind is done automatically

rewind()




```
void main()
{
char c, ch;
FILE *fx;
fx = fopen("f1.txt", "r");  // opening file to read
printf("Line of text written in file data is:\n");
while ((c = getc(fx))!=EOF)  // reading each character
printf("%c",c);
rewind(fx);     // moving position to beginning
while ((ch = getc(fx))!=EOF)
printf("%c",ch);
}
```



fseek()

fseek() function is used to move the file position to a desired location within the file

fseek (file_pointer, offset, position)

file_pointer:	Pointer to the file specified
Offset:	Number of bytes to be moved. If offset is prefixed with a negative sign, it means that pointer needs to be moved in the backward direction
Position:	Start position for moving

fseek()

fseek (file_pointer, offset, position)

Position can take up following three values:

Value	**Meaning**
SEEK_SET or 0	Beginning of file
SEEK_CUR or 1	Current position
SEEK_END or 2	End of file

fseek()

fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

ftell ()

- **ftell()** function returns the current position of file position indicator relative to the beginning of file
- Will return the number of bytes from the beginning of the file

Format: ftell(fx) ----fx is the file pointer

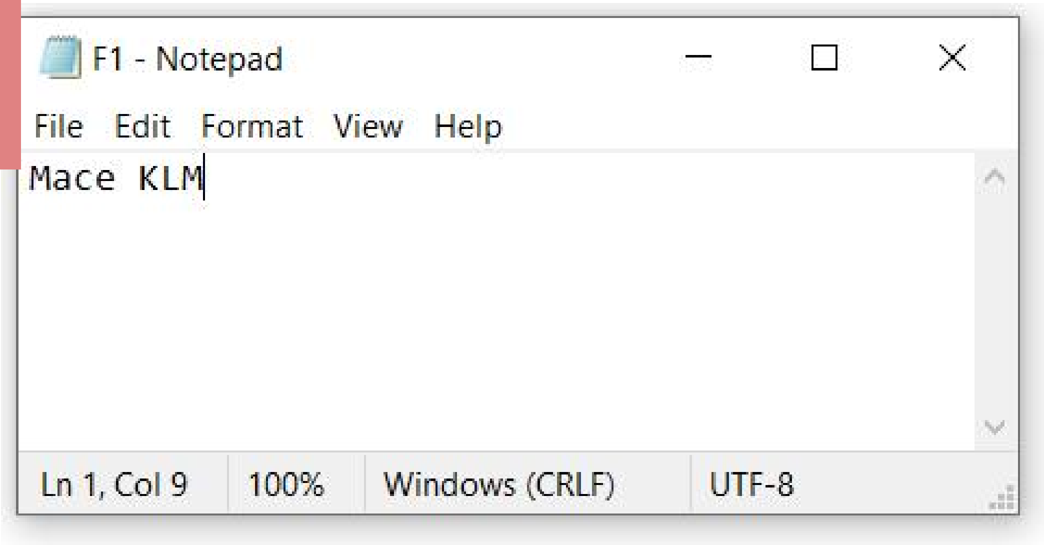
- n = ftell(fx) -------- n will give the current position relative to the beginning

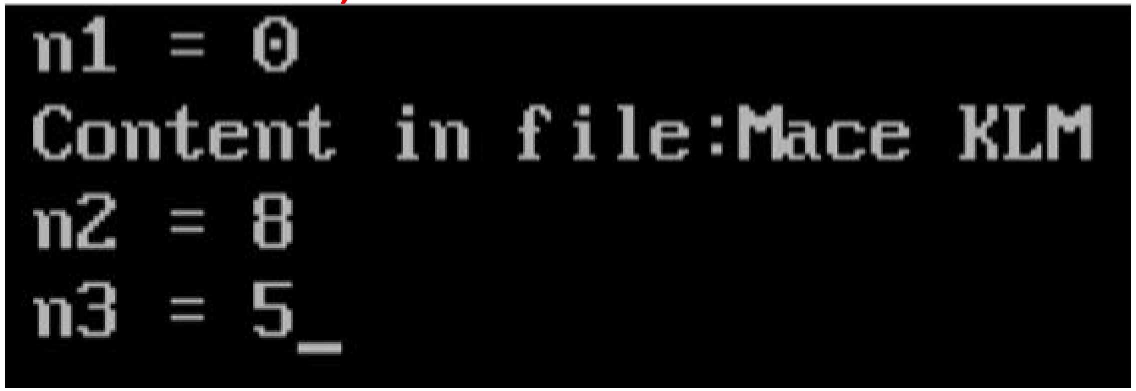
ftell ()

- n = ftell() ------ means *n* bytes are already written or read
- ftell() function is useful in saving the current position of file which can be used later in program

fseek() & ftell()

```
void main()
{
FILE *fx;
 int n1, n2, n3;
 char ch;
 fx = fopen("f1.txt","r");  // opening file to read
 n1 = ftell(fx);      // while opening position at start, n1 = 0
 printf("n1 = %d\n", n1);
 printf("Content in file:");
   while((ch = getc(fx))!=EOF)  // reading each character
   printf("%c", ch);
   n2 = ftell(fx);              // after reading position reaches end, n2 = 8
   printf("\nn2 = %d", n2);
 fseek(fx, -3, 1);  // moving
 n3 = ftell(fx);
 printf("\nn3 = %d", n3);
}
```





feof()

feof(): Function is used to locate the end of a file while reading the data.

- Return value:
  - Zero --- when position indicator is not at end of file
  - Non-zero ---- when end of file is reached

fread - fwrite

```
fread( ptr, int size, int n, FILE *fp );
fwrite( ptr, int size, int n, FILE *fp );
```

- *fread* and *fwrite* -- Used to read and write an entire block of data to and from a file
- These functions require 4 arguments:

fread ()

fread(ptr, int size, int n, FILE *fp);

- These functions require 4 arguments:

1. ptr : ptr is the reference of where data will be stored after reading
2. Size: Total number of bytes to be read from file (size of block)
3. N: Number of times the data block to be read
4. fp: Pointer to file where the records will be read

fread ()

Read integer value from file

```
FILE *fx
fread (&s, sizeof(int), 1, fx);
```

- Will read contents from file and store in variable *s*

Read array from file

```
FILE *fx
int a[10]
fread (a, sizeof(a), 1, fx);
```

- Would read an array of 10 integers from the file and stores in variable *a*

fread ()

Read first n elements of an array

```
FILE *fx
int a[10]
fread (a, sizeof(int), 5, fx);
```

- Would read first 5 integers from the file and stores in variable *a*

fread ()

Read a structure variable

```
struct student
  {
     char name[10];
     int roll;
     float marks;
};
struct student student1;
 fread(&student1, sizeof(student1), 1, fp);
```

fwrite ()

```
fwrite( ptr, int size, int n, FILE *fp );
```

- These functions require 4 arguments:

1. ptr : reference of an array or a structure stored in memory
2. Size: total number of bytes to be written
3. N: Number of times the data block to be written
4. fp: Pointer to file where the records will be written

fwrite ()

```
fwrite (&s, sizeof(int), 1, fx);
```

- Would write contents of *int* type variable *s* to the file pointed by *fx*

```
fwrite (arr, sizeof(int), 5, fx);
```

- Would write the first five elements of array, *arr*, to the file pointed by *fx*

```
int i, a[10], b[10];
FILE *fx;
 fx = fopen("sample.txt","w");   // opening file to write
printf("Enter 5 numbers\n");
for(i=0; i<=4;i++)
 scanf("%d",&a[i]);
fwrite(a, sizeof(int), 5, fx);  // writing numbers from array a to file
fclose(fx);
fx = fopen("sample.txt","r");   // opening file to read
fread(b, sizeof(int), 5, fx);       // reading first 5 numbers from file to assign to b
printf("The numbers stored in file 'sample' are:\n");
for(i=0; i<=4;i++)
 printf("%4d",b[i]);
fclose(fx);
```

```
Enter 5 numbers
3 4 5 6 7
The numbers stored in file 'sample' are:
   3   4   5   6   7
```

MODULE 5

Pointers

POINTERS

- A **pointer** is a variable that stores the address of another variable

- Example: An **integer variable** holds an integer value, however an **integer pointer** holds the address of an integer variable

Operators used with pointers:

1. The address operator: & --------- Gives the address of variable

2. The indirection operator: * -------- Gives the value of variable that the pointer is pointing to. ----- value at the address

POINTER - DECLARATION

- Pointers must be declared before they are used in program
- A pointer declaration has the following form

  data_type *pointer_variable_name;

- Example: 1) int *p; 2)float *x;
  - *p* is a pointer to an integer quantity

POINTER - INITIALISATION

- Pointer initialization is done with the **syntax**:

pointer = &variable;

Example:

int a;   // Variable declaration

int *p;   // Pointer variable declaration

p = &a;   // Pointer initialization---- stores address of *a* in pointer *p*

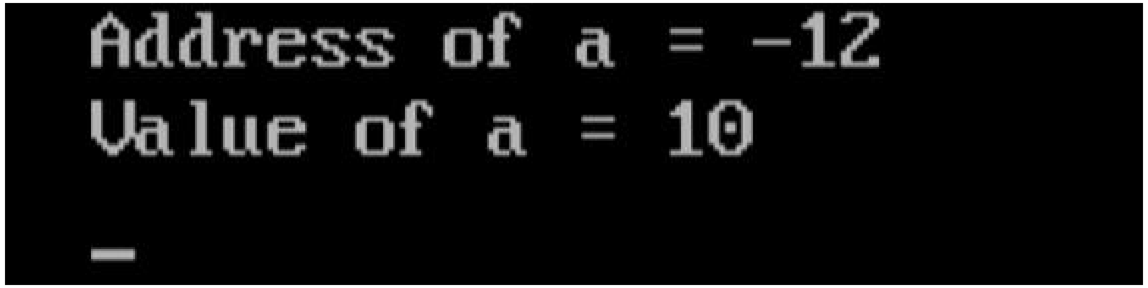
POINTER - INITIALISATION

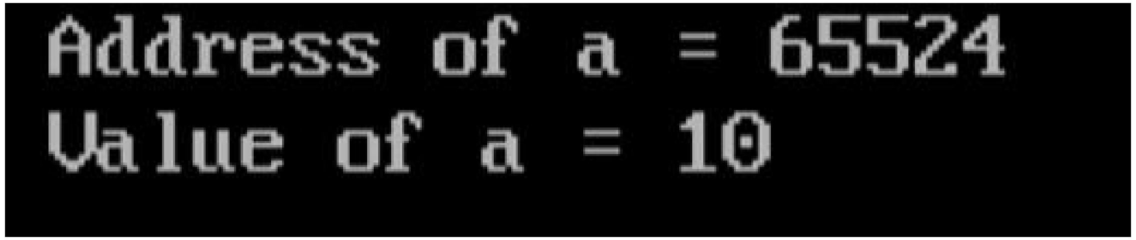
- int a = 10;
- This initialisation reserves a memory space to hold integer value, which stores the value 10 at this location
- p = &a; ----- 5000
- *p = 10 ------ value stored at address

Variable	a
Value	10
Address	5000

POINTERS

```
/* Program to print address of a variable and its value*/
void main()
{
   int a, *p;
   a = 10;
   p = & a;
   printf(" Address of variable, a = %d \n", p);
   printf(" Value in variable, a = %d \n", *p);
}
```





NULL POINTER

- Null pointer is a special constant to initialise pointers that point to nothing
- A null pointer indicates that the pointer does not point to a valid memory location
- We can create a null pointer by assigning null value during the pointer declaration

Example:

```
int *p = NULL;

int *p = 0;
```

Accessing Data Through Pointers

- There are two ways to access value of variables:

1. **Direct access:** We use directly the variable name
2. **Indirect access**: We use a pointer to the variable

Example:

int quantity, *p, n;   // declares integer variables and pointer variable

quantity = 100;   // initialising variable

p = & quantity;   // assigns address of *quantity* to pointer *p*

n = *p   // pointer (*) returns value of variable it points to

Accessing Data Through Pointers

- There are two ways to access value of variables:

1. **Direct access:** We use directly the variable name
2. **Indirect access**: We use a pointer to the variable

Example:

```
int quantity, *p, n;
quantity = 100;
p = & quantity;
n = *p
```

```
Address of quantity = 65524
Value of quantity = 100
Value of p = 65524
Value of n = 100
```

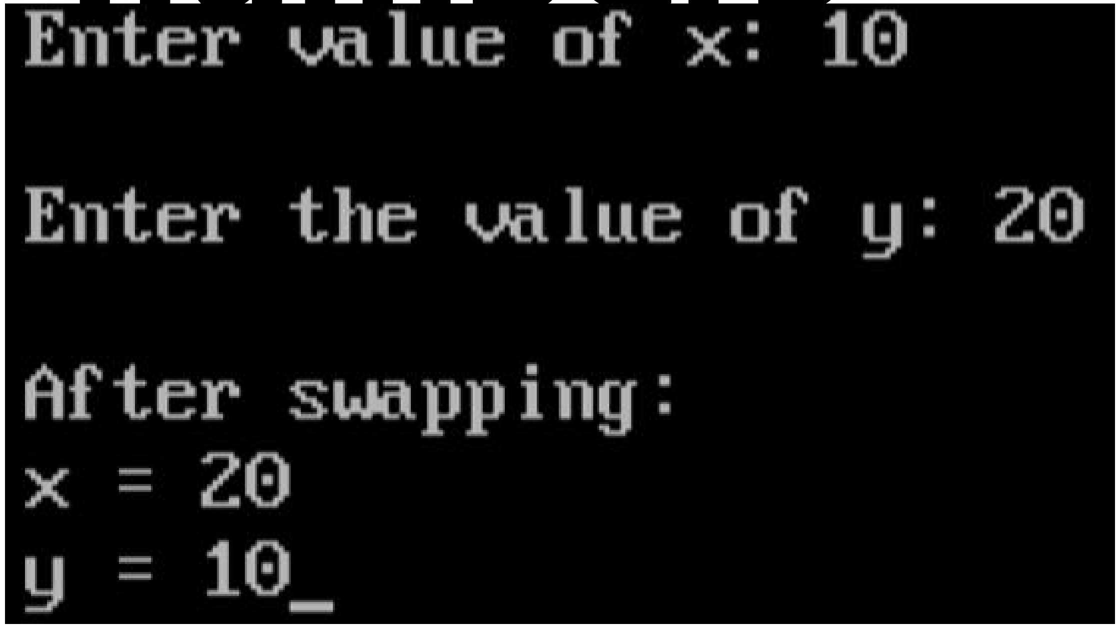
Pointer Arithmetic Operations

```
void main()
{
  int a, b, *p1, *p2, x, y;
  a=12;
  b=4;
  p1 = &a;
  p2 = &b;
  x = *p1 * *p2;
  y = *p1 + *p2;
 printf("X = %d\n", x);
 printf("Y = %d", y);
}
```

```
X = 48
Y = 16
```

Pointer Swapping numbers

```
void main()
{
int x, y, *p, *q;
 p = &x;
 q = &y;
 printf("Enter value of x: ");
 scanf("%d",&x);
 printf("\nEnter the value of y: ");
 scanf("%d", &y);
 int temp = *p;
 *p = *q;
 *q = temp;
 printf("\nAfter swapping:\n");
 printf("x = %d\ny = %d", x,y);
}
```



POINTERS AND ARRAYS

- When an array is declared, compiler allocates a base address and sufficient amount of storage to contain all elements
- Base address is the location of the first element (index 0) of the array
- Example: int x[5] = {1,2,3,4,5};

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

(Base address)

Assuming that each integer requires two bytes

POINTERS AND ARRAYS

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

- **Example: int x[5] = {1,2,3,4,5};** ---------- The name x is defined as a constant pointer, pointing to the first element, x[0]
- Therefore, the value of **x** is 1000, which is the location where x[0] is stored
- x = &x[0] ------ x = address of first element

POINTERS AND ARRAYS

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

- p = x ------If p is declared as a pointer to point to array X
- Therefore, p = &x[0]

p = &x[0] = 1000

p + 1 = &x[1] = 1002

p + 2 = &x[2] = 1004

p + 3 = &x[3] = 1006

p + 4 = &x[4] = 1008

POINTERS AND ARRAYS

```
void main()
{
  int a[5] = {1,2,3,4,5};
  int *p, i;
  p = a;  ----- a means address of a[0]
  printf("Printing array elements using pointer\n");
  for(i=0;i<5;i++)
  {
     printf("\n%d",*p);
     p++;
   }
}
```

OUTPUT:

1
2
3
4
5

```c
void main()
{
int x[5] = {1,2,3,4,5};
 int *p, sum = 0, i=0;
 p = x;  // p = address of x[0]
 printf("Element    Value     Address\n");
 while (i<5)
 {
  printf("x[%d]          %d        %u\n", i, *p, p);
  sum = sum + *p;
  i++;
  p++;
 }
printf("Sum = %d\n",sum);
}
```

```
Element     Value     Address
x[0]          1       65516
x[1]          2       65518
x[2]          3       65520
x[3]          4       65522
x[4]          5       65524
Sum = 15
```

int x[5] = {1, 2, 3, 4, 5};

(i) What is the value of (x + 2)?

(ii) What is the value of * (x+ 2)?

(iii) What is the value of (*x + 2)?

Assuming that each integer requires two bytes

```
Element     Value     Address
x[0]          1       65516
x[1]          2       65518
x[2]          3       65520
x[3]          4       65522
x[4]          5       65524
Sum = 15

value of x = 65516
value of (x+2) = 65520
value of *(x+2) = 3
value of (*x+2) = 3
```

Pointers and 2D Arrays

- Example of 1D array: int x[5] = {1,2,3,4,5};

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5

- The nth element of this array can be accessed by many ways such as:

1. x [n]
2. *(x + n)
3. *(&x[0] + n)

Pointers and 2D Arrays

- 2D arrays can be considered as an array of a number of one-dimensional arrays
- Consider Eg: int a [2][3];
- This can be treated as an array consisting of 2 one dimensional arrays with 3 elements each
- In 2D arrays also, the array name represents the address of its zeroth element
- The address of the 0th element is given by *a* or *a+0*

```
int a;                // Variable declaration
int *p;     // Pointer variable declaration
p = &a;     // Pointer initialization---- stores address of a in pointer p
```

Example:

```
int quantity, *p, n;
quantity = 100;
p = & quantity;
n = *p
```

```
Address of quantity = 65524
Value of quantity = 100
Value of p = 65524
Value of n = 100
```

POINTERS AND 1D ARRAYS

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

- p = x ------If p is declared as a pointer to point to array X
- Therefore, p = &x[0]

p = &x[0] = 1000

p + 1 = &x[1] = 1002

p + 2 = &x[2] = 1004

p + 3 = &x[3] = 1006

p + 4 = &x[4] = 1008

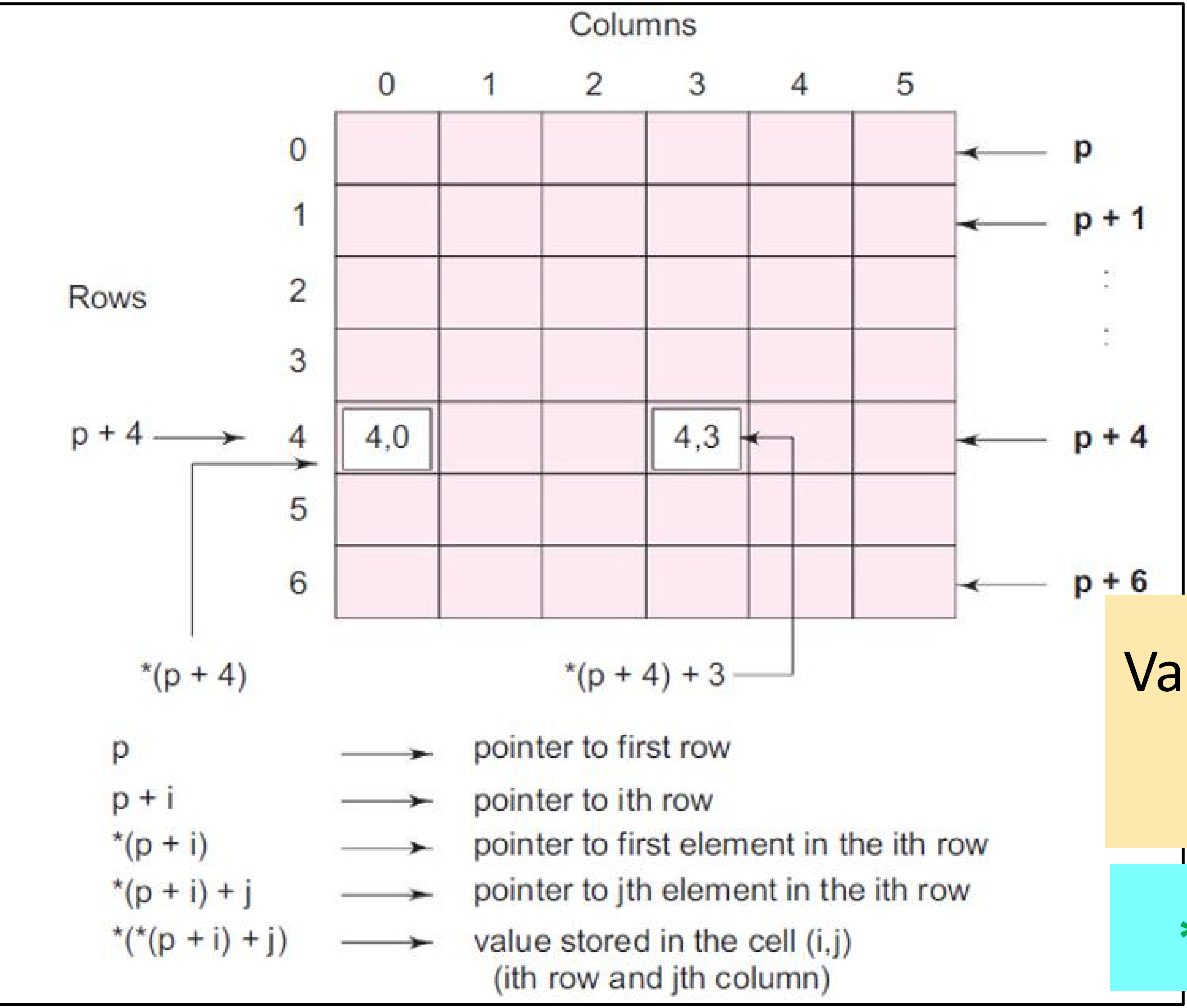
*(p + i) OR *(x + i) represents element x[i]

Read & print a 1D array

```
void main()
{
 int n, arr[20], i, *p;
 p = arr;  // p assigned address of first element
 printf("Enter size of array\n");
 scanf("%d", &n);
 printf("Enter elements of array\n");
 for (i=0;i<n;i++)
    scanf("%d",(p+i));   // (p+i) equivalent to &arr[0].
 printf("Printing array elements using pointer\n");
 for(i=0;i<n;i++)
  {
   printf("\n%d",*p);
   p++;
  }
}
```

```
Enter size of array
3
Enter elements of array
3
4
5
Printing array elements using pointer

3
4
5
```

2D Arrays



Value stored in the cell (i, j) :

*(* (p +i) + j)

* (* (mat + row) + col)

POINTERS AND 2D ARRAYS

An element in a two dimensional array can be represented by:

***(* (p +i) + j) OR *(* (a +i) + j)**

Address of the element:

*** (a +i) + j**

```
/*Program to find sum of two matrices*/
void main()
{
int a[10][10],b[10][10], row, col,i,j;
int s[10][10];
printf("Enter order of matrices\n");
scanf("%d%d",&row, &col);
printf("Enter elements of matrix_a row-wise\n");
for (i=0; i<row; i++)                         //Entering the Matrix
{
  for(j=0; j<col; j++)
  {
    scanf("%d",*(a+i)+j);          //  *(a+i)+j  is equivalent to &a[i][j]
  }
}
```

```c
for (i=0; i<row; i++)                         // Finding sum of matrices
{
   for(j=0; j<col; j++)
   {
      *(*(s+i)+j) = *(*(a+i)+j) + *(*(b+i)+j);    // *(*(s+i)+j) is equivalent to s[i][j]

   }
}
for (i=0;i<row;i++)                           // PRINTING SUM
{
   for(j=0;j<col;j++)
   {
      printf("%4d", *(*(s+i)+j) );
   }
printf("\n");
}
```

- If `p = &a[0]`, then:

```
*p        is an  alias  for  a[0]
*(p+0)    is also an  alias  for  a[0]

*(p+1)    is also an  alias  for  a[1]
*(p+2)    is also an  alias  for  a[2]
```

```
int a[10] = { 11, 22, 33, 44, 55, 66, 77, 88, 99, 777 };
int *p;

p = &a[0];                          // p points to variable a[0]

printf("*p        = %d, a[%d] = %d\n", *p,     0, a[0] );
printf("*(p + 1) = %d, a[%d] = %d\n", *(p+1), 1, a[1] );
printf("*(p + 2) = %d, a[%d] = %d\n", *(p+2), 2, a[2] );
printf("*(p + 3) = %d, a[%d] = %d\n", *(p+3), 3, a[3] );
```

```
*p       = 11, a[0] = 11
*(p + 1) = 22, a[1] = 22
*(p + 2) = 33, a[2] = 33
*(p + 3) = 44, a[3] = 44
```

```
int a;              // Variable declaration
int *p;     // Pointer variable declaration
p = &a;     // Pointer initialization---- stores address of a in pointer p
```

POINTERS AND 1D ARRAYS

Elements	X[0]	X[1]	X[2]	X[3]	X[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

- p = x ------If p is declared as a pointer to point to array X
- Therefore, p = &x[0]

p = &x[0] = 1000

p + 1 = &x[1] = 1002

p + 2 = &x[2] = 1004

p + 3 = &x[3] = 1006

p + 4 = &x[4] = 1008

*(p + i) OR *(x + i) represents element x[i]

POINTERS AND 2D ARRAYS

An element in a two dimensional array can be represented by:

***(* (p +i) + j) OR *(* (a +i) + j)**

Address of the element:

*** (a +i) + j**

Pointers and Strings

- Strings are character arrays and are declared and initialised as:

- char str [6] = “hello”; ------ other formats discussed in module 3

Element	str[0]	str[1]	str[2]	str[3]	str[4]	str[5]
Value	h	e	l	l	o	\0
Address	1000	1001	1002	1003	1004	1005

The variable name of the string str holds the address of the first element of the array

Pointers and Strings

Index	0	1	2	3
Value	P	I	C	\0

```
void main()
{
 char name[4] = "PIC";     // declaring string
 char *ptr;                // declaring pointer
 ptr = name;               // initialising pointer
 while (*ptr != '\0')
 {
  printf("%c",*ptr);
  ptr++;
 }
}
```

OUTPUT

PIC

Pointers and Strings

Printing the contents of string:

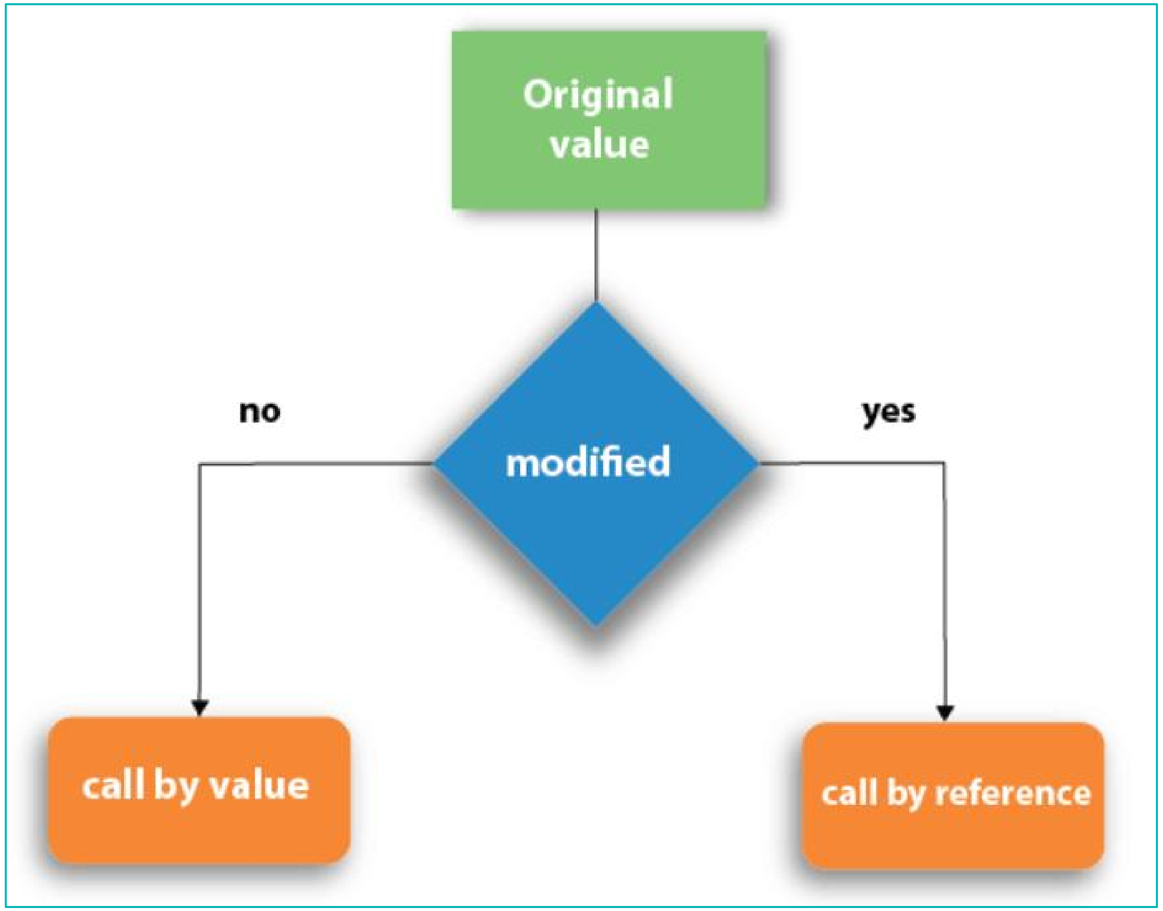
1. printf(“%s”, str);
2. puts (str);

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  char *name;
  name = "MACE";
  puts(name);
  getch();
}
```

Call by Reference

- Pointers can be passed to functions
- Address of a variable can be passed to the function as arguments
- Arguments are to be declared as pointer type
- When an address is passed to a function, the parameters receiving the address should be pointer

The process of calling a function using pointers to pass the address of variables is called **call by reference**



- **<u>In call by value method,</u>** we can not modify the value of the actual parameter by the formal parameter.

Call by Reference

```
void main()
{
  void add (int*);
  int x = 10;
  printf("Before function call, x = %d\n", x);
  add (& x);  // call by reference or address
  printf("After function call, x = %d", x);
  getch();
}
void add (int *p)
{
*p = *p + 10;    // add 10 to the value stored at address p
}
```

```
Before function call, x = 10
After function call, x = 20_
```

p is the address of the variable x,

**p holds the value at the address p*

```
void swap(int *a, int *b);
main()
{
  int m = 10, n = 20;
  printf("m = %d\n", m);
  printf("n = %d\n\n", n);
  swap(&m, &n);                    //passing address of m and n to the swap function
printf("After Swapping:\n\n");
printf("m = %d\n", m);
printf("n = %d", n);        }
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

```c
void main()
{
 void swap_p (int*, int*);
 void swap (int, int);
 int x = 10, y = 20;
 printf("\nValue of x and Y:\n\n");
 printf("X = %d\tY = %d\n\n", x,y);
 printf("After call by value:\n\n");
 swap(x,y);
 printf("X = %d\tY = %d\n\n", x,y);
 printf("After call by reference:\n\n");
 swap_p(&x,&y);
 printf("X = %d\tY = %d\n\n", x,y);      }
```

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

```
Value of x and Y:

X = 10  Y = 20

After call by value:

a = 20  b = 10

X = 10  Y = 20

After call by reference:

X = 20  Y = 10
```

```
void main()
{
 void swap_p (int*, int*);
 void swap (int, int);
 int x = 10, y = 20;
 printf("\nValue of x and Y:\n\n");
 printf("X = %d\tY = %d\n\n", x,y);
 printf("After call by value:\n\n");
 swap(x,y);
 printf("X = %d\tY = %d\n\n", x,y);
 printf("After call by reference:\n\n");
 swap_p(&x,&y);
 printf("X = %d\tY = %d\n\n", x,y);      }
```

```
void swap_p (int *a, int *b)
{
 int temp = *a;
 *a = *b;
 *b = temp;
}
```

```
Value of x and Y:

X = 10  Y = 20

After call by value:

a = 20  b = 10

X = 10  Y = 20

After call by reference:

X = 20  Y = 10
```

```c
void main()
{
int sum_array(int*, int);
 int arr[20], n, i, *p, sum;
 p = arr;
 printf("Enter size of array\n");
 scanf("%d", &n);
 printf("Enter the elements of array\n");
 for (i=0; i<n; i++)
  scanf("%d", &arr[i]);
 sum = sum_array(arr,n);
 printf("Sum = %d", sum);
 getch();
}
```

```c
int sum_array(int *array, int size)
{
 int s = 0, j;
 for (j = 0; j < size; j++)
 {
  s = s + *array;
  array++;
 }
 return (s);
}
```

THANK YOU