COMPUTER PROGRAMMING

Year   : **2016 - 2017**
Subject Code   : **ACS001**
Regulations   : **R16**
Class   : **I B.Tech**
Branch   : **CSE/ ECE/ IT/ EEE**

Team of Instructors

- Dr. K Srinivasa Reddy, Professor, CSE

- Dr. G Ramu, Professor, CSE

- Ms. B Padmaja, Associate Professor, CSE

- Ms. P. Ila Chandana Kumari, Associate Professor, IT

- Ms. K. Laxmi Narayanamma, Associate Professor, IT

- Ms. B Rekha, Assistant Professor, IT

Prepared By

Ms. K. Laxmi Narayanamma, Associate Professor, IT

Ms. B Rekha, Assistant Professor, IT

Text Books

1.Stephen G. Kochan, "Programming in C", Addison-Wesley Professional, 4th Edition,2014.

2.B. A. Forouzan, R. F. Gillberg, "C Programming and Data Structures", CengageLearning, India, 3rdEdition, 2014.

UNIT-1
INTRODUCTION TO COMPUTERS

# **Introduction to Computers**

<u>Objectives:</u>

- To review basic computer systems concepts
- To be able to understand the different computing environments and their components
- To review the history of computer languages
- To be able to list and describe the classifications of computer languages
- To understand the steps in the development of a computer program
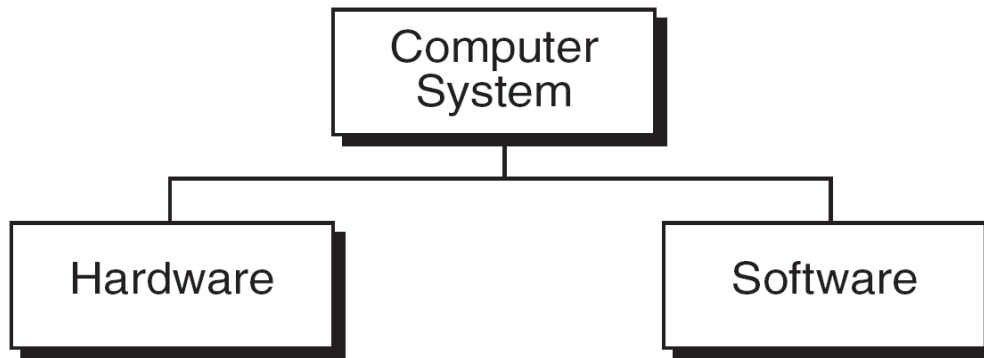- To review the system development life cycle

- **What is a Computer?**

  A **COMPUTER** is an electronic device that can:

  Receive information,Perform processes,Produce output and Store information for future use.
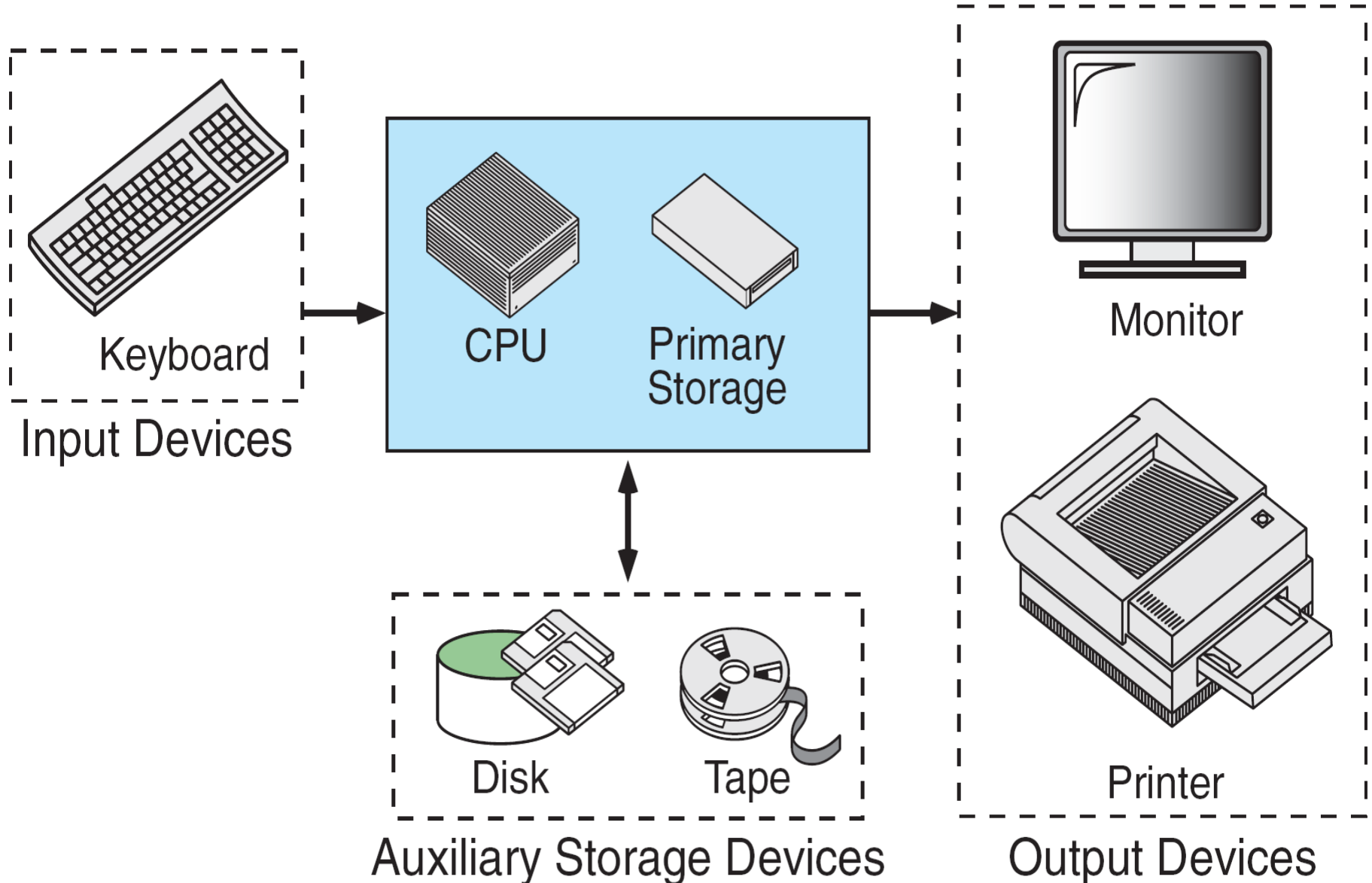
INPUT	→	CPU(Processor)	→	OUTPUT
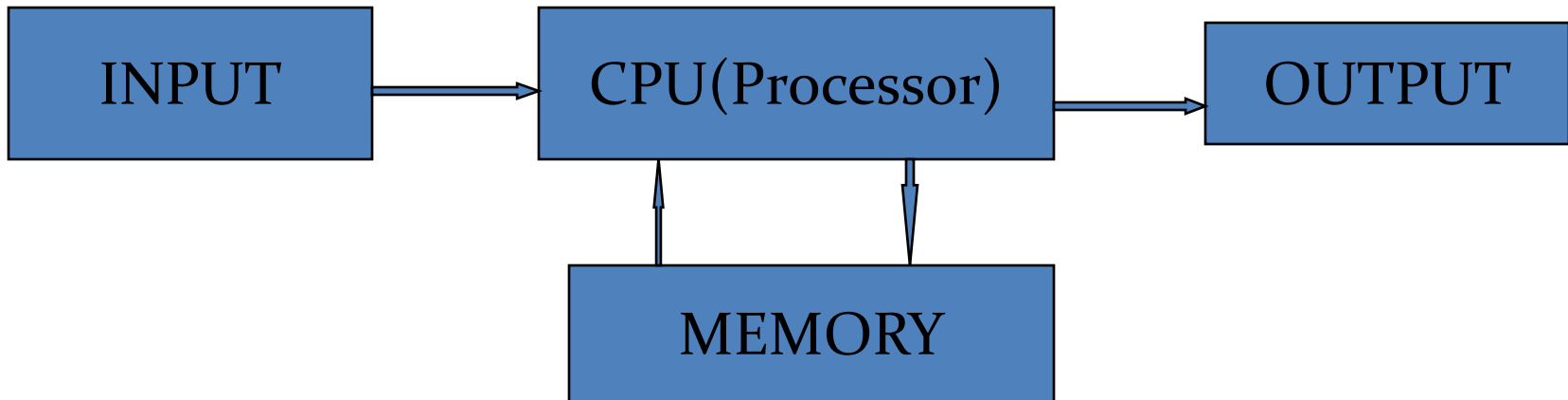
MEMORY

Fig: Information Processing Cycle

- *A computer system made of two major components: hardware and software.*
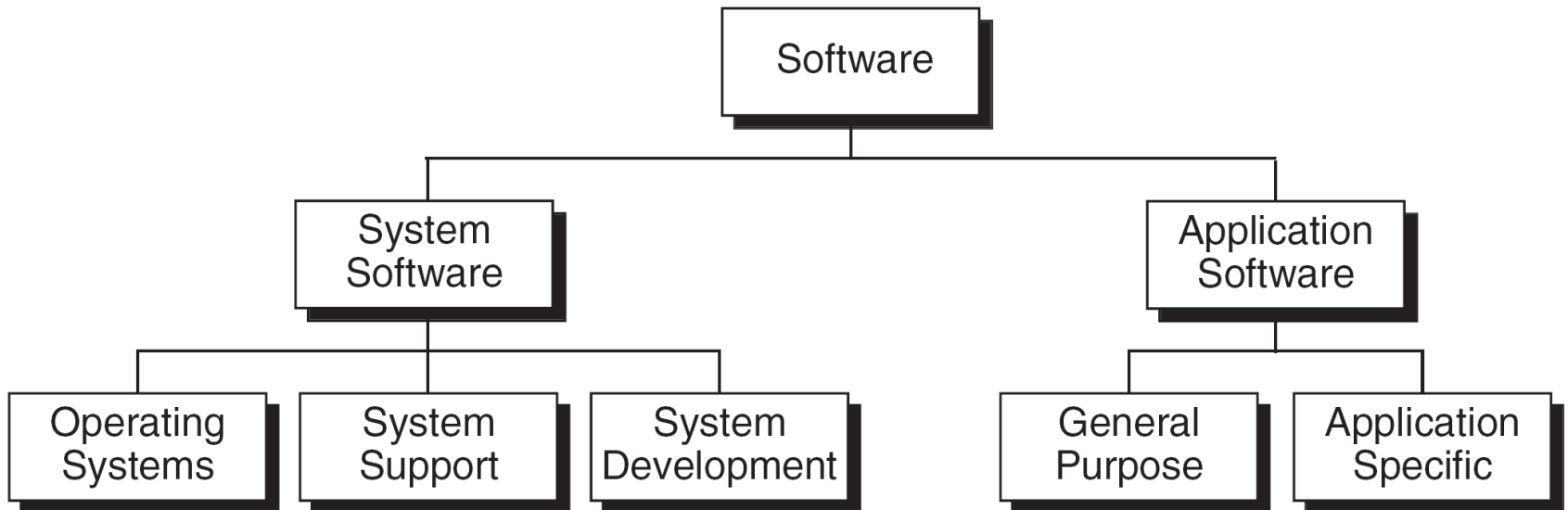

- **Hardware - the physical parts that make up the computer.**

    Eg: CPU, memory, disks, CD-ROM drives, printer.

- **Software - computer programs and applications.**

    Eg: Operating system, word processor, games, etc.

Basic Hardware Components

Keyboard

Input Devices

CPU

Primary Storage

Monitor

Disk

Tape

Auxiliary Storage Devices

Printer

Output Devices

Software:

Types of Software:

Software
├── System Software
│   ├── Operating Systems
│   ├── System Support
│   └── System Development
└── Application Software
   ├── General Purpose
    └── Application Specific

System Software

- Operating system is a Software component that provides an interface between user and system hardware.

  Ex: Windows ,Linux,Unix,Solaris etc.

- System support Software provides system utilities and other operating services.

  Eg: Sort programs, Formatting programs, Linkers, Loaders

- System Development software includes the language translators (Compilers, Assemblers etc.) that convert programs into machine language for execution, debugging tools to ensure that the programs are error-free and CASE tools for software engineering processes

  Eg: C Compiler,Java Compiler

ApplicationSoftware

- Application specific software can be used for a specific intended purpose

  Ex: Pay roll, Inventory Management, Library management etc.

- General Purpose software is intended for use in more than one application

  Ex: Word Processors, Database Management systems and Computer-   aided Design Systems.
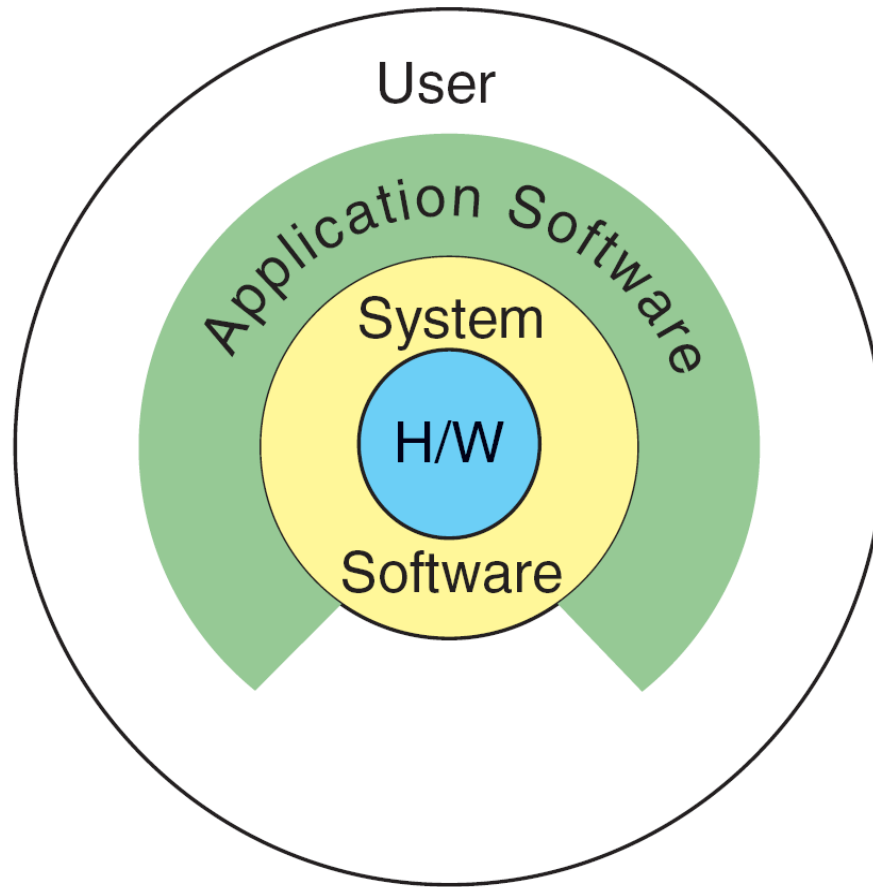
Relation Ship Between System and Application Software

Components of a Computer System:

- The Primary Components of a Computer System are

  1) **Input devices.**

  2) Central Processing Unit

  3) Memory.

  4) Output devices.

- Input Devices: Input devices are Hardware Components that accepts the input from the User.

  Ex: Keyboard , Mouse, Scanner, Microphone etc.

- CPU:(CU+ALU):

  - The central processing unit (CPU) is the "brain" of the computer.

  - It performs a large number of operations at a high speed.

  - Control Unit Interprets Instructions to the Computer.

  - ALU Performs the Arithmetic and logical Operations

  Ex: Intel Pentium, Motorola, IBM RISC.
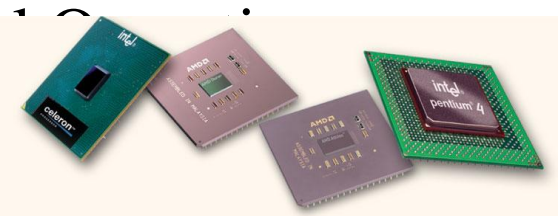
- **Memory**: The function of Memory is Storing the data and Instructions.

  - Memory is Divided into two types :

      1) Primary Memory(RAM)

      2) Secondary Memory(ROM)

  - Primary Memory is a Volatile Memory that is when the

      power loss the data stored in the Memory lost.

      Ex: RAM(Random Access Memory).

  - Secondary Memory is a Non-Volatile Memory that is Even if the power loss it holds the data.

      Ex: Harddisk,CD-ROM,DVD-ROM,Floppy Flash Memory etc.

<u>Memory Data Representation :</u>

- Data in memory is stored as binary digits (BITS) e.g. 011100101010
- 1 BYTE = 8 bits
- 1 byte usually stores 1 text character.

- The size of memory is measured in terms of how many bytes it can hold.
  - 1 kilobyte = 2^{10} bytes = 1024 bytes
  - 1 megabyte = 2^{20} bytes = ~1 million bytes
  - 1 gigabyte = 2^{30} bytes = ~1 billion bytes
  - 1 terabyte = 2^{40} bytes = ~1 trillion bytes

- One megabyte can hold approximately 500 pages of text information.

Output devices

- Output devices make the information resulting from the processing available for usage.

    - printer - produces a hard copy of your

       output

    - screen - produces a visual display of your output for browsing

    - speakers, etc.
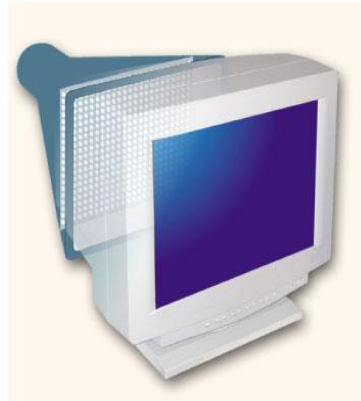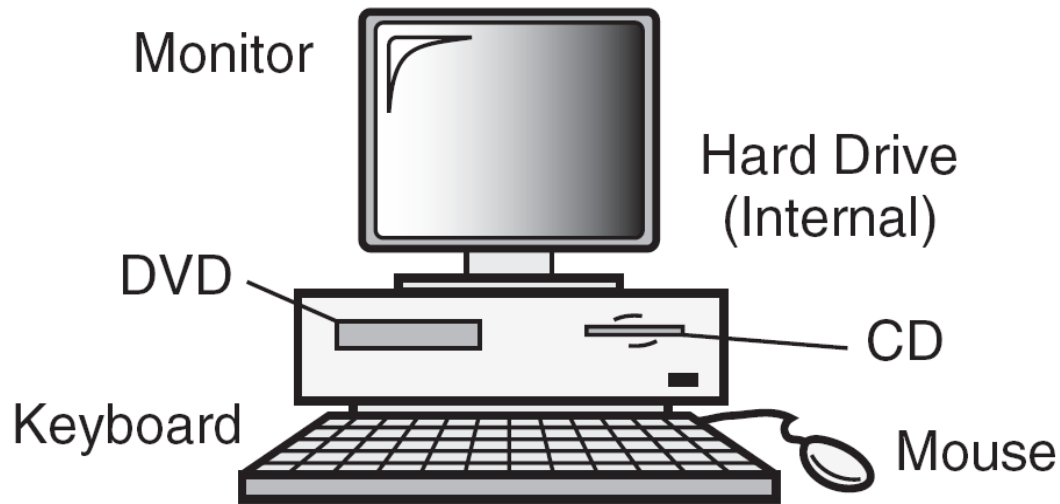
Computing Environments

There are four computing environments:

1) Personal Computing Environment
2) Time-Sharing Environment
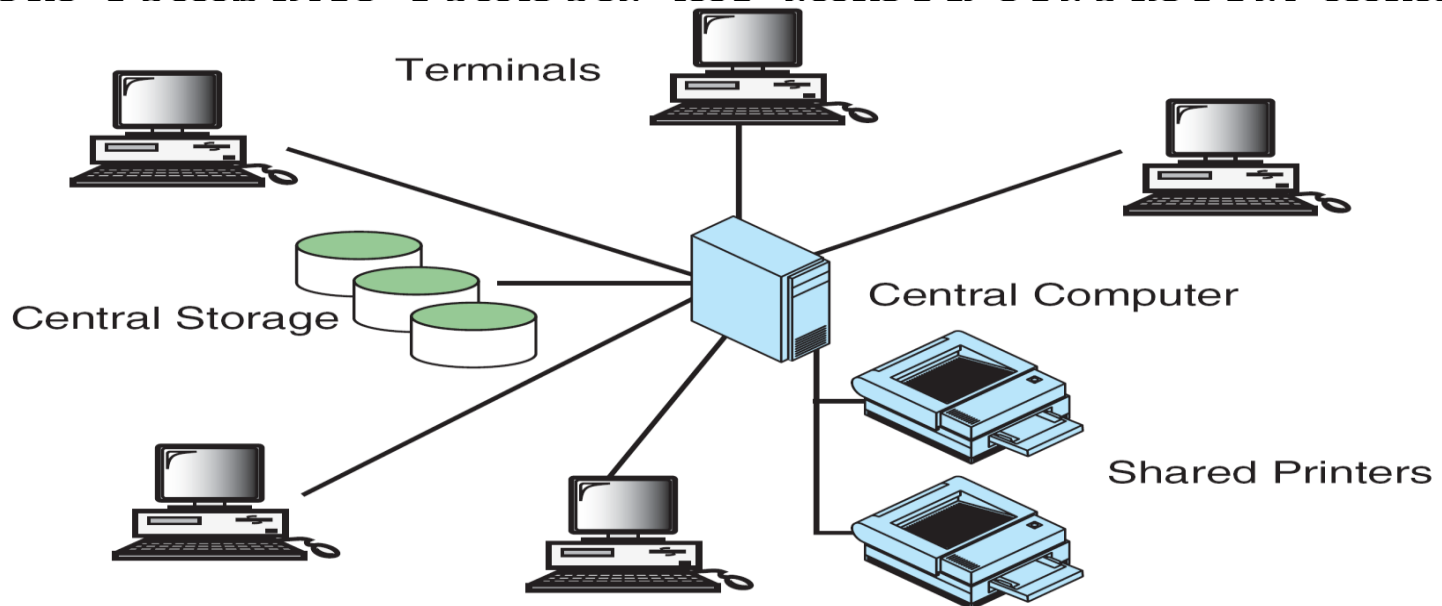3) Client/Server Environment
4) Distributed Computing

Personal Computing Environment

- All of the computer hardware components are tied together in the personal computer. The whole computer is available to the user.

Monitor

Hard Drive (Internal)

DVD

CD

Keyboard

Mouse

Time-Sharing Environment

- Many users are connected to a computer system. The terminals used are often non-programmable.

  – The output devices and auxiliary storage devices are shared by all of the users.

  – In the time-sharing environment, all computing is done by the central computer.

  – Central computer controls the shared resources, manages the

Terminals

Central Storage

Central Computer

Shared Printers

Client/Server Environment

- A client /server computing environment splits the computing function between a central computer and users' computers.

- Some of the computational responsibility is moved from the central computer and assigned to the client computers.

- The central computer called server that manages the shared data and does some part of the computing

- Because of the work sharing, the response time and monitor display are faster and the users are more productive.
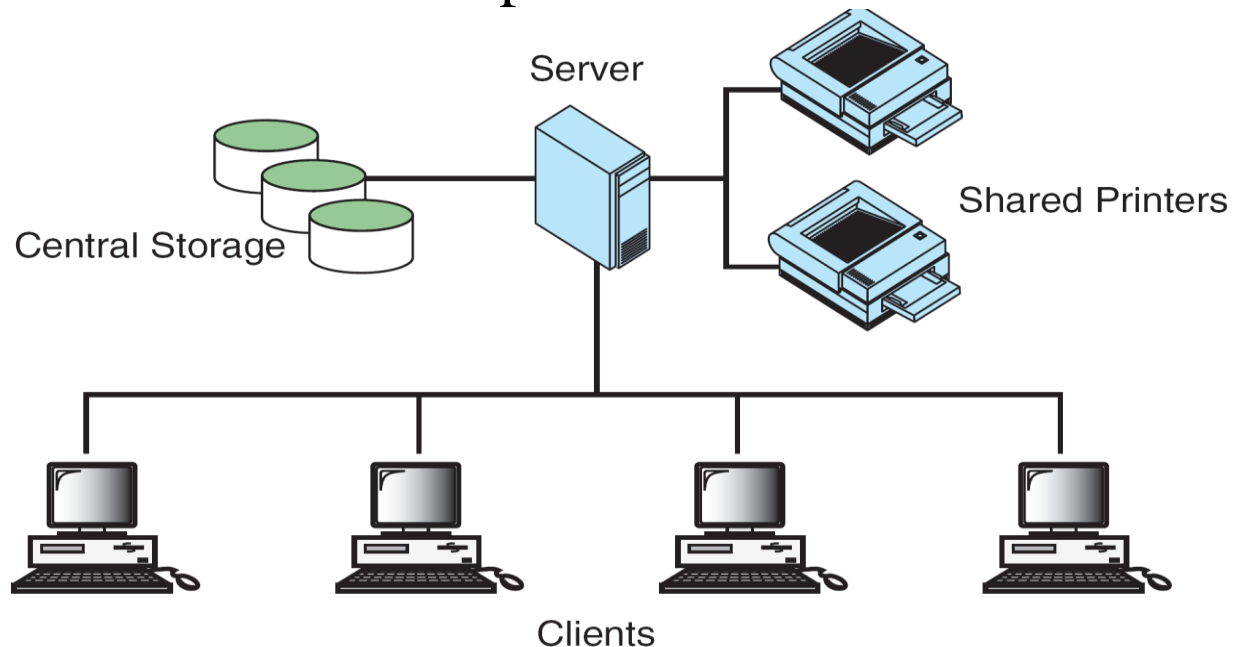
Distributed Computing

- A distributed computing environment provides integration of computing functions between different clients and servers.

- Distributed computing utilizes a network of many computers, each accomplishing a portion of an overall task, to achieve a computational result much more quickly than with a single computer.
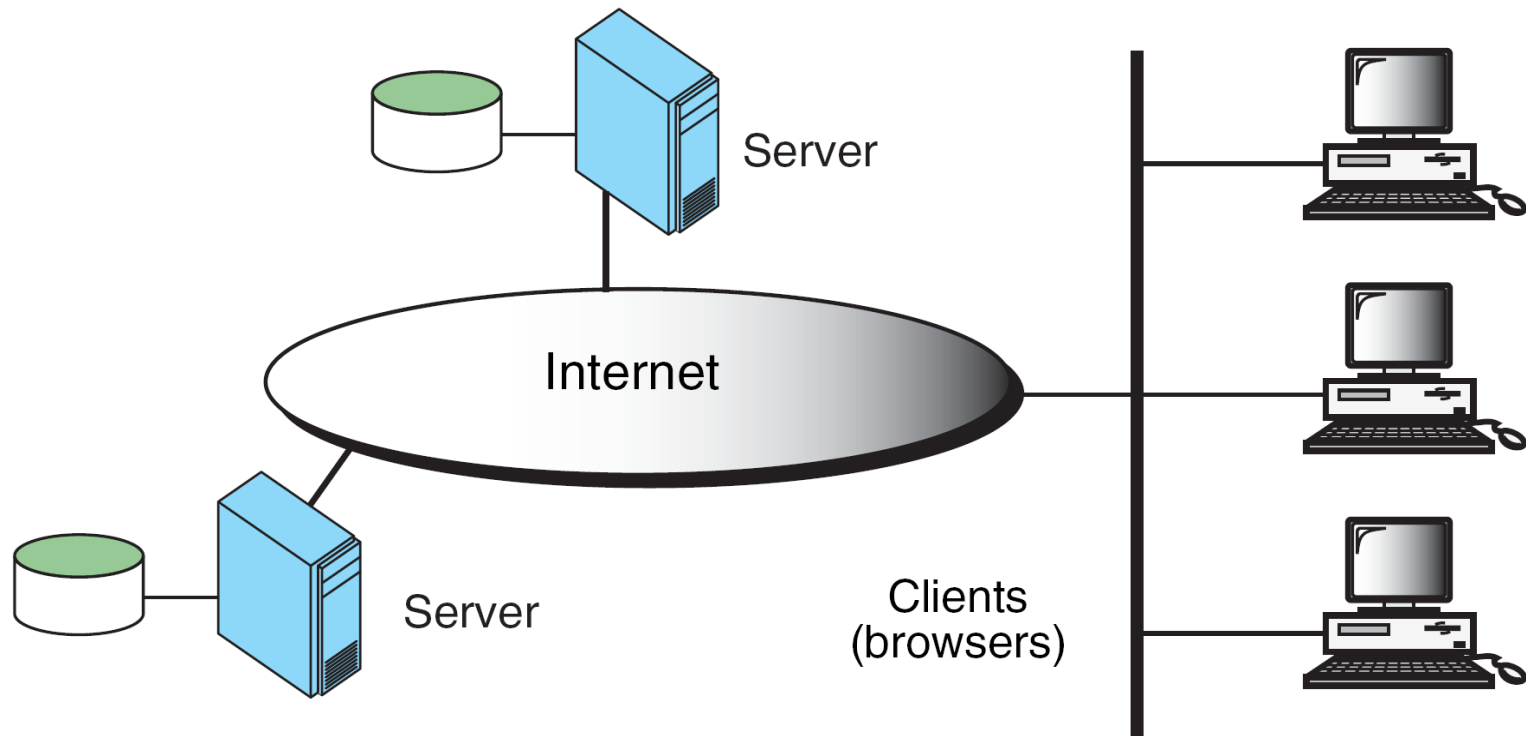
Computer Languages

- To write a program for a computer, we must use a computer language.

  - There are three types of Computer Languages

    1) Machine Languages

    2) Symbolic Languages
    3) High-Level Languages
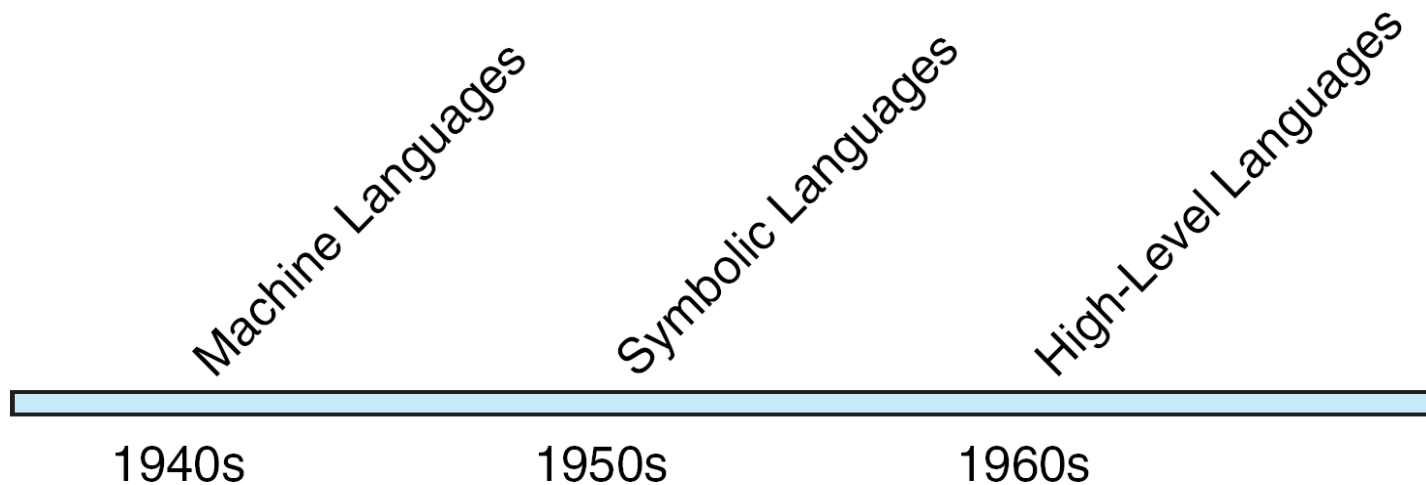
Machine Languages

- Each Computer has its own machine language which is made up of 0''s and 1's.

- The instructions in machine language must be in streams of 0's and 1's because internal circuits of a computer are made of devices that can be in one of two states: on or off.

- Machine language is highly difficult to program as it needs the complete knowledge of the computer's instructions

- Difficult and Rarely used

- NOTE:

  The only language understood by computer hardware is machine language.

Symbolic Languages

- The computer operations are represented in symbols or mnemonics to represent various machine language instructions .

    Ex: Add   instead of   0001

- These languages are also called Assembly Languages.

- Each assembly language instruction translates to one machine language instruction

- Programming is easier in Assembly language compared to developing programs in machine language

- Symbolic languages are machine dependent

- Assemblers convert the assembly language instructions to machine language instructions

High-level Languages

- The need to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level languages

- High level languages are portable to many different computers, allowing the programmers to concentrate on the application problem rather than on the computer

- Compilers are used to convert high-level language programs to machine language programs

- The process of converting high-level language into machine language is called compilation.

- One high-level language statement can get converted to one or more machine language statements

- **Example**

   A Statement A= A+B in Different Languages

Machine Language	Assembly Language	High Level Language
0000 0000	CLA	A= A+B
0001 0101	ADD A	
0001 0110	ADD B	
0011 0101	STA A	

Creating and Running Programs

- The steps involved in Creating and Running Programs are:

  1) Writing and Editing Programs
  2) Compiling Programs
  3) Linking Programs
  4) Executing Programs

Writing and Editing Programs:

- To solve a particular problem a Program has to be created as a file using text editor / word processor. This is called source file.

- The program has to be written as per the structure and rules defined by the high-level language that is used for writing the program (C, JAVA etc).

Compiling Programs:

- The compiler corresponding to the high-level language will scan the source file, checks the program for the correct grammar (syntax) rules of the language.

- If the program is syntactically correct, the compiler generates an output file called 'Object File' which will be in a binary format and consists of machine language instructions corresponding to the computer on which the program gets executed.

- If the source program contains syntax errors, the compiler lists these errors and will not generate the object file. The program is to be corrected for the errors and recompiled.

- The object file contains references to other programs which will be needed for the execution of the program. These programs are called library functions. These programs are to be combined with the Object File.

Linking Programs:

- Linker program combines the Object File with the required library functions to produce another file called " executable file". Object file will be the input to the linker program.

- The executable file is created on disk. This file has to be put into (loaded) the memory.

Executing Programs:

- Loader program loads the executable file from disk into the memory and directs the CPU to start execution.

- The CPU will start execution of the program that is loaded into the memory .

- During Program Execution, the program reads data for processing, either from the user (key-board) or from a file.After the program processes the data, it prepares the output. Output can be to the user's monitor or to a file.

- When the program has finished its job, it informs the Operating System.OS then removes the program from memory.
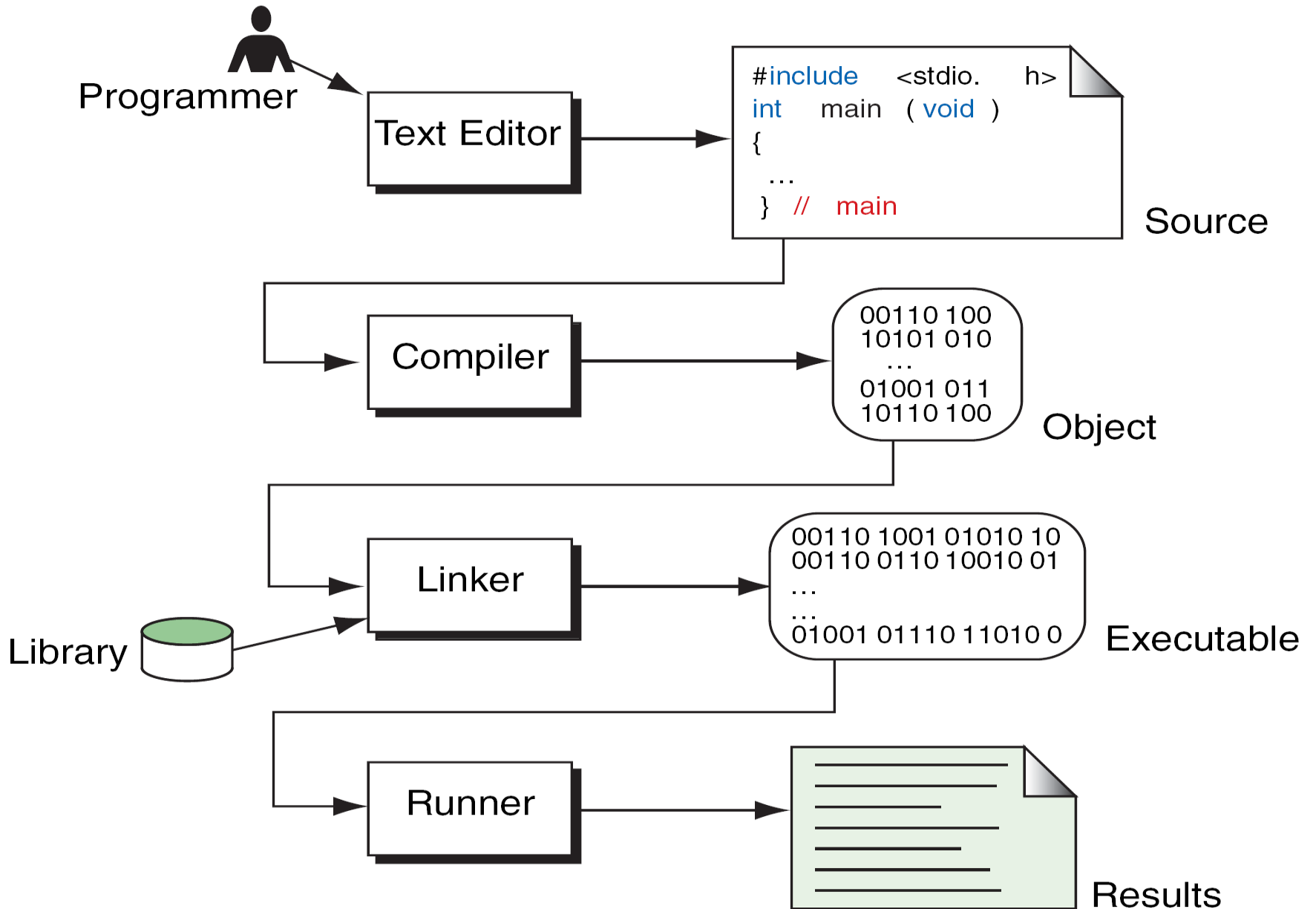
Fig: Building a C Program

Algorithm

– Precise step-by-step plan for a computational procedure that begins with an input value and yields an output value in a finite number of steps.

– It is an effective method which uses a list of well-defined instructions to complete a task, starting from a given initial state to achieve the desired end state.

– An algorithm is written in simple English and is not a formal document.

– An algorithm must:
  - Be lucid, precise and unambiguous
  - Give the correct solution in all cases
  - Eventually end

- it is important to use indentation when writing solution in algorithm because it helps to differentiate between the different control structures.

Instead of

Read n;for i=1 to n add all values of A[i] in sum;Print sum/n;

Write

Read n;

For i=1 to n add all values of A[i] in sum;

Print sum/n;

is more readable and easy to understand.

Properties of algorithms

1) Finiteness:

- an algorithm terminates after a finite numbers of steps.

2) Definiteness:

- each step in an algorithm is unambiguous. This means that the action specified by the step cannot be interpreted in multiple ways & can be performed without any confusion.

3) Input:

- An algorithm accepts zero or more inputs.
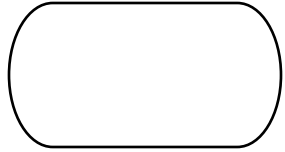
4) Output:

- It produces at least one output.

5) Effectiveness:

- It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

Flowcharts

– A flowchart is a type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a problem.

– Data is represented in the boxes, and arrows connecting them represent direction of flow of data.

– Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields .
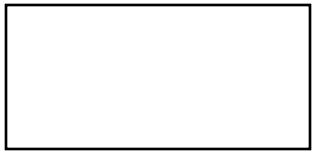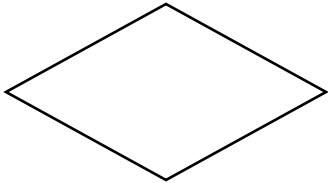
Common Flowchart Symbols:

Terminator: Shows the starting and ending points of a program
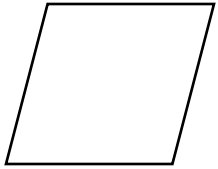
Data Input or output: Allows the user to input data or to display the results .

Processing: Indicates an operation performed by the computer, such as a variable Assignment or mathematical operation
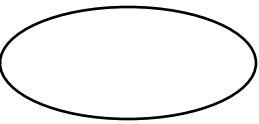
Decision: A diamond has two flow lines going out. One is labeled as 'Yes' Branch and the other as 'no' branch.
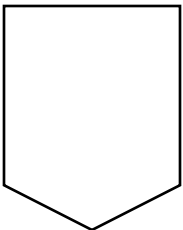
- **Common Flowchart Symbols:**

Predefined Process. Denotes a group of previously defined statements. Ex: "Calculate m!" Program executes the necessary Commands to compute m factoria**l**

Connector. Connectors avoid crossing flowlines, Connectors come in pairs, one with a flowline in and the other with a flowline out.

Off Page Connector: Come in Pairs, Extends Flow charts to more than a page

Flowline. Flowlines connect the flowchart symbols and show the sequence of operations during the program execution.
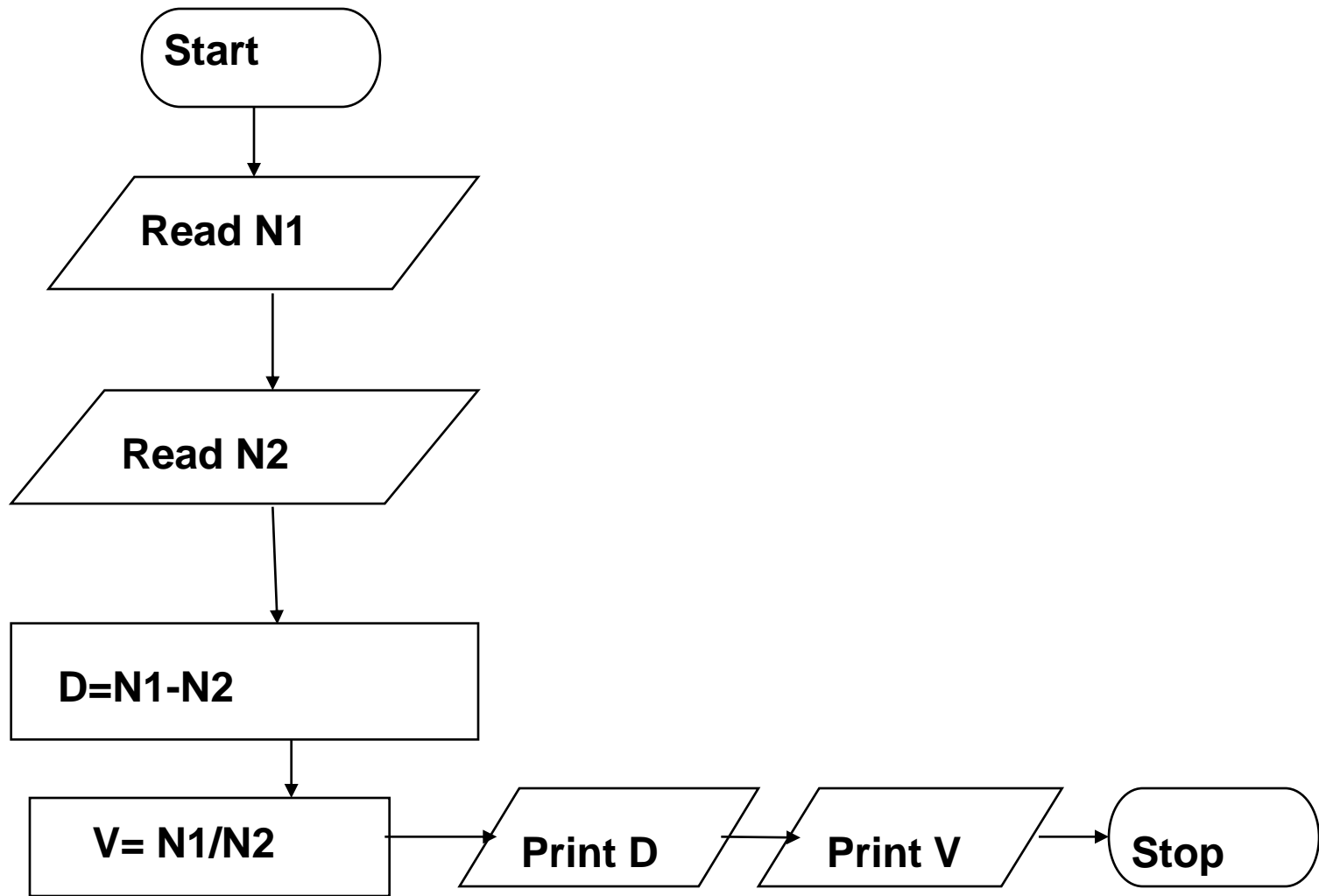
Examples

- Example 1: Finding the sum of two numbers.
- – Variables:
  - – • A: First Number
  - – • B: Second Number
  - – • C: Sum (A+B)
- – Algorithm:
  - – • Step 1 – Start
  - – • Step 2 – Input A
  - – • Step 3 – Input B
  - – • Step 4 – Calculate C = A + B
  - – • Step 5 – Output C
  - – • Step 6 – Stop

Flowcharts:
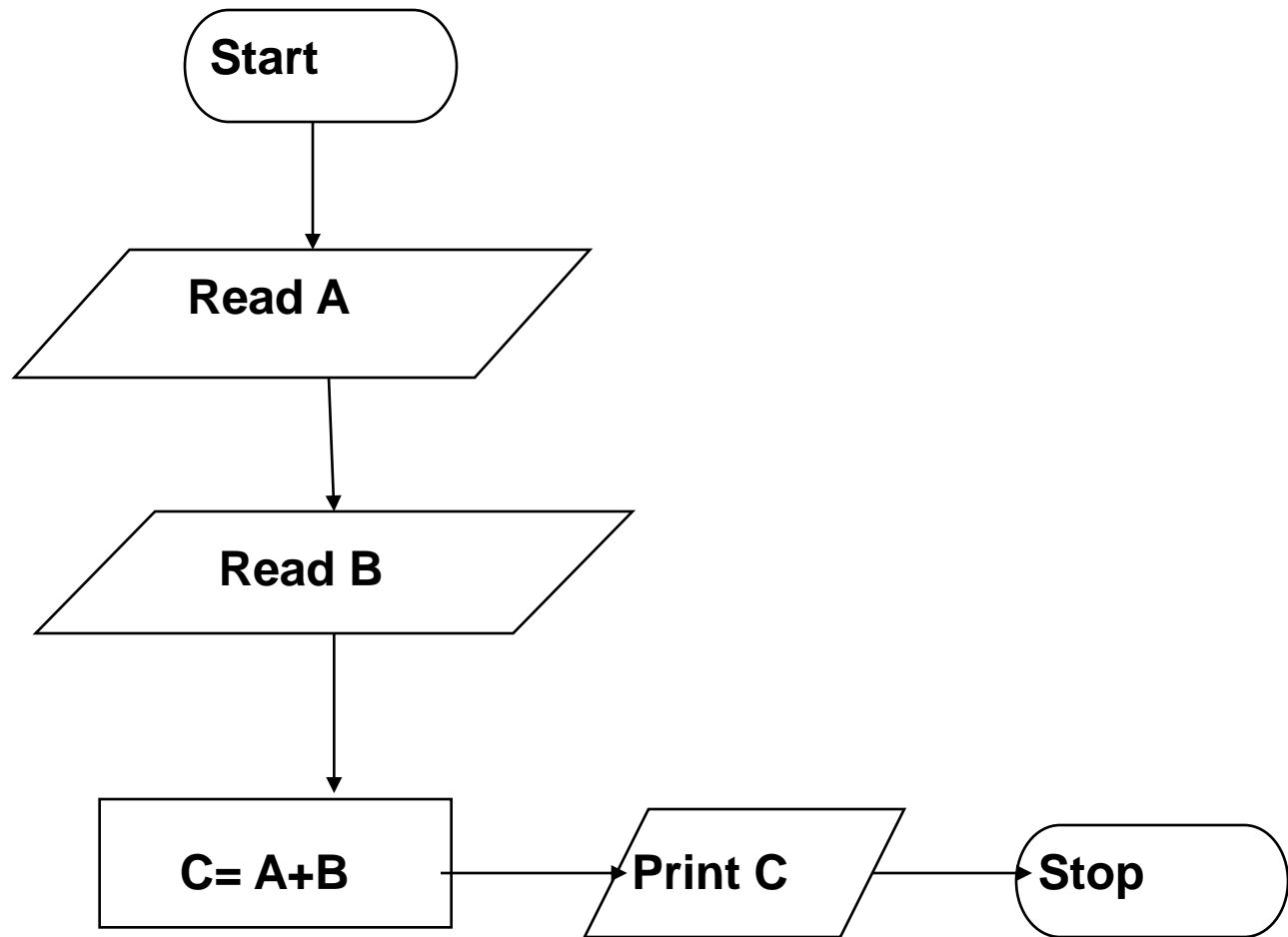
- **Example 2**: Find the difference and the division of two numbers and display the results.

- – Variables: -- Algorithm:

  • N1: First number   * Step 1: Start

  • N2: Second number   * Step 2: Input N1

  • D : Difference   * Step 3: Input N2

  • V : Division   * Step 4: D=N1–N2

                                         * Step 5: V=N1 /N2

                                         * Step 6: Output D

                                         * Step 7: Output V

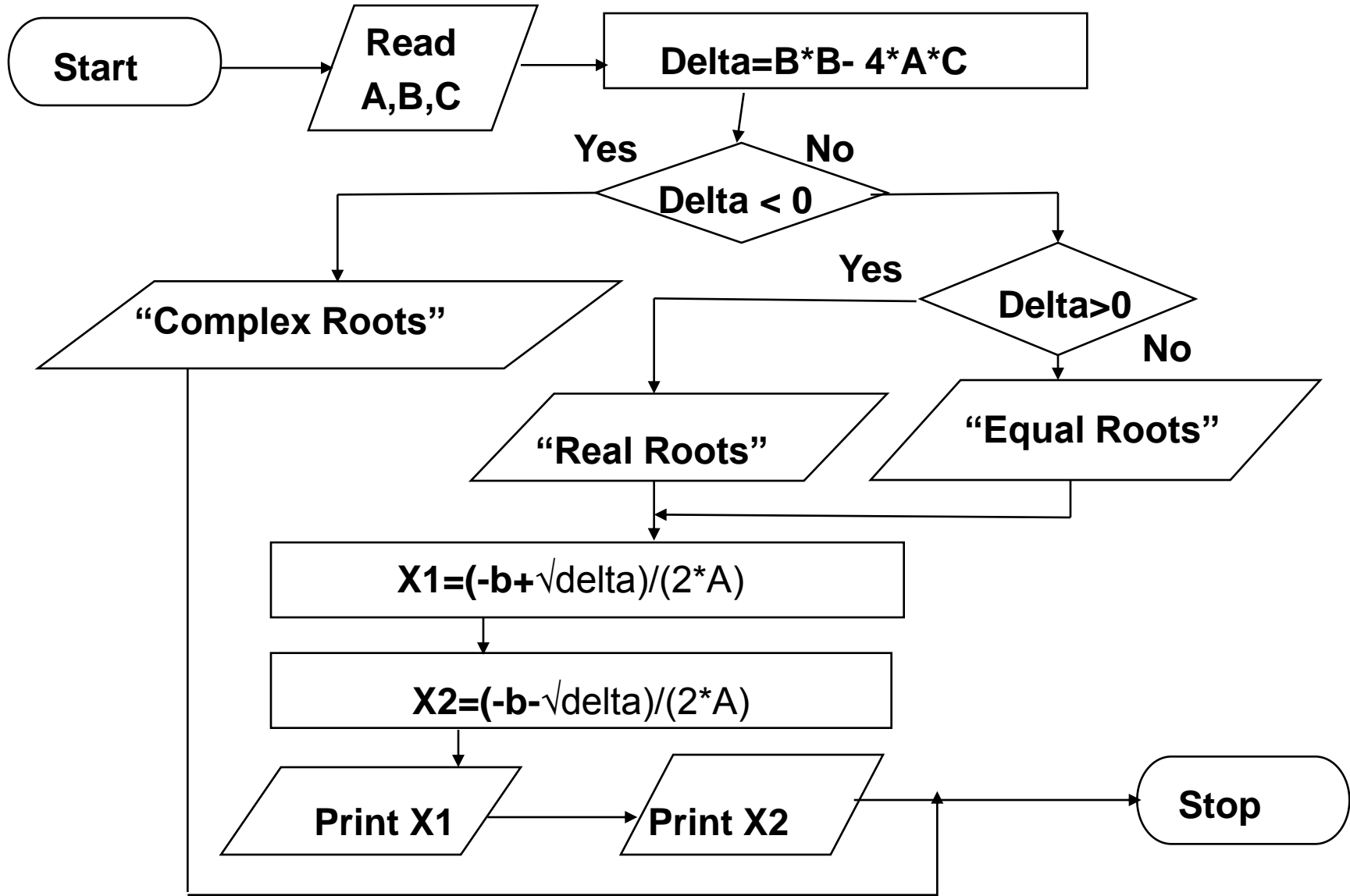   * Step 8: Stop

Flow Chart

Example 3:

Work on the algorithm and the flow chart of the problem of calculating the roots of the equation $Ax^2 + Bx + C = 0$

• Variables:

- A: Coefficient of X^2
- B: Coefficient of X
- C: Constant term
- delta: Discriminant of the equation
- X1: First root of the equation
- X2: Second root of the equation

- Algorithm:
  - Step 1: Start
  - Step 2: Input A, B and C
  - Step 3: Calculate delta = $B^2 - 4AC$
  - Step 4:
    If delta<0 go to step 6, otherwise go to 5
  - Step 5:
    If delta>0 go to step 7, otherwise go to 8
  - Step 6:
    Output "complex roots". Go to step 13
  - Step 7: Output "real roots". Go to step 9
  - Step 8:
    Output "equal roots". Go to step 9
  - Step 9: Calculate X1=(-b+√delta)/(2A)
  - Step 10: Calculate X2=(-b-√delta)/(2A)
  - Step 11: Output X1
  - Step 12: Output X2
  - Step 13: Stop

Flowcharts:

Start → Read A,B,C → Delta=B*B- 4*A*C

Delta < 0
- Yes → "Complex Roots"
- No → Delta>0
  - Yes → "Real Roots"
  - No → "Equal Roots"

X1=(-b+√delta)/(2*A)

X2=(-b-√delta)/(2*A)

Print X1 → Print X2 → Stop

Intoduction to C :

- A high-level programming language developed in 1972 by Dennis Ritchie at AT&T Bell Labs.
- C is a structured Language
- It was designed as a language to develop UNIX operating system
- American National Standards Institute (ANSI) approved first version of C in 1989 which called C89
- Major changes were made in 1995 and is known as C95
- Another update was made in 1999 and is known as C99
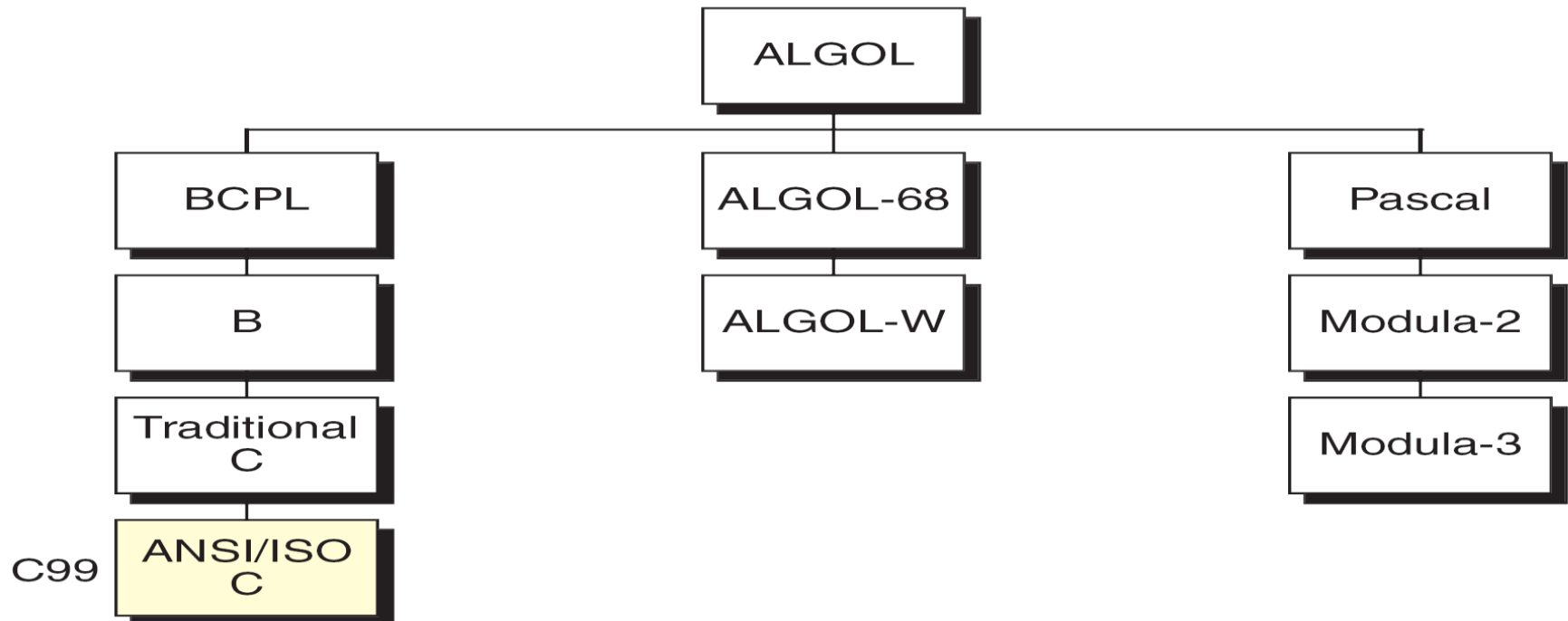
Characteristics of C

- A high level programming language .

- Small size. C has only 32 keywords. This makes it relatively easy to learn.

- Makes extensive use of function calls.

- C is a structured programming.

- It supports loose typing (as a character can be treated as an integer and vice versa).

- Facilitates low level (bitwise) programming

- Supports pointers to refer computer memory, array, structures and functions.

- C is a Portable language.

- C is a core language

- C is an extensible language

Uses of C Language:

- C language is primarily used for system programming. The portability, efficiency, the ability to access specific hardware addresses and low runtime demand on system resources makes it a good choice for implementing operating systems and embedded system applications.

- C has been so widely accepted by professionals that compilers, libraries, and interpreters of *other* programming languages are often implemented in C.

- For portability and convenience reasons, C is sometimes used as an intermediate language by implementations of other languages. Example of compilers which use C this way are BitC, Gambit, the Glasgow Haskell Compiler, Squeak, and Vala.

- C is widely used to implement end-user applications

- C Programs

Structure Of C Program:

Preprocessor Directives

Global Declarations

int main (void)
 {

Local Declarations

Statements

 } // main

Other functions as required.

Preprocessor directives:

- Every C program is made of one or more Preprocessor directives or commands.

- They are special instructions to the preprocessor that tell it how to prepare the program for compilation.

- The preprocessor directives are commands that give instructions to the C preprocessor.

- A preprocessor directive begins with a number symbol (#) as its first non-blank character.

- A common preprocessor command is "include". The "include" command tells the preprocessor that information is needed from selected libraries known as "header files".

- Preprocessor commands can start in any column, but they traditionally start in column 1.

    Ex: #include <stdio.h>

- This command tells the preprocessor that definitions from the library file in the brackets < > is included in the program. The name of the header file is "stdio.h". This is an abbreviation for "standard input / output header file.

- C requires that certain standard libraries be provided in every ANSI C implementation.

Global Declaration Section :

- Contains declarations that are visible to all parts of the program

Declaration section :

- It is at the beginning of the function. It describes the data that will be used in the function. Declarations in a function are known as local declarations as they are visible only to the function that contains them.

Statements:

- Statements follows the declaration section. It contains the instructions to the computer.
- Every statement ends with a semi colon.

Comments:

- Comment about the program should be enclosed within /* */.
- Any number of comments can be written at any place in the program.
- Comments in the code helps to understand the code
- Comments cannot be nested.
  - For example, /* Cal of SI /* Author sam date 01/01/2002 */ */
    is invalid.
- A comment can be split over more than one line, as
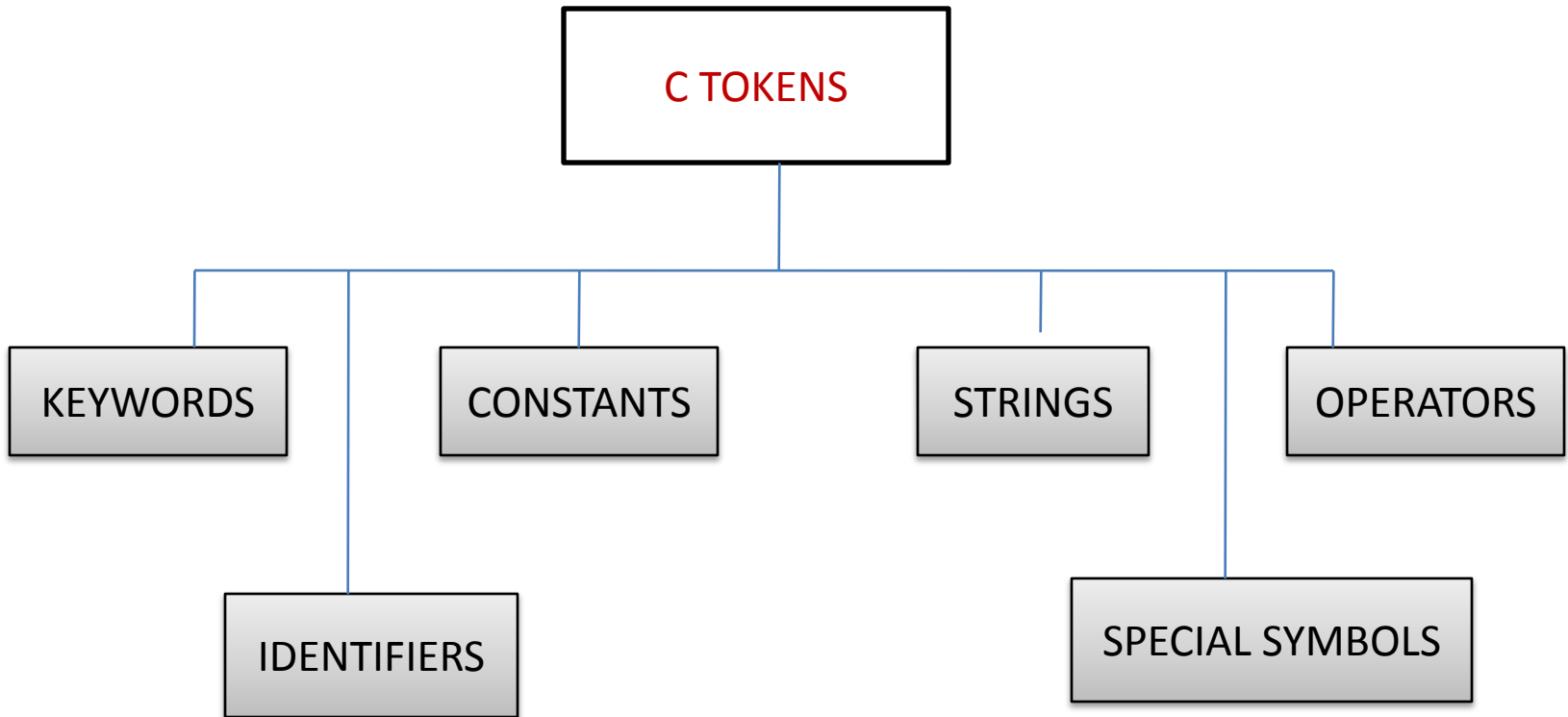   /* This is
    a jazzy
    comment */

main( ):

- The executable part of the program begins with the function 'main'. All statements that belong to main are enclosed in a pair of braces { }.

First C Program

```
#include <stdio.h>
void main ()
{
    printf("Hello World:\n");
}
```

- The main function contains single statement to print the message.
- The print statement use a library function to do the printing.

C TOKENS

KEYWORDS

- Keywords are predefined, reserved words used in programming that have special meaning. Keywords are part of the syntax and they cannot be used as an identifier. For example: int money;

- Here, **int** is a keyword that indicates **'money'** is a variable of type integer.

Identifiers

- Identifiers are names given to program elements such as variables, arrays and functions.

- Each identified object in the computer is stored at a unique address.

- If we didn't have identifiers that we could use to symbolically represent data locations, we would have to know and use object's addresses. Instead, we simply give data identifiers and let the compiler keep track of where they are physically located.
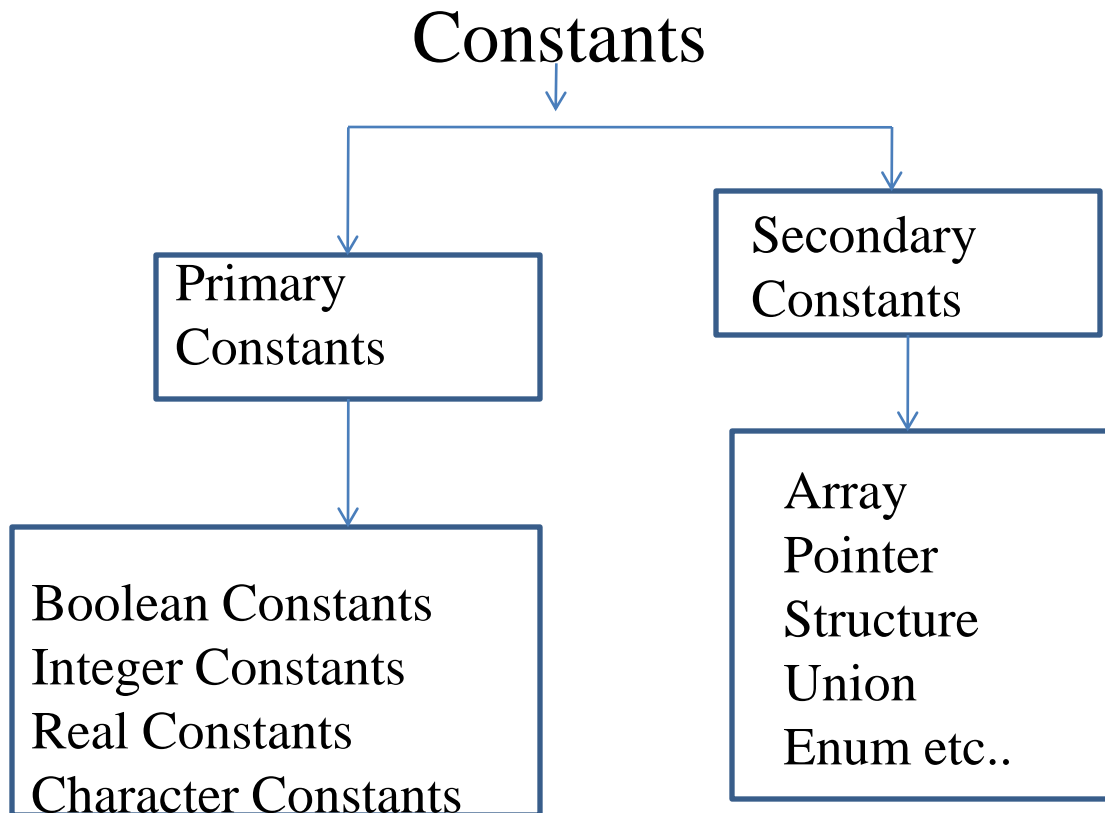
Rules for Identifiers:

1. First character must be alphabetic character or underscore.

2. Must consist only of alphabetic characters, digits, or underscores.

3. First 63 characters of an identifier are significant.

4. Cannot duplicate a keyword.

Examples of Valid and Invalid Names:

Valid Names		Invalid Name	
a	// Valid but poor style	$sum	// $ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

Constants:

- Constants are data values that cannot be changed during the execution of a program.

- C constants can be divided into two major categories:

Constants

Primary Constants

Secondary Constants

Boolean Constants
Integer Constants
Real Constants
Character Constants

Array
Pointer
Structure
Union
Enum etc..

- **Boolean constants**
  - A Boolean data type can take only two values. The values are *true* and *false*.
- **Integer Constants**

Rules for Constructing Integer Constants

- An integer constant must have at least one digit.

- It must not have a decimal point.

- It can be either positive or negative.

- If no sign precedes an integer constant it is assumed to be positive.

- No commas or blanks are allowed within an integer constant.

Example of Integer Constans

Representation	Value	Type
+123	123	int
–378	–378	int
–32271L	–32,271	long int
76542LU	76,542	unsigned long int
12789845LL	12,789,845	long long int

Real Constants:

- ## Rules for Constructing Real Constants

  - Real constants are often called Floating Point constants. It consists of integral part and fractional part. The real constants can be in

  – Fractional form

  – Exponential form.

  - In Fractional Form

  – A real constant must have at least one digit.

  – It must have a decimal point.

  – It could be either positive or negative.

  – Default sign is positive.

  – No commas or blanks are allowed within a real constant.

Examples of Real Constants

Representation	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
–2.0f	–2.0	float
3.1415926536L	3.1415926536	long double

- Exponential Form
  - In exponential form of representation, the real constant is represented in two parts. The part appearing before 'e' is called mantissa, whereas the part following 'e' is called exponent.

- ➢Rules for constructing real constants expressed in exponential form:
  - The mantissa part and the exponential part should be separated by a letter e.
  - The mantissa and exponent part may have a positive or negative sign.
  - Default sign of mantissa part is positive.
  - The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

    Ex: $1.23 \times 10^5 = 123000.0$ is written as 1.23e5 or 1.23E5
  - 0.34e-4 = 0.000034

Character Constants

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within two single quotes (apostrophes).

- The maximum length of a character constant can be 1 character.

- The character in the character constant comes from the character supported by the computer.

Ex: `A`   `I` `5` `=` `a`

ASCII Character	Symbolic Name
null character	`'\0'`
alert (bell)	`'\a'`
backspace	`'\b'`
horizontal tab	`'\t'`
newline	`'\n'`
vertical tab	`'\v'`
form feed	`'\f'`
carriage return	`'\r'`
single quote	`'\''`
double quote	`'\"'`
backslash	`'\\'`

Fig: Symbolic Names for Control Characters

Coding constants

Literal Constant

- A literal is an unnamed constant used to specify data.
   Ex: a=b+5;
   hear 5 is a literal constant.

Defined constants

- A defined constant uses the preprocessor command #define
     Ex: #define rate 0.85
 Preprocessor replaces each defined name, rate with the value 0.85 wherever it is found in the source program

Memory Constants

- Memory constant use a C type qualifier, const to indicate that the data cannot be changed
     Ex:const float PI = 3.14159;

STRINGS

- A string in C is merely an array of characters. The length of a string is determined by a terminating **null** character: '\0' .

- So, a string with the contents, say, "abc" has four characters: 'a' , 'b' , 'c' , and the terminating **null** character. The terminating **null** character has the value zero.

SPECIAL SYMBOLS

- The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

- [] () {} , ; : * ... = #

- **Braces{}:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

- **Parentheses():** These special symbols are used to indicate function calls and function parameters.

- **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Variables:

- Variables are named memory locations that have a type, such as integer or character, which is inherited from their type.
- The type determines the values that a variable may contain and the operations that may be used with its values.
- To declare a variable specify data type of the variable followed by its name. Variable declaration always ends with a semicolon
- Variable names should always be meaningful and must reflect the purpose of their usage in the program.

Variable Declaration

Synatax: Type   var_name;

Ex: int   emp_num;

        float   alary;

        char   grade;

        double balance_ amount;

        unsigned short int   acc_no;

Variable Initialization

– When a variable is defined, it contains unknown value. The variable has to be initialized with a known value

– If a variable is not initialized, the value of variable may be either 0 or garbage depending on the storage class of the variable.

– We must initialize any variable with known data before executing the function

```
char code = 'b';

int  i    = 14;

long long natl_debt = 1000000000000;

float     payRate  = 14.25;

double    pi       = 3.1415926536;
```

B	code
14	i
1000000000000	natl_debt
14.25	payRate
3.1415926536	pi

Program   Memory

Fig. Variable Initialization

Types

- A type defines a set of values and a set of operations that can be applied on those values.

    Ex: – Type - light switch
        Values –'ON' or 'OFF'
        Operations – 'turn on' and 'turn off'
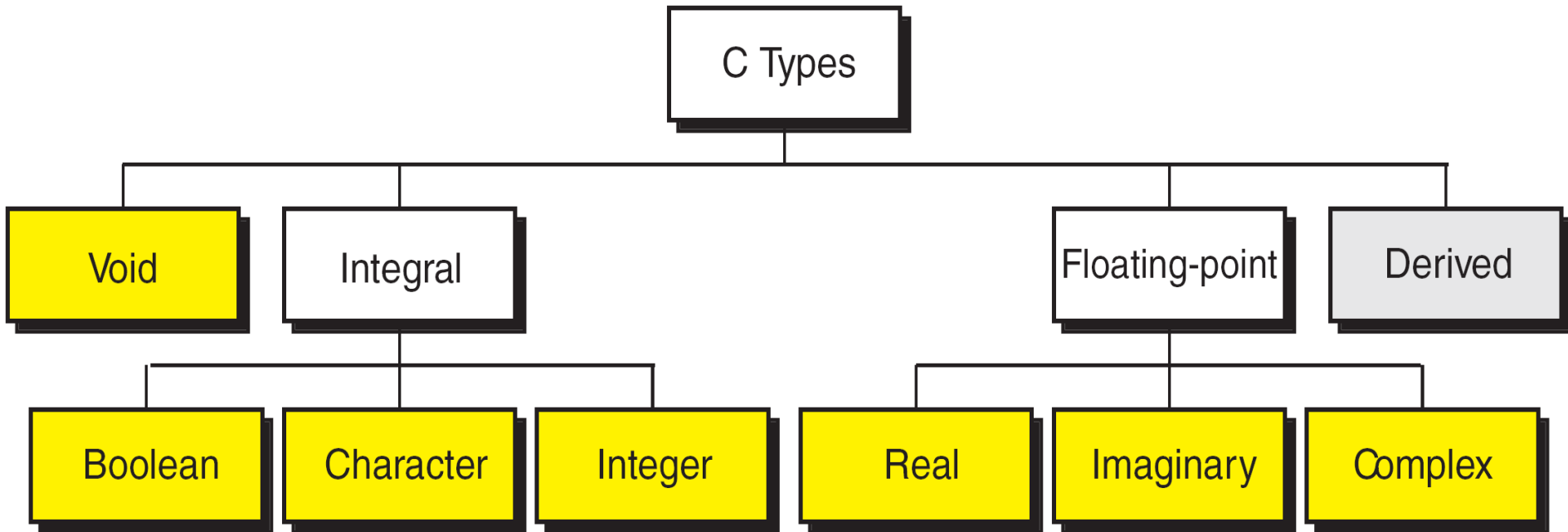
Void Type:

- Is identified by the key word 'void' and no operations.
- It is used to designate that a function has no parameters.
- It can also be used to define that a function has no return value.

Integral Type:

Boolean:

- Boolean type can represent only two values: *true* or *false*
- Referred by the keyword *Bool*
- Is stored in memory as 0 (false) or 1 (true)

Character:

- A character is any value that can be represented in the computer's alphabet
- It is referred by the keyword *char*
- One byte is used to store *char*. With 8 bits, 256 different values can be possible for the *char* type
- Character can be signed or unsigned.

Integer

- An integer type is a number without a fraction part
- C supports four different sizes of the integer type and is denoted by the keyword   int
  - » short int
  - » int $sizeof\ (short) \le sizeof\ (int) \le sizeof\ (long) \le sizeof\ (long\ long)$
  - » long int
  - » long long int
- Each integer size can be signed or unsigned integer. If the integer is signed, one bit is used for signed(0 is plus, 1 is minus). An unsigned integer can store a positive number that is twice as large as the signed integer of the same size.

short int

int

long int

long long int
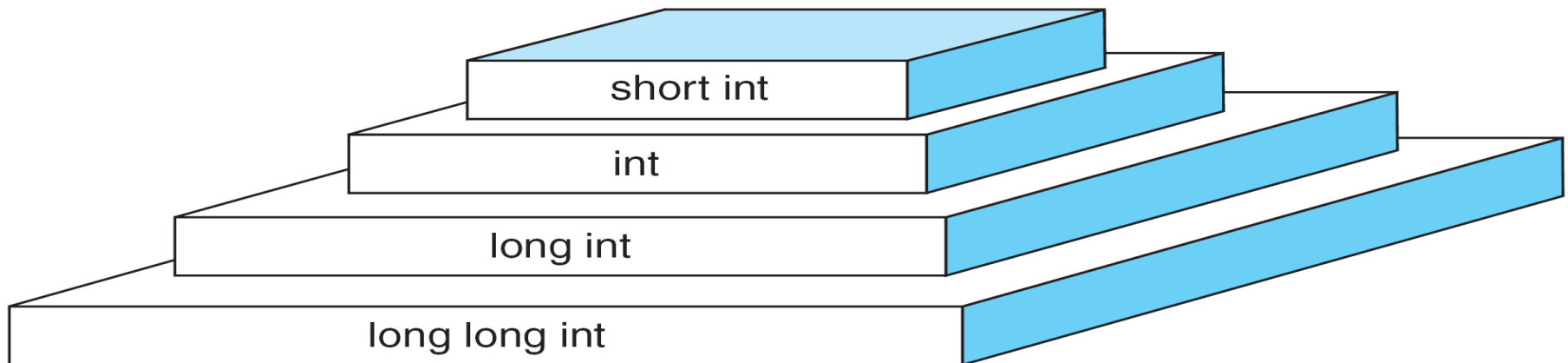
Fig: Integer Types

Floating-point type:

Real

- Real type holds values that consists of integral and fractional part.
- C support types float and double.
- Real type values are always signed.
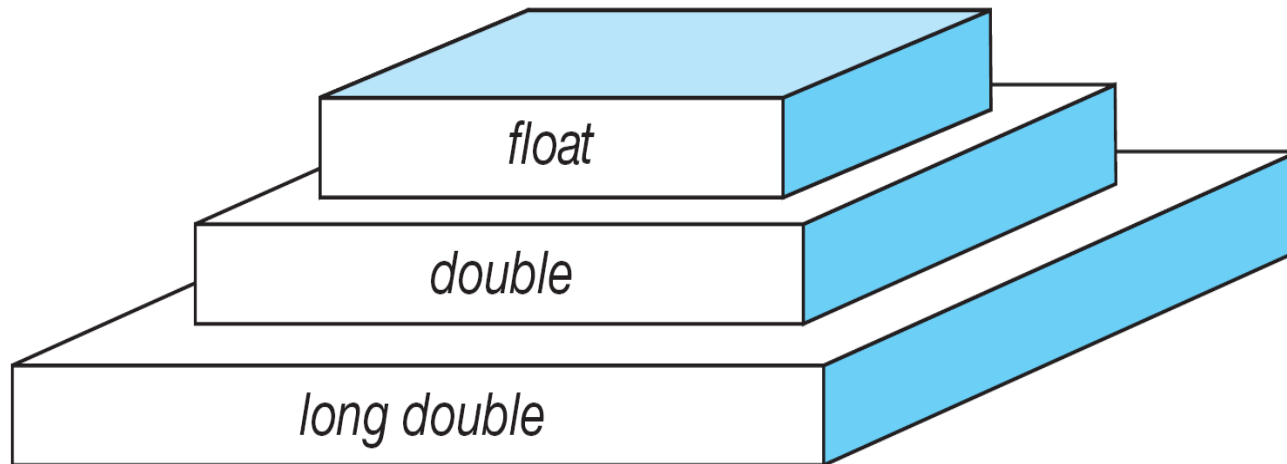
Fig : Floating-Point Types

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

- Data Types

Data Type		Abbreviation	Size (byte)	Range
char	`char`		1	**-128 ~ 127**
	`unsigned char`		1	**0 ~ 255**
int	`int`		2 or 4	**-2^{15} ~ 2^{15}-1 or -2^{31} ~ 2^{31}-1**
	`unsigned int`	`unsigned`	2 or 4	**0 ~ 65535 or 0 ~ 2^{32}-1**
	`short int`	`short`	2	**-32768 ~ 32767**
	`unsigned short int`	`unsigned short`	2	**0 ~ 65535**
	`long int`	`long`	4	**-2^{31} ~ 2^{31}-1**
	`unsigned long int`	`unsigned long`	4	**0 ~ 2^{32}-1**
float			4	
double			8	

Note: $2^7 = 128$, $2^{15} = 32768$, $2^{31} = 2147483648$

Operators in C

- C language supports a lot of operators to be used in expressions. These operators can be categorized into the following major groups:

    1) Arithmetic operators

    2) Relational Operators

    3) Equality Operators

    4) Logical Operators

    5) Unary Operators

    6) Conditional Operators

    7) Bitwise Operators

    8) Assignment operators

    9) Comma Operator

    10) Sizeof Operator

Arithmetic operators

- Assume the values a=9 and b=3

OPERATION	OPERATOR	SYNTAX	COMMENT	RESULT
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a – b	6
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Modulus	%	a % b	result = a % b	0

Relational Operators

- Thease operators compares two values so also called Comparison operators.

- Relational operators return true or false value, depending on the conditional relationship between

OPERATOR	MEANING	EXAMPLE
<	LESS THAN	3 < 5 GIVES 1
>	GREATER THAN	7 > 9 GIVES 0
<=	LESS THAN OR EQUAL TO	100 <= 100 GIVES 1
>=	GREATER THAN EQUAL TO	50 >=100 GIVES 0

Operators in C

Equality Operators

- C language supports two kinds of equality operators to compare their operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operator.

OPERATOR	MEANING
==	RETURNS 1 IF BOTH OPERANDS ARE EQUAL, 0 OTHERWISE
!=	RETURNS 1 IF OPERANDS DO NOT HAVE THE SAME VALUE, 0 OTHERWISE

Operators in C

Logical Operators

- C language supports three logical operators.

  They are 1) Logical AND (&&)

  2) Logical OR (||)

  3) Logical NOT (!)

- In case of arithmetic expressions, the logical expressions are evaluated from left to right.

A	B	A &&B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

A	! A
0	1
1	0

Operators in C

Unary Operators

- Unary operators act on single operands.
- C language supports three unary operators.

  They are :

    1) Unary minus(-)

    2) Increment (++)

    3) Decrement(--)

- When an operand is preceded by a minus sign, the unary operator negates its value.

  Ex: int x=5;   int y= -x; then y store the value -5.

- The increment operator   increases the value of its operand by 1.

  Ex: int x=3,y;

        printf("x=%d",x++);   x=3

         y=x;   y=4;

         printf("y=%d",++y);   y=5;

Operators in C

Unary Operators

- The decrement operator decreases the value of its operand by 1.

  Ex: int x=3,y;

    printf("x=%d",x--);   x=3

     y=x;   y=2;

     printf("y=%d",--y);   y=1;

Conditional Operator or Ternary Operator

- The conditional operator (?:) is just like an if .. else statement.
- The syntax of the conditional operator is

        exp1 ? exp2 : exp3

   Ex: a=10   b=5

        large = (a > b) ? a : b

        large=10.

Operators in C

Bitwise Operators

- Bitwise operators perform operations at bit level.

  They are : 1) Bitwise AND(&)

  2) Bitwise OR   (|)

  3) Bitwise XOR (^)

  4) Bitwise Shift   (<< and >>)

  5) Bitwise NOT   (~)

- The bitwise AND operator (&) is a small version of the boolean AND (&&) as it performs operation on bits instead of bytes, chars, integers, etc.

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Ex: x=2 y=3   x&y

      0 0 1 0

      0 0 1 1

x&y=   0 0 1 0

Operators in C

- The bitwise OR operator (|) is a small version of the boolean OR (||) as it performs operation on bits instead of bytes, chars, integers, etc.

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Ex: x=2 y=3 x|y

$$0\ 0\ 1\ 0$$
$$0\ 0\ 1\ 1$$

x|y= 0 0 1 1

- The bitwise XOR operator (^) performs operation on individual bits of the operands.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ex: x=2 y=3 x^y

$$0\ 0\ 1\ 0$$
$$0\ 0\ 1\ 1$$

x^y= 0 0 0 1

Operators in C

- In bitwise Shift operations, the digits are moved, or *shifted*, to the left or right.

- The CPU registers have a fixed number of available bits for storing numerals, so when we perform shift operations; some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end.

Ex:

In a left arithmetic shift,the right side end filled with 0's.

int x = 11000101;

Then x << 2 = 00010100

In a right arithmetic shift,the left side end filled with 0's.

int x = 11000101;

Then x >> 2 = 00110001

- The bitwise NOT (~), or complement, is a unary operation that performs logical negation on each bit of the operand.

- By performing negation of each bit, it actually produces the 1's complement of the given binary value.

  Ex: int x=4,y;

  y=(~x);   Hear x is 0 1 0 0

  y=11   ~x is 1 0 1 1

Operators in C

Assignment operators

- The assignment operator is responsible for assigning values to the variables.
- The equal sign (=) is the fundamental assignment operator.
- C supports other assignment operators that provide shorthand (Compact)ways to represent common variable assignments.

OPERATOR	SYNTAX	EQUIVALENT TO
/=	variable /= expression	variable = variable / expression
\=	variable \= expression	variable = variable \ expression
*=	variable *= expression	variable = variable * expression
+=	variable += expression	variable = variable + expression
-=	variable -= expression	variable = variable - expression
&=	variable &= expression	variable = variable & expression
^=	variable ^= expression	variable = variable ^ expression
<<=	variable <<= amount	variable = variable << amount
>>=	variable >>= amount	variable = variable >> amount

Comma operator

- The Comma operator in C takes two operands. It works by evaluating the first and discarding its value, and then evaluates the second and returns the value as the result of the expression.

- Comma separated operands when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression.

- Among all the operators, the comma operator has the lowest precedence.

    Ex: int a=2, b=3, x=0;

       x = (++a, b+=a);

     Now, the value of x = 6.

Operators in C

Sizeof Operator

- Sizeof operator used to calculate the sizes of data types.

- It can be applied to all data types.

- The operator returns the size of the variable, data type or expression in bytes.

Ex:

→ sizeof(char) returns 1 byte, that is the size of a character data type.   If we have,

int a = 10;

unsigned int result;

result = sizeof(a);

then result = 2 bytes.

Basic Input - Output

- Data is input to and output from a stream. A stream is a source of or destination for data.
- Stream is Associated with a Physical devices such as Terminals or with a file stored in auxiliary memory.
- C language uses two types of Streams:
  1) Text Stream
  2) Binary Stream
- <u>Text Stream:</u> Text Stream is a sequence of Characters divided into lines with each line terminated by a new line.
- <u>Binary Stream:</u> Binary Stream is a Sequence of data values such as integer,real,or complex using their memory representation.
- In C language standard input device is a Keyboard and the out put device is a Monitor.

Basic Input - Output

Data Source   Program

keyboard

Input Text Stream → Data

monitor
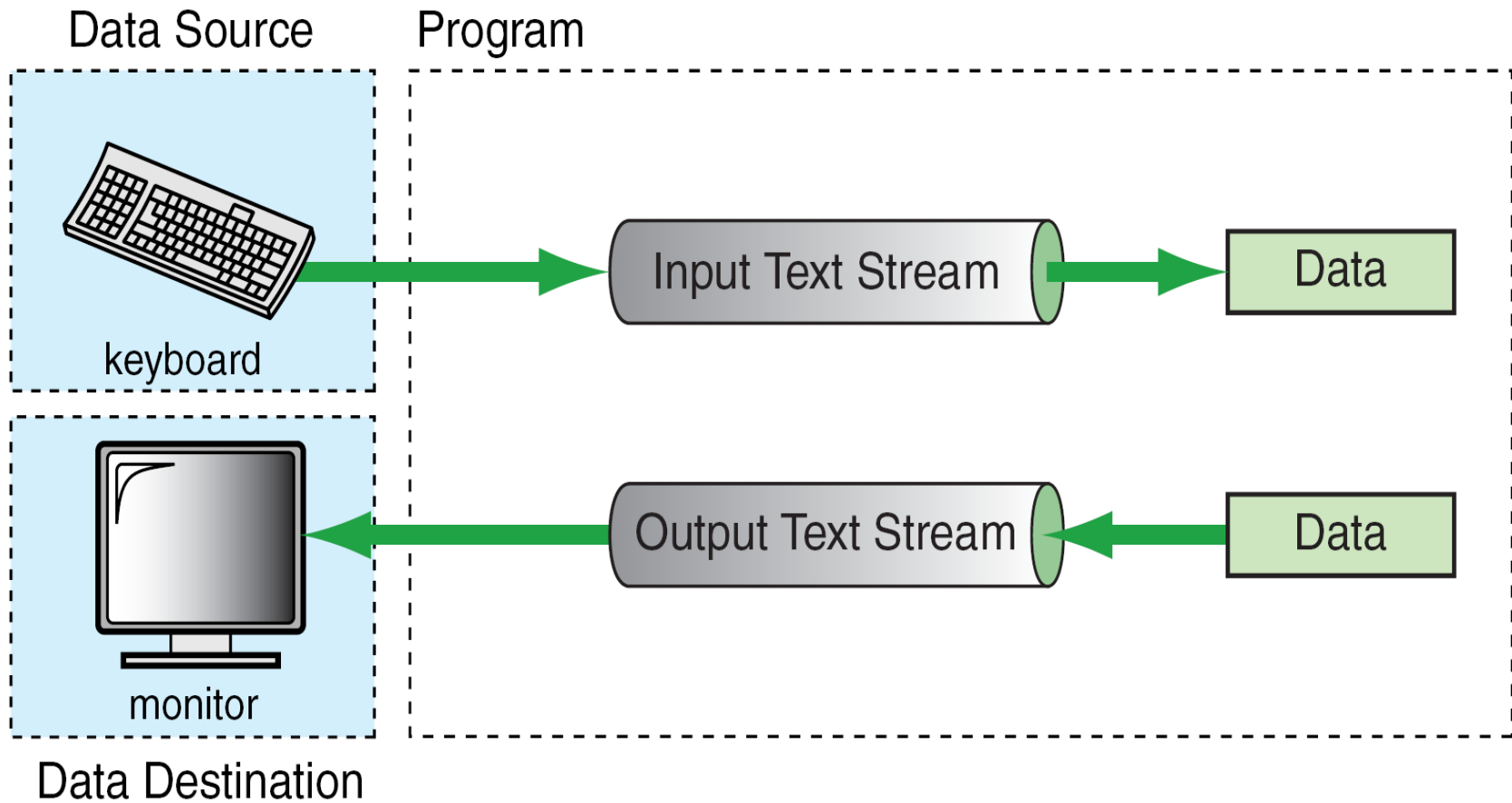
Output Text Stream ← Data

Data Destination

Fig: Stream Physical Devices

Basic Input - Output

- C language provides two formatting functions
  - *printf* for output formatting
  - *scanf* for input formatting

- *printf* function converts data stored in program into a text stream for output to the monitor.

- *scanf* converts the text stream coming from the keyboard to data values and stores them in program variables .

- Printf and scanf functions are data to text stream and text stream to data converters.

Basic Input - Output

Output Formatting:**printf**

- **The** printf function takes a set of data values ,converts them to a text stream using formatting instructions contained in a format control string and sends the resulting text stream to the standard output.
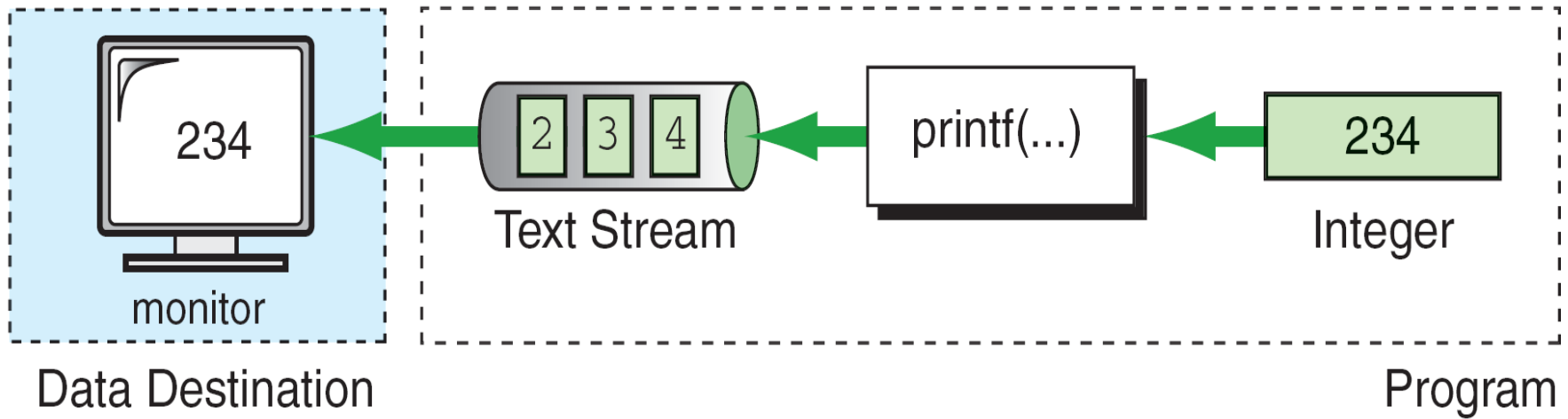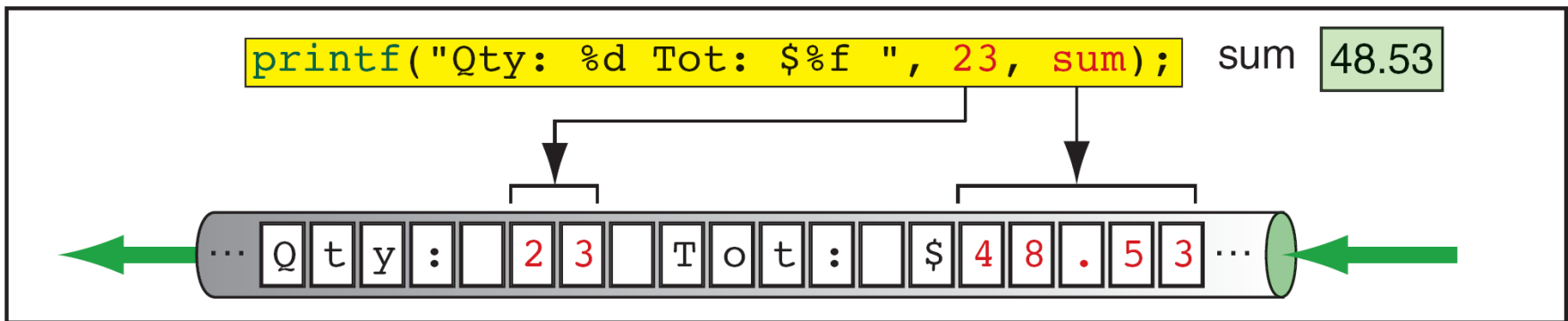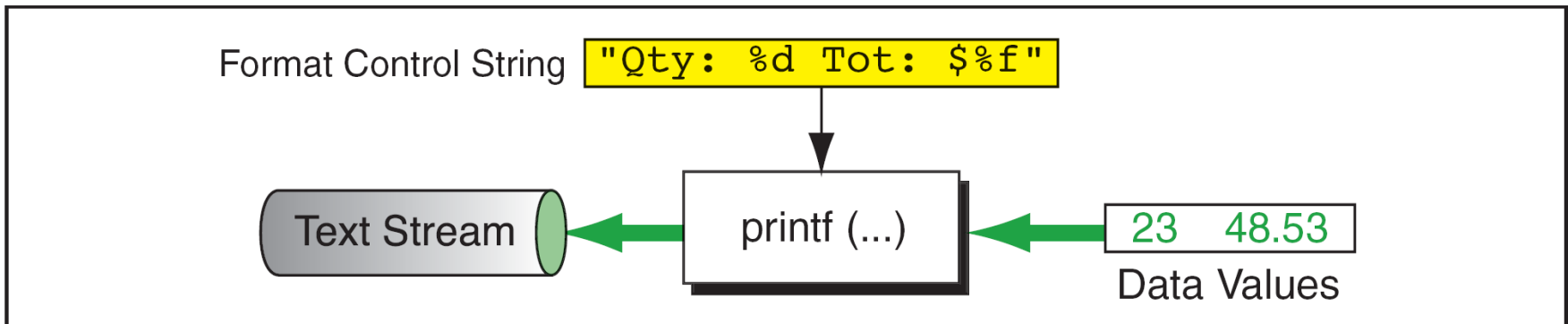

- In the above diagram an Integer 234 stored in the program is converted to a text stream of three numeric ASCII characters '2' '3' '4' and then is

Basic Input – Output

- The following information is passed to the Printf Function:

  1) The format control string including any textual data to be inserted into the text stream.

  2) A set of zero or more data values to be formatted.

(a) Basic Concept

Format Control String `"Qty: %d Tot: $%f"`

Text Stream ← printf (...) ← 23   48.53

Data Values

(b) Implementation

`printf("Qty: %d Tot: $%f ", 23, sum);`   sum   48.53

... Q t y :   2 3   T o t : $ 4 8 . 5 3 ...

Basic Input – Output

- In the above first diagram shows the format string and the data values as parameters for the print function.

- With in the control string we have specified quantity(Qty:) and total(Tot:) as textual data and two converstion specifications(%d and %f).

- In the Second diagram shows the formatting operation and the resulting text stream.
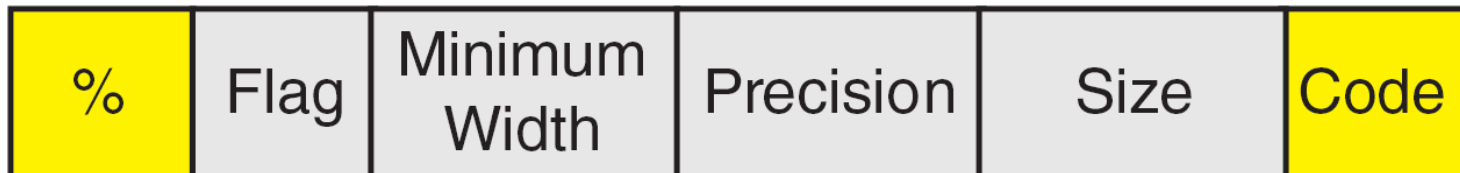
Basic Input – Output

Format Control String Text

- The control string also contains text to be printed such as instructions to the user,captions or identifiers and other text intended to make the output more readable.
- It also prints control characters like(\t,\n).

Conversion Specification

- To insert the data into the stream we use the conversion specification .
- It contains a start token(%),a conversion code and four optional modifiers.

Fig: Conversion Specification

%	Flag	Minimum Width	Precision	Size	Code

Basic Input – Output

Code

- Approximately there are 30 different codes are available in C language to describe data types.

- Hear we concern only three code like character(c),Integer(d),Floating point(f).

Size

- The size modifier is used to modify the type specified by the conversion code.

- There are four different sizes: h,l(el),ll(el,el) and L.

- The h is used with the integer codes to indicate a short integer value.

- The l is used to indicate a long integer value.

- The ll is used to indicate a long long integer value.

- The L is used with floating point numbers to indicate a long double value.

Basic Input – Output

Type	Size	Code	Example
char	None	c	%c
short int	h	d	%hd
int	None	d	%d
long int	None	d	%ld
long long int	ll	d	%lld
float	None	f	%f
double	None	f	%f
long double	L	f	%Lf

Fig:Format Codes for Output

Precision

- The precision modifier is used if a floating point number is being printed then we may specify the number of decimal places to be printed.

    The format of precision modifier is   .m

     Hear m is the number of decimal digits.

- If precision is not specified printf prints six decimal positions.

Width

- Width modifier is used to specify the minimum number of positions in the output.

- This is very useful to align output in columns.

- If we don't use a width modifier each output value will take just enough room for the data.

Flag

- The flag modifier is used with four print modifications:

  1. Justification   2. Padding   3. sign   4.numaric conversion

1.Justification

- It controls the placement of a value when it is shorter than the specified width.

- By default the justification is right.

- To left justify a value the flag is set to minus(-).

2. Padding

- It defines the character that fills the unused space when the value is smaller than the print width.

- It can be a space or zero.

- By default the unused width is filled with spaces.

- If the flag is 0 the unused width is filled with zeros.

Sign

- The sign flag defines the use or absence of a sign in numeric value.

- There are three formats to specify the sign

1. default formatting inserts a sign when the value is negative.

2. When the flag is set to plus(+) signs are printed for both positive and negative.

3. If the flag is space the positive numbers are printed with a leading space and negative numbers with a minus sign.

Numeric conversions

- Prefix o to the output value when used with the octal conversion specifier.

- Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.

Basic Input – Output

Flag Type	Flag Code	Formatting
Justification	None	right justified
	–	left justified
Padding	None	space padding
	0	zero padding
Sign	None	positive value: no sign negative value: –
	+	positive value: + negative value: –
	None	positive value: space negative value: –

Fig: Flag Formatting Options

Basic Input – Output

Input formatting : scanf

- The standard input formatting function scanf takes a text stream from the keyboard.

- Extracts and formats data from the stream according to a format control string.

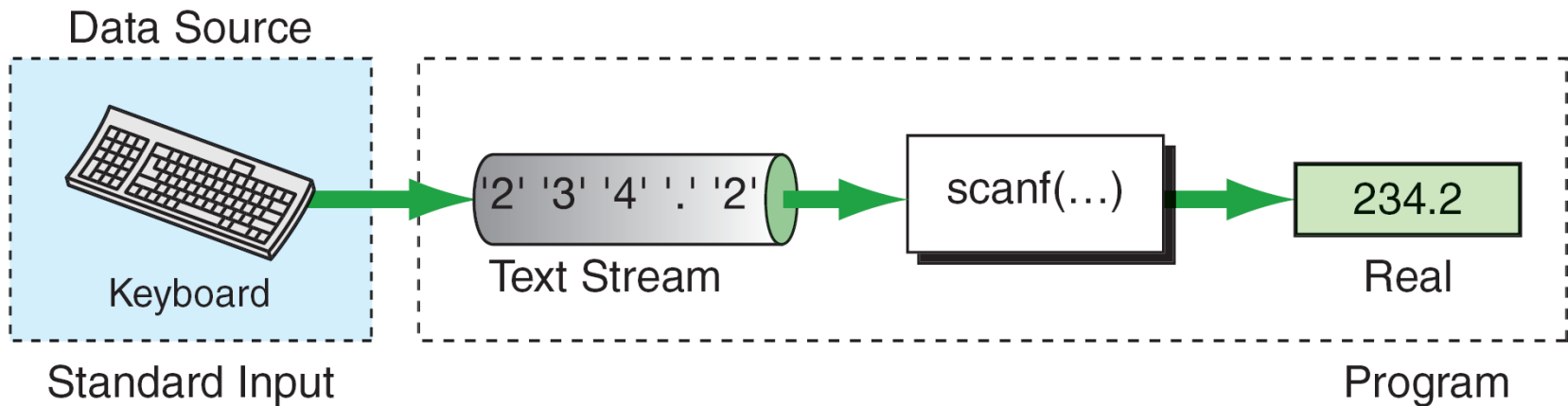- Then stores the data in specified program variables.

Fig: Formatting Text from an Input Stream

- In the above diagram the stream of 5 characters '2','3','4','.','2' are
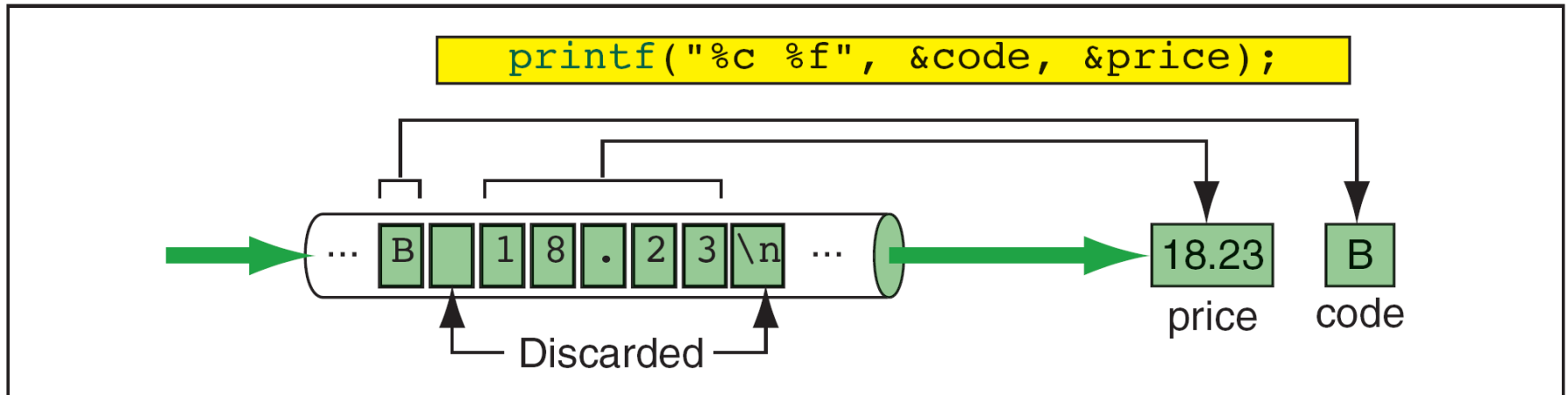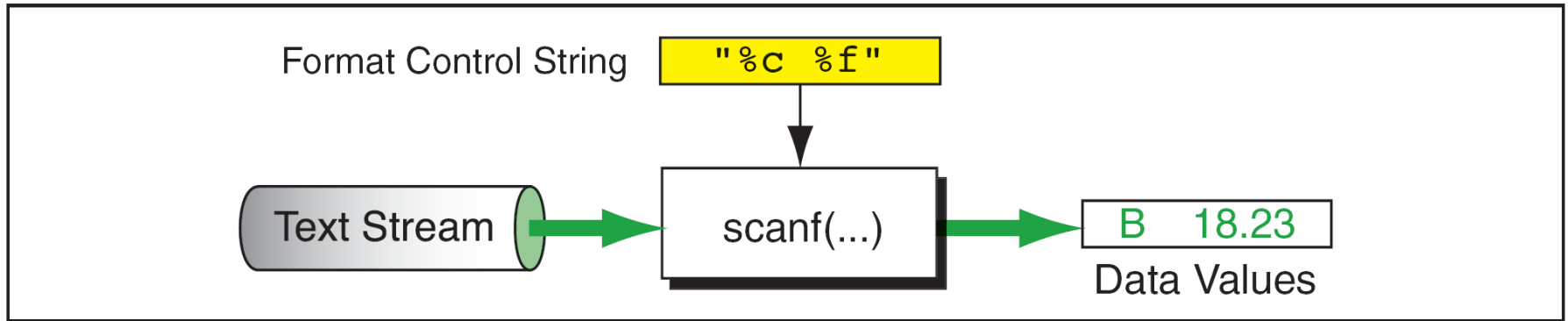
Basic Input – Output

- General syntax of Scanf function is:

  scanf ("control string", arg1, arg2, ………….argn);

- The control string specifies the type and format of the data that has to be obtained from the keyboard and stored in the memory locations pointed by the arguments arg1, arg2,…, argn.

- With the exception of character specification , leading white spaces are discarded.

Basic Input – Output

(a) Basic Concept

Format Control String `"%c %f"`

Text Stream → scanf(...) → B   18.23

Data Values

`printf("%c %f", &code, &price);`

... B   1 8 . 2 3 \n ... → 18.23   B

price   code

Discarded

(b) Implementation

Fig: Input Stream Formatting

Format Control String

- Like printf formatting string scanf is also enclosed with in a set of quotation marks.

- It contains on or more conversion specifications that describe the data type and indicate any special formatting rules and characters.

Conversion Specification

- To format input data stream we use a conversion specification that contains a start token(%) , conversion code and three optional modifiers.

%	Flag	Maximum Width		Size	Code

Fig: Conversion Specification

Basic Input – Output

- There are three differences between the conversion codes for input and output formatting:

1) There is no precision in an input conversion specification.

   -It is an error to include a precision

2) There is only one flag for input formatting , the assignment suppression flag(*).

   -The assignment suppression flag tells the scanf the next input field is to be read but not store.

   Ex: scanf("%d %*c %f",&x, &y);

3) The width modifier specifies the maximum number of characters that are to be read for one format code.

Basic Input – Output

Input Parameters

- Every conversion specification there must be a matching variable in the address list.

- The address of a variable is indicated by prefixing the variable name with an ampersand (&).

- In C language '& ' is known as address operator.

- The first conversion specification matches the first variable address , the second conversion specification matches the second variable address and so on.

- The variable type match the conversion type.

- The c compiler does not verify that they match . If they don't match the input data will not be formatted properly when they are stored in the variables.

Basic Input – Output

- The scanf terminate input process when the user signal that there is no more input by keying end of file(EOF).

  Ex: Ctrl+Z

- If scanf encounters an invalid character when it is trying to convert the input to the stored data type ,it stops.

  Ex: Character is trying to read a numeric .

Expressions

- An Expression is a sequence of operands and operators that reduces to a single value.

- An expression is a simple or complex.

- A simple expression contain only one operator.

  Ex: 3+6,-a

- A complex operator contains more than one operator.

  Ex: 2*5+7-8

- An expression contains operator and operand.

- An operator is a syntactical token that requires an action to be taken.

- An operand is an object on which an operator is performed.

- An Expression always reduces to a single value.

Expressions

- A Simple Expression is divided into Six Categories based on the number of operands , relative position of the operand and operator and the precedence of operator.
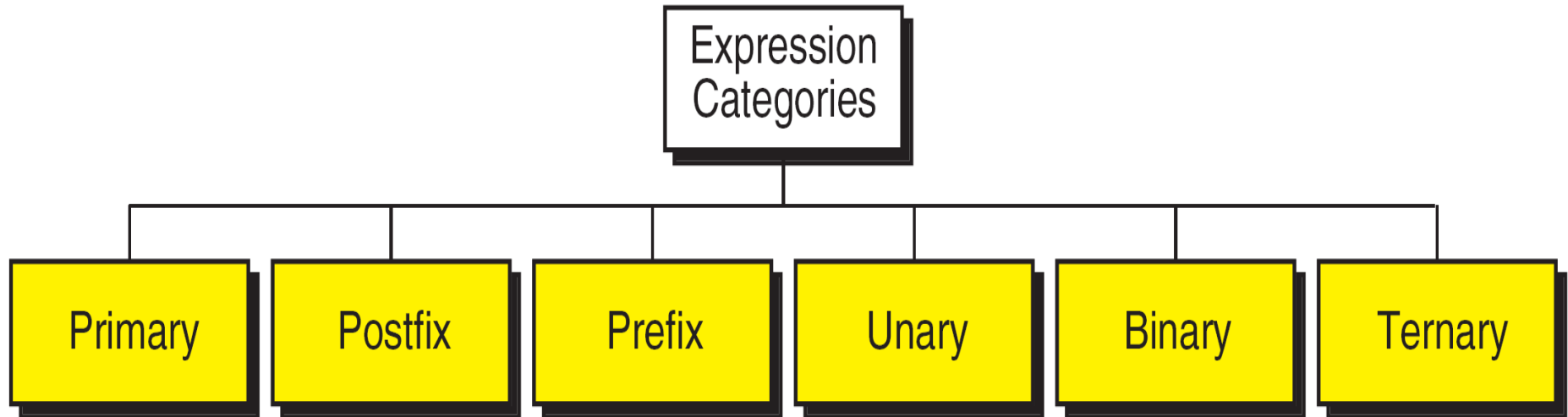
Expression Categories

| Primary | Postfix | Prefix | Unary | Binary | Ternary |

Fig: Expression Categories

Primary Expressions

- A primary Expression Consists of only one operand with no operation.

- The operand in a primary expression can be a name, constant or a parenthesized expression.

- A primary expression is evaluated first in a complex expressions.

Name

- Name is any identifier for a variable , a function , or any other object .

  Ex: a   b12   price   calc   INT_MAX   SIZE

Literal Constant

- A constant is a piece of data whose value can't change during the execution of the program.

  Ex: 5   123.98   'A'   "Welcome"

Parenthetical Expressions

- Any value enclosed in a parentheses must be reducible to a single value and is therefore a primary expression.

- A complex expression enclosed with in parentheses to make a primary expression.

Ex: (2*3+4)   (a=23+b*6)

Postfix Expression

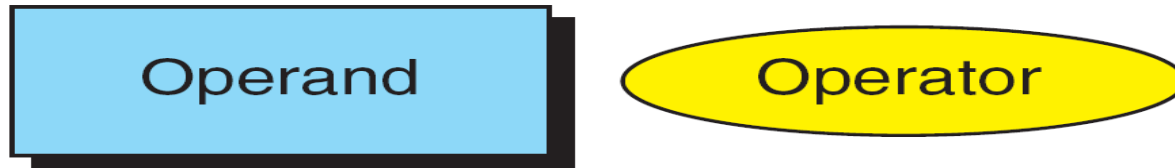- Postfix Expression consists of one operand followed by one operator.

Operand   Operator

Fig: Postfix Expressions

- Some of the operators that create a postfix expression are function call,

Function Call

- Function calls are postfix expressions.
- The function name is the operand and the operator is the parentheses that follow the name.
- The parentheses may contain arguments or be empty.
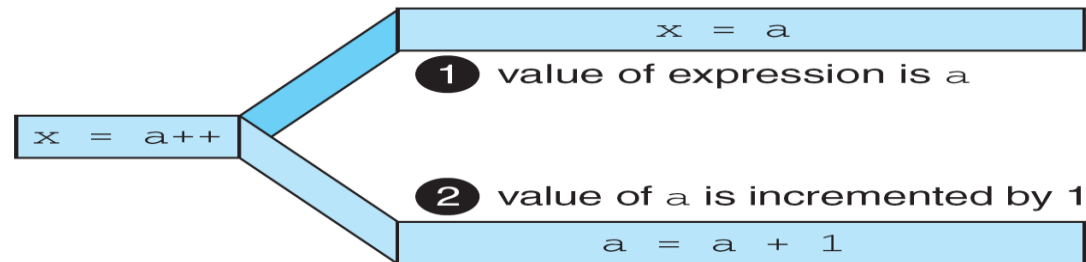
  Ex: printf("Hello World");   or printf( );

Postfix Increment/Decrement

- Postfix increment and decrement are also postfix operators.
- Postfix increment the variable is incremented by 1.

  Ex: int a=5,x;

  x=a++;

  x=6;

- Hear the effect of
- Postfix decrement the variable value is decremented by 1.

Expressions

Ex: int a=5,x;

      x=a--;

      x=4;

Prefix Expressions

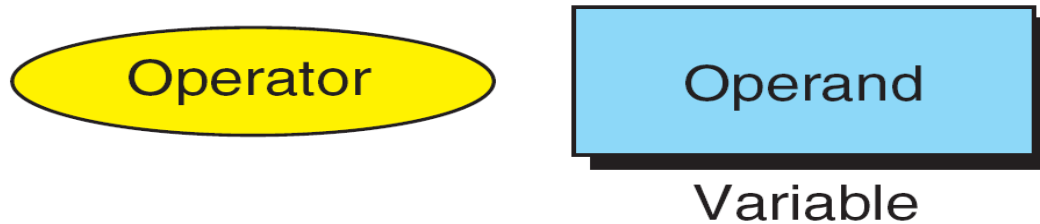- In prefix expression the operator comes before the operand.

Fig:Prefix Expression

Prefix Increment and Decrement

- Similar to postfix increment and decrement the prefix increment and decrement operators are shorthand notations for adding or subtracting 1from the value.

Expressions

Ex: int a=5,x;

    x=a++;

    x=6;

Postfix Increment

int a=5,x;
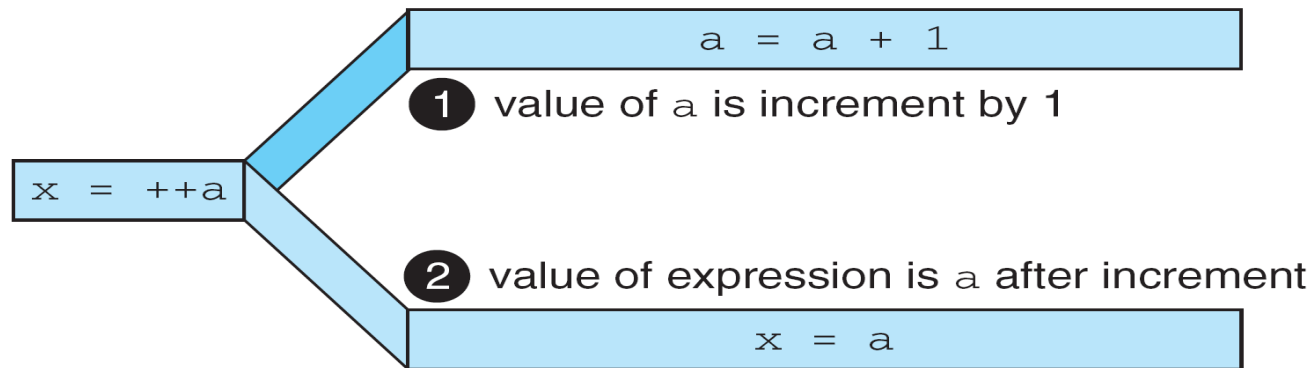
x=a--;

x=4;

Postfix Decrement

Fig:Result of Prefix ++a

- When we need the value of the expression to be the current value of the variable we use the postfix operator.

- When we need the value to be the new value of the variable we use prefix operator.

Unary Expression

- It is like a prefix expression consists of one operator and one operand.

- Prefix expression need a variable as the operand.

- Unary Expression can have an expression or variable as the operand.
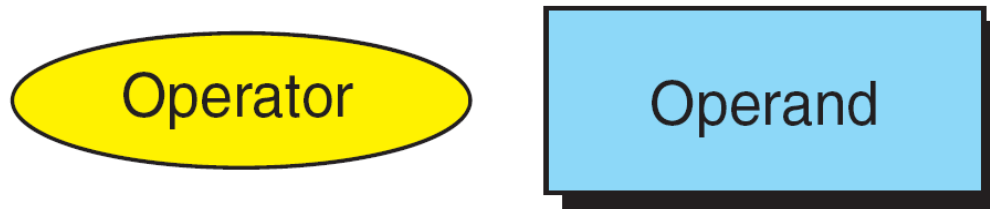
Operator   Operand

Fig:Unary Expressions

- Some of the Unary expressions are sizeofoperator,   plus/minus operator and the cast operator.

Sizeof

- The size of operator tells the size , in bytes , of a type or a primary expression.

- By specifying the size of an object during execution , we make our program more portable to other hardware.

  Syntax :   sizeof(type);

Unary plus/Minus

- These two operators are use to compute the arithmetic value of an operand.

- The plus operator does not change the value of an expression.

- The minus operator change the sign of a value algebraically.i.e to change it from plus to minus and minus to plus.

Expressions

Cast Operator

- The cast operator converts one expression type to another.

    Ex: To Convert integer type to float we use cast operator.

        int x;

        float y=float(x);

Binary Expressions

- Binary Expressions are formed by an operand-operator-operand combination.

| Operand | Operator | Operand |

Fig:Binary Expressions

Multiplicative Expressions

- Name of the expression takes from the first operator , include the multiply , divide , and modulus operator.

- These operators have the highest priority among other binary operators.

- Multiply operator is the product of the two operands.

  Ex: 10*3   //result=30

      true*4   //result=4

      'A'*2   //result=2

      22.3   //result=44.6

- The type of the result depends on the conversion rules.

  - if both operands are Integers result is Integer.

  -if any one of the operand is floating point the result is floating point.

  -if both operands are floating point the result is floating point.

Expressions

- The result of divide operator depends on the type of operand.

  Ex: 10/3   //result=3

      true/4   //result=0

      'A'/2   //result=32

- The modulus(%) operator divides the first operator by the second operator and returns the reminder rather than quotient.

  Ex: 10%3   // result=1

      true%4   // result=1

       'A'%10   // result=5

      22.3%2   // result=Error because modulo can not be float.

- Both operands must be integral types and operator returns the remainder as an integer type.

Expressions

Additive Expressions

- In this type of Expressions the second operand is added to or subtracted from the first operand , depending on the operator used.

    Ex: 3+7   // result=10

            3-7   // result=-4

Assignment Expressions

- The assignment expression evaluates the operand on the right side of the operator(=) and places its value in the variable on the left.

- The assignment expression has a value and a side effect.

- The value of the total expression is the value of the expression on the right of the assignment operator(=).

- The side effect places the expression value in the variable on the left of the assignment operator.

- There are two forms of Assignment Expressions:

        1.Simple   2.Compound

Expressions

1.Simple Expressions

- Simple assignment is found in algebraic expressions.

  Ex: a=5 ; b=x+1   i=i+1

- The left variable must be able to receive the effect.

- That is it must be a variable , not a constant.

- If the left operand cannot receive a value and we assign a value we get a compile time error.

2.Compound Assignment

- A compound assignment is a shorthand notation for a simple assignment.

- It requires that the left operand be repeated as a part of the right expression.

- To evaluate the compound expression first change it to simple expression.

Expressions

- The left operand in an assignment expression must be a single variable.

Compound Expression	Equivalent Simple Expression
x *= expression	x = x * expression
x /= expression	x = x / expression
x %= expression	x = x % expression
x += expression	x = x + expression
x -= expression	x = x - expression

Fig: Expansion of Compound Expressions

Ex: x*=y+3 evaluated as x=x*(y+3).

Precedence

- To determine the order in which different operators in a complex expression are evaluated.

  Ex: 1) 2+3*4   is evaluated as (2+(3*4)) =(2+12)=14.

     In the above example + having the precedence 12 and * having the precedence 13 so first multiply 3*4 then add 2+12.

     2) -6++ is evaluated as (-(6++))=-7.

Associativity

- Associativity is applied when we have more than one operator of the same precedence level in an expression.

- Associativity can be left-to-right or right-to-left.

- Left-to-right associativity evaluates the expression by starting on the left and moving to the right.

- Right-to-left associativity evaluates the expression by proceeding from the right to left.

Operator Type	Operator	Associativity
Primary Expression Operators	() [] . -> *expr*++ *expr*--	left-to-right
Unary Operators	* & + - ! ~ ++*expr* --*expr* (*typecast*) sizeof	right-to-left
Binary Operators	* / %	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	|	
	&&	
	||	
Ternary Operator	?:	right-to-left
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	right-to-left

Precedence and Associativity

Ex: (LTR)

1. 3 * 8 / 4 % 4 * 5

    ((((3*8)/4)%4)*5) value is 10.

Fig:Left-to-Right Associativity

- Hear all the operators having the same precedence so the associativity is Left to Right.

Precedence and Associativity

Ex:(RTL)

1. a += b *= c -= 5

   (a += (b *= (c -= 5)))

   (a = a + (b = b * (c = c-5)))

   if a has initial value of 3 , b has initial value of 5 , and c has initial value of 8 then expression becomes

   (a = 3 + (b = 5 * (c =8-5)))

   (a = 3 + (b = 5 * 3))

   (a = 3 + 15)

   a=18

Side Effects

- A side effect is an action that results from the evaluation of an expression.
- C language evaluates the expression from right of the assignment operator and place result in left variable.
- Changing the value of left variable is a side effect.

  Ex: x=x+3 expression has three parts . Assume x=3

  1. the value of right of the right of the assignment operator is 6

  2. the value of hole expression is also 6

  3. the side effect x receives the value 6.

- In c language six operators generate side effect.
  - prefix increment and decrement
  - postfix increment and decrement
  - Assignment
  - Function call

Expressions Without Side Effects

Ex: 1. a * 4 + b / 2 – c * b

 Assume the values for a=3 , b=4 , c=5

Substitute the values in the Expression

   3 * 4 + 4 / 2 – 5 * 4

 Evaluate expression based on their precedence

(3 * 4) + (4 / 2) – (5 * 4)

   12 + 2 - 20

    the value is -6.

• In the above expression there is no side effect , all the variables have
the same value after the expression has been evaluated.

Expression With Side Effects

Ex: 1. --a * (3 + b) / 2 – c++ * b

 Assume the values a=3 , b=4 , c=5

calculate the value of Parenthesized expression first

--a * 7 / 2 – c++ * b

--a * 7 / 2 – 5 * b

2 * 7 / 2 – 5 * b

14 /2 – 5 * b

7 – 5 * 4

7 – 20

 -13

- After the side effect the variables have the value

Type Conversion

- Converting one type of data to another type of data is called type conversion.

- There are two types of Conversions .

    1. Implicit type conversion

    2. Explicit Type Conversion

Implicit Type Conversion

- When the two types of a binary expression are different then the c compiler automatically converts one type to another. This is known implicit type casting.

- We can assign the conversion ranks to the integral and floating point arithmetic types.

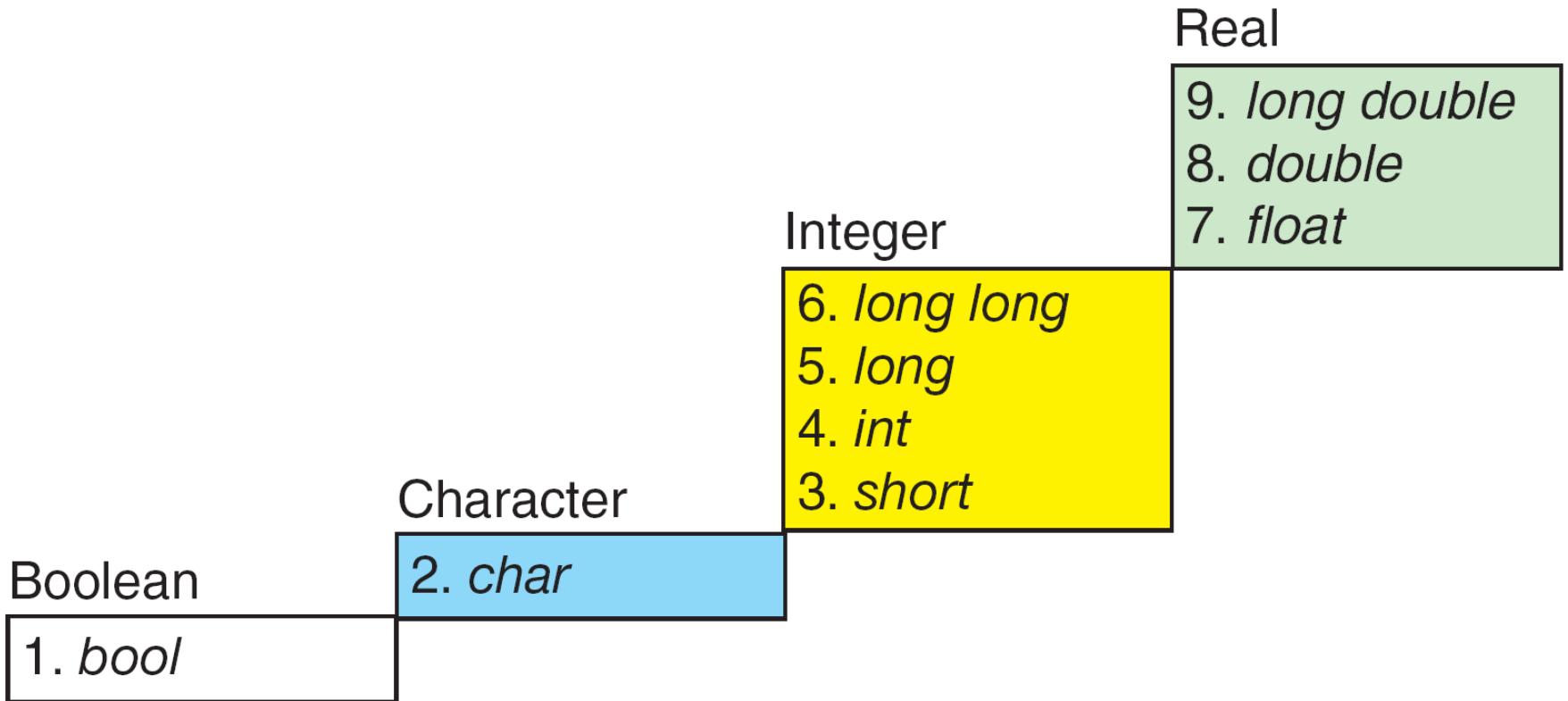- A long double real has a higher rank than a long

Type Conversion

Real

9. *long double*
8. *double*
7. *float*

Integer

6. *long long*
5. *long*
4. *int*
3. *short*

Character

2. *char*

Boolean

1. *bool*

Fig: Conversion Rank

Type Conversion

Conversion in Assignment Statements

- In a simple assignment expression involve an assignment operator and two operands.

- Depending on the rank of right expression the left variable takes promotion or demotion.

- Promotion occurs if the right expression has lower rank.

- Demotion occurs if the right expression has higher rank.

Promotion

- The rank of the right expression is evaluated to the rank of the left variable.

- Generally no problem with the promotion.

  Ex: char c='A';   i=c   //value of i is 65

        int i=1234;   d=i // value of d is 1234.0

        long double d=3456.2345;

Demotion

- If the right expression having the higher rank than the left variable then the right expression value is demoted to the left variable.

- When the integer or real is stored into a char , the least significant byte is converted to a char and stored.

- When a real is stored in a int the fraction part is dropped.

Ex: bool b= false;

char c='A';

int k=65;

b=c;   // value of b is 1(true)

c=k+1;   // value of c is B.

Explicit Type Conversion

- In Explicit type conversion the programmer convert data from one type to another.

- It uses Unary cast operator.

- To convert the data from one type to another , specify the new type in parentheses before the value to be converted.

   Ex: int a;

               float f=(float) a;

Statements

- A statement causes an action to be performed by the program.

- Most of the statements ends with semicolon , some do not.

- There are 11 types of statements.

Null Statement

- The null statement is just a semicolon.

  Syntax: ;   //null statement

- null statement is used where we must have a statement but no action is required.

Expression Statement

- An expression is turned into a statement by placing a semicolon after the expression.

    Syantax: expression ;

Return Statement

- A return statement terminates a function.

- All functions , including main , must have a return statement.

- When there is no return statement at the end of the function , the

Statements

Syntax: return statement;

- The return statement returns a value to the calling function.

- The main function returns 0 to the operating system.

- A return value of zero tells the operating system that the program executed successfully.

Compound Statement

- A compound statement is a unit of code consisting of zero or more statements.

- It is also called block.

- All the c functions contain a compound statement known as the function body.

- A compound statement consists of an opening brace , an optional declaration and definition section and an optional statement section , followed by a closing brace .

Statements

```
{
    // Local Declarations
    int  x;
    int  y;
    int  z;

    // Statements
    x  =  1;
    y  =  2;
    ...
}   // End Block
```

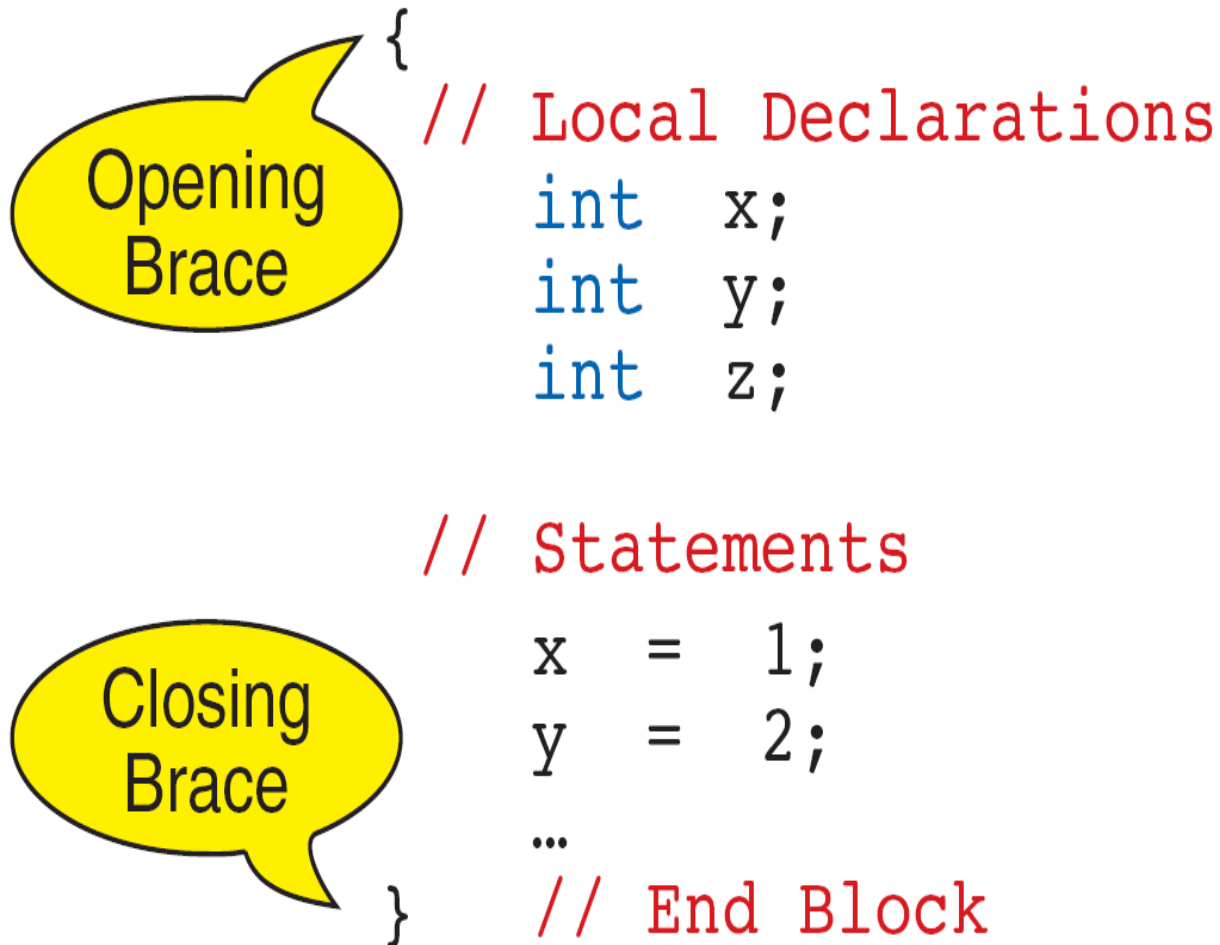*Opening Brace*

Closing Brace

Fig:Compound Statement

Statements

- A compound statement does not need a semicolon.
- If we put semicolon after the compound statement the compiler thinks that we have put an extra null statement.

The Role of Semicolon

- Semicolon is used in two different situations

  1. Every declaration in c is terminated by a semicolon.

  2. Most statements in c are terminated by a semicolon.

- Semicolon should not be used with a preprocessor directive such as the include and define.

UNIT-II
CONTROL STRUCTURES, ARRAYS AND STRINGS

Conditional or Decision Statements

- Decision control statements are used to alter the flow of a sequence of instructions.

- These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not.

- The decision is described to the computer as a conditional statement that can be answered either true or false.

- Different types of control statements are:

  1 . if statement

  2 . if – else statement

  3 . nested if Statement

  4 . else if ladder

  5 . Switch statement

1.If statement or Null Else Statement

- If statement is the simplest form of decision control statements that is frequently used in decision making.

Syntax:

    if (test expression)

    {

       statement 1;

       ....
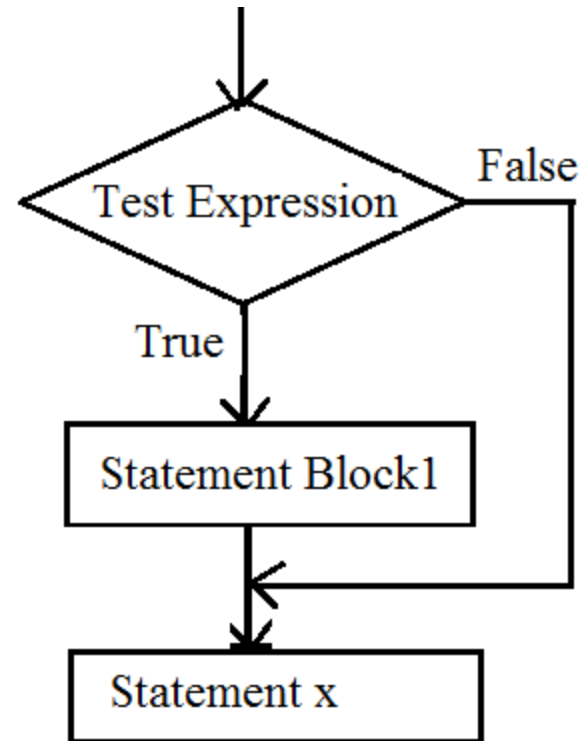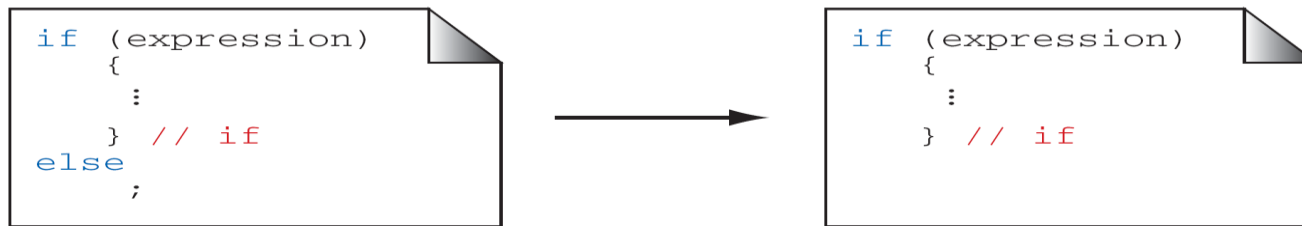
       statement n;

    }

    statement x;

Fig: If Statement flow chart

Statements

- First the test expression is evaluated. If the test expression is true, the statements of if block (statement 1 to n) are executed otherwise these statements will be skipped and the execution will jump to statement x.

- In this case the else statement is not required.

```
if (expression)
    {
      ⋮
    } // if
else
      ;
```

```
if (expression)
    {
      ⋮
    } // if
```

2.If- else Statement

- In the if-else construct, first the test expression is evaluated.

- If the expression is true, statement block 1 is executed and statement block 2 is skipped.

- Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored.

- In any case after the statement block 1 or 2 gets executed the control will pass to statement x.

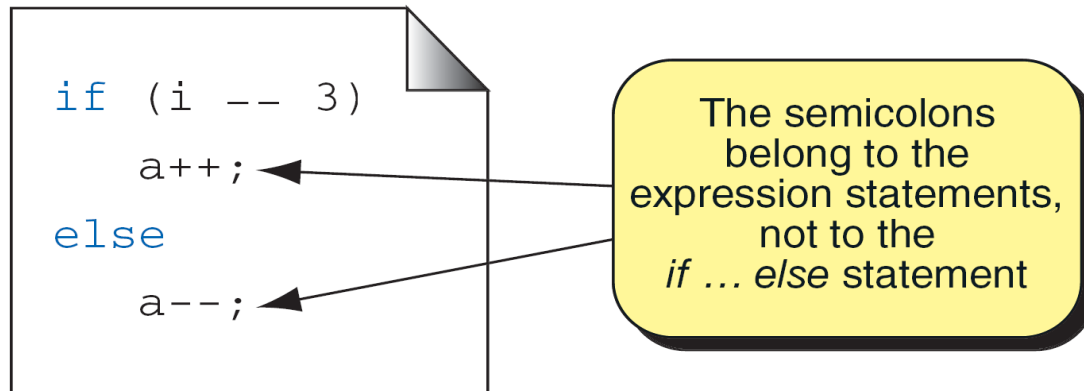# Statements
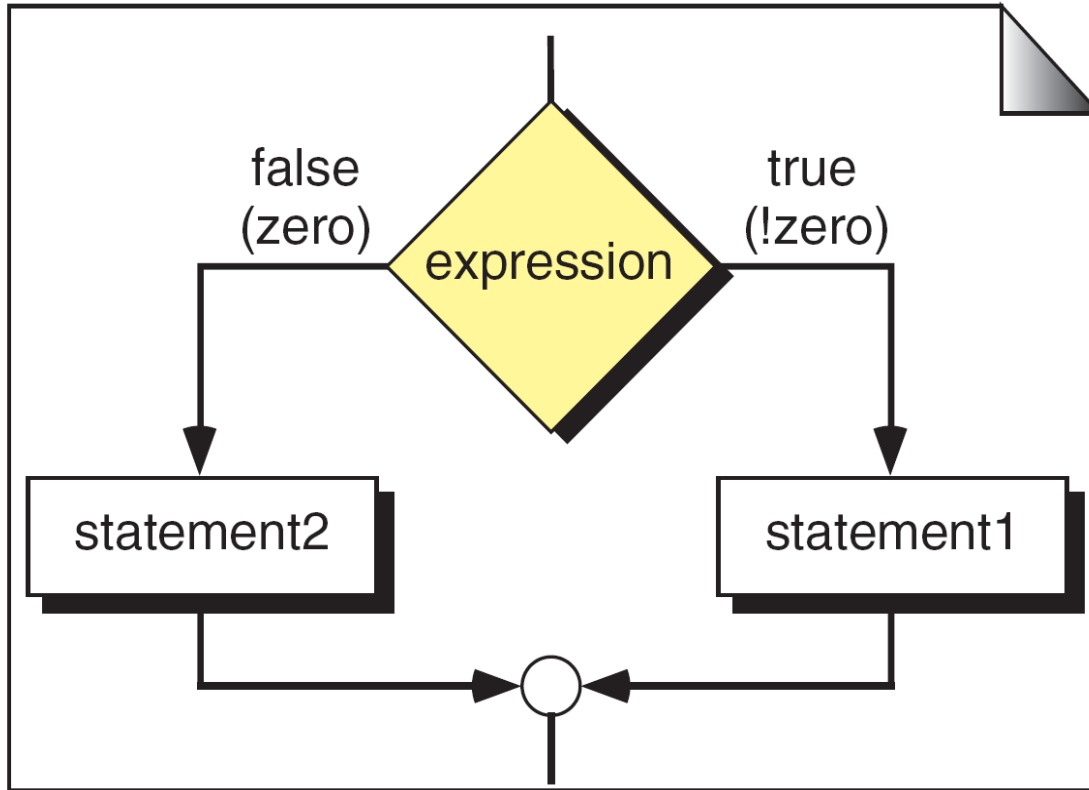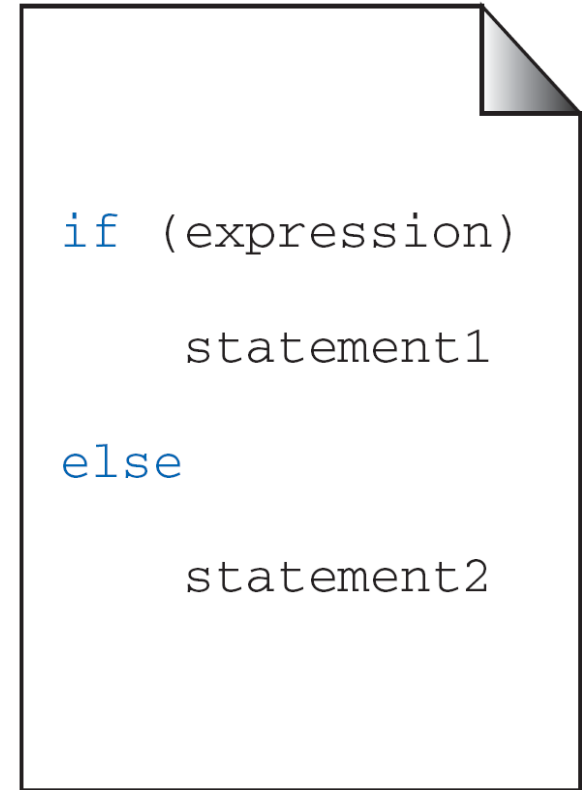
Fig:if...else Logic Flow

Statements

1. The expression must be enclosed in parentheses.

2. No semicolon (;) is needed for an *if...else* statement; statement 1 and statement 2 may have a semicolon as required by their types.

3. The expression can have a side effect.

4. Both the true and the false statements can be any statement (even another *if...else* statement) or they can be a null statement.

5. Both statement 1 and statement 2 must be one and only one statement. Remember, however, that multiple statements can be combined into a compound statement through the use of braces.

6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.

Fig : Syntactical Rules for *if...else* Statements

Statements

- In the first rule the expression must be enclosed in parentheses.
- The second rule no semicolon for if else statement and statement 1 and statement 2 ends with semi colon.

  Ex:

  ```
  if (i -- 3)
        a++;
  else
        a--;
  ```

  The semicolons belong to the expression statements, not to the *if ... else* statement

- The third rule it is common in c code expressions that have side effect.

  Ex: if(++lineCnt > 10){

          pritf("\n");

          lineCnt=0;

       }

       else   printf(....);

- Rules 4 and 5 are related .the multiple statements can be used through the braces.

- The sixth rule states that the true and false statements can be exchanged by complementing the expression.
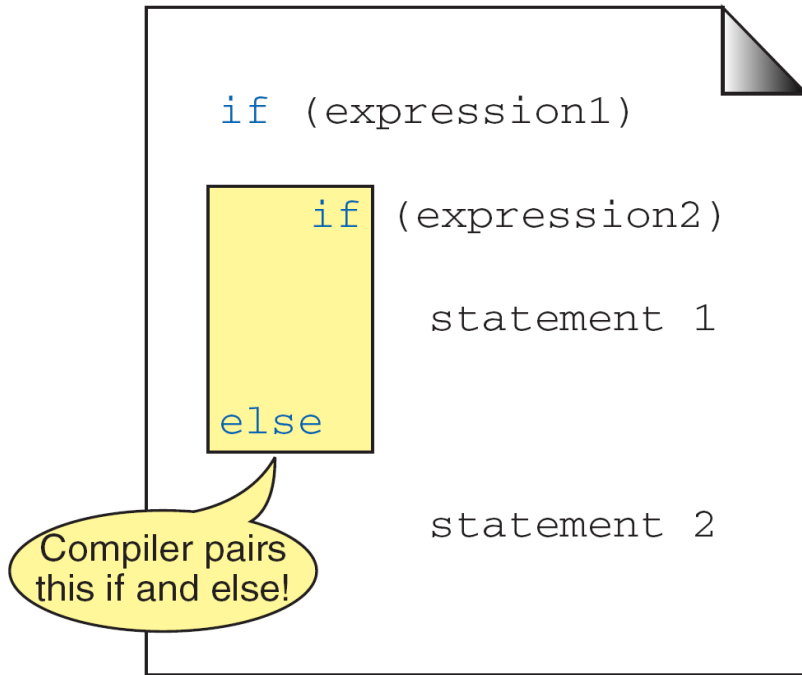
3 . Nested if Statement

- An if else is included within an if else is called nested if statement.

- There is no limit to how many level can be nested , but if the number of levels are increased it becomes more difficult.
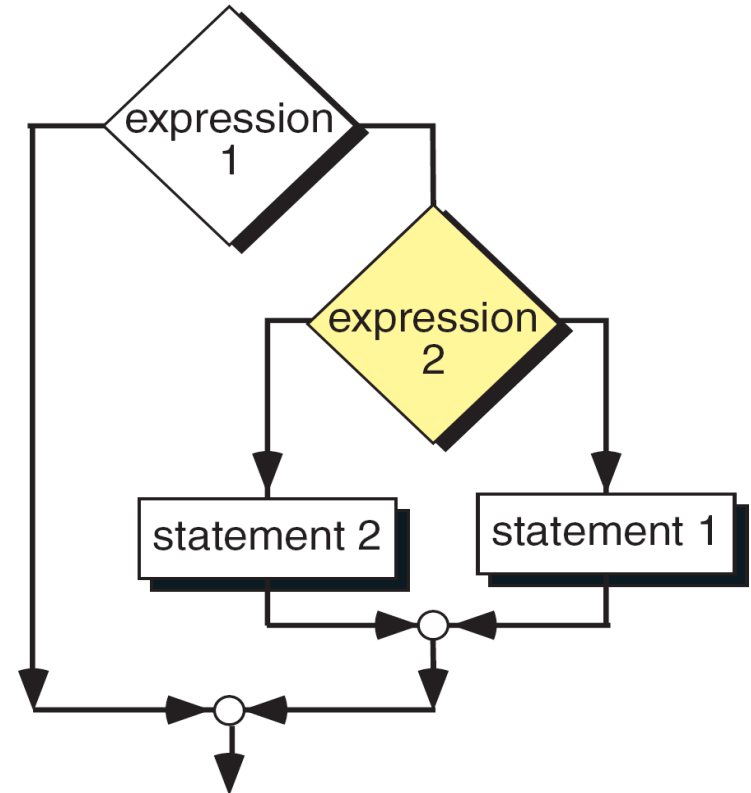
Dangling Else Problem

- when there is no matching else for every if .

```
if (expression1)
    if (expression2)
        statement 1
    else
        statement 2
```

Compiler pairs this if and else!

(a) Code

(b) Logic Flow

Fig : Dangling else

Statements

Solution to Dangling else Problem

- Always pair an else to the most recent un paired if in the current block.

- By using a compound statement , we simply enclose the true actions in braces to make the second if a compound statement .

Conditional Expressions

- An alternative for traditional if –else for two way selection.

- The conditional expression has three operands and a two token operator.

  Syntax: expression1 ? expression2 : expression3

- It first evaluates the left most expression1. If the expression1 is true , then the value of the conditional expression is the value of expression2.

- If the expression1 is   false   , then the value of the conditional expression is the value of expression3.

  Ex:

  a == b ? c++ : d++ ;

Else if ladder

- To make a multi way decision on the basis of a value that not an integral. We go for else if .

- Else if is not a c construct ,it is a style of coding to make a multi way selection based on a value that is not integral.

  Syntax: if(expression)
  ```
          {
              Statements;
          }
       else if(expression2)
          {
              Statements;
          }
       else
          {
              Statements;
          }
  ```
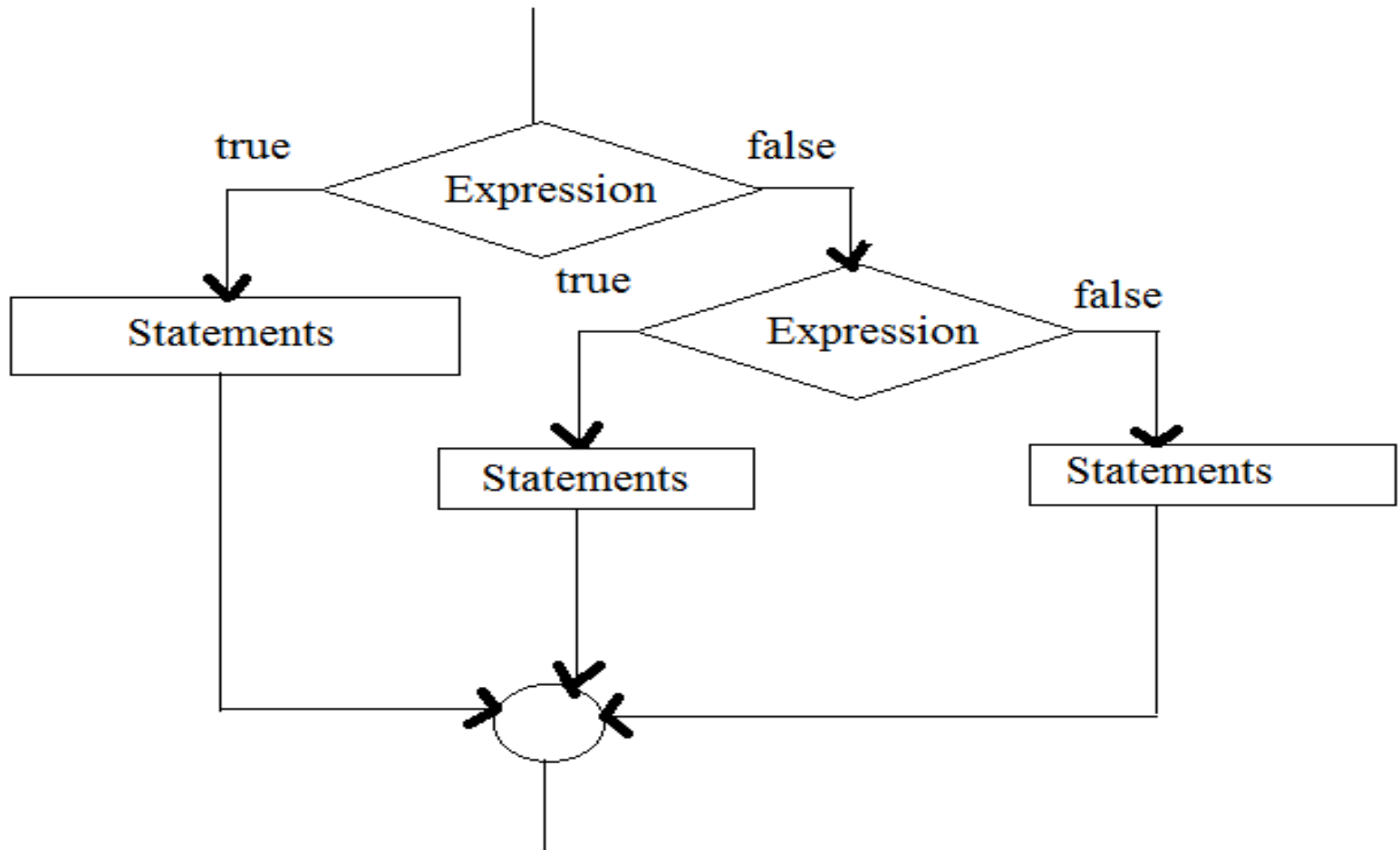
Statements

Fig : Flow chart for else if ladder

Switch

- Switch is a composite statement used to make a decision between many alternatives.

- The switch statement can be used only when the selection condition is reduced to an integral expression.

Statements

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

Fig : switch Statement Syntax

Statements

- The switch expression can use any expression that reduces to an integral value.

- The selection alternatives known as case labels must be integral types.

- Every possible value in the switch expression a separate case label is defined.

- Every thing from a case label to the next case label is sequence.

- Case label simply provides an entry point to start the execution of the code.

- The **default label** is a special form of the case label . It is executed none of the other case values matches the value in the switch expression.

- Once the program enters into a case it executes the code for all the following cases until the end .

Statements

1. The control expression that follows the keyword *switch* must be an integral type.

2. Each *case* label is the keyword *case* followed by a constant expression.

3. No two *case* labels can have the same constant expression value.

4. But two *case* labels can be associated with the same set of actions.

5. The *default* label is not required. If the value of the expression does not match with any labeled constant expression, the control transfers outside of the *switch* statement. However, we recommend that all *switch* statements have a *default* label.

6. The *switch* statement can include at most one *default* label. The *default* label may be coded anywhere, but it is traditionally coded last.

Switch Statement Rules

Loop

- To repeat an operation or a series of operation many times is called looping .

- The loop must terminate when the work is completed.

- Design the loop to check condition before or after the loop.

Pretest Loops

- In each iteration, the control expression is tested first. If it is true, the loop continues; otherwise, the loop is terminated.

Posttest Loops

- In each iteration, the loop action(s) are executed. Then the control expression is tested. If it is true, a new iteration is started; otherwise, the loop terminates.

Statements

Loop Initialization

- Initialization is the process of set the stage for the loop actions.

- It is done before the first execution of the loop body.

- Initialization may be explicit or implicit.

- In an explicit initialization we include code to set the beginning values of key loop variables.

- In an implicit initialization there is no direct initialization code ,it relies on preexisting situation to control the loop.

Loop Update

- The action that causes when the loop is executed is known as loop update.

- Update is done in each iteration , usually in the last action.

- The body of the loop is repeated n times , then the updating is also done n times.

Event Controlled Loops

- An event changes the control expression from true or false.

  Ex :   when Reading data , reaching the end of the data changes the expression from true to false.

Counter – Controlled Loops

- When we know the number of times an action is to be repeated then we use counter controlled loops.

- We must initialize , update , and test the counter.

Statements

- There are three loop statements in c language

  1) do..while loop

  2) while loop

  3) for loop

```
              ┌──────────┐
              │  loops   │
              └────┬─────┘
       ┌───────────┼───────────┐
  ┌─────────┐ ┌─────────┐ ┌─────────┐
  │ do..while│ │  while  │ │   for   │
  └─────────┘ └─────────┘ └─────────┘
```
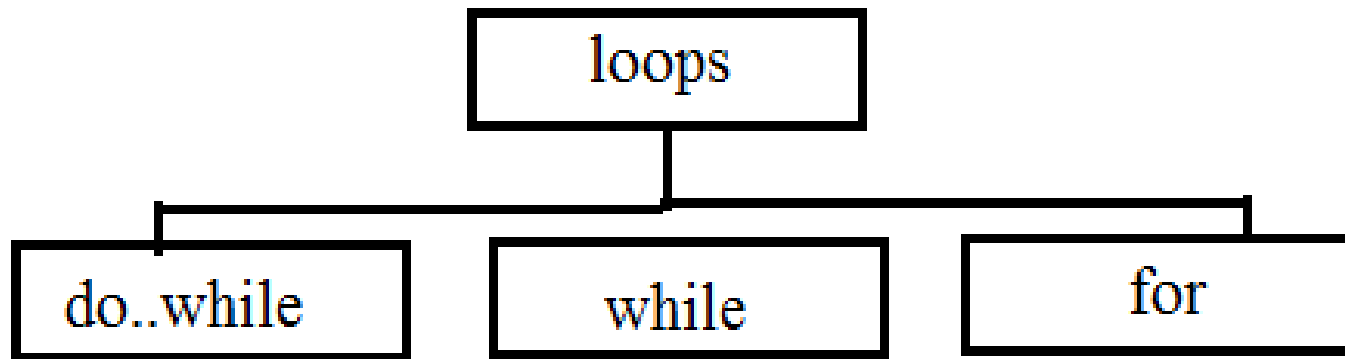
C lopp Constructs

- The first one is post test loop and remaining two are pretest loops.

- All the three loops support event and counter controlled loops.

- While and do while are commonly used for event controlled and for is used for counter controlled.

Do…while loop

- The do while statement is a pretest loop.

- Do while statement test the expression after the execution of the body.

- Do while is concluded with a semicolon.
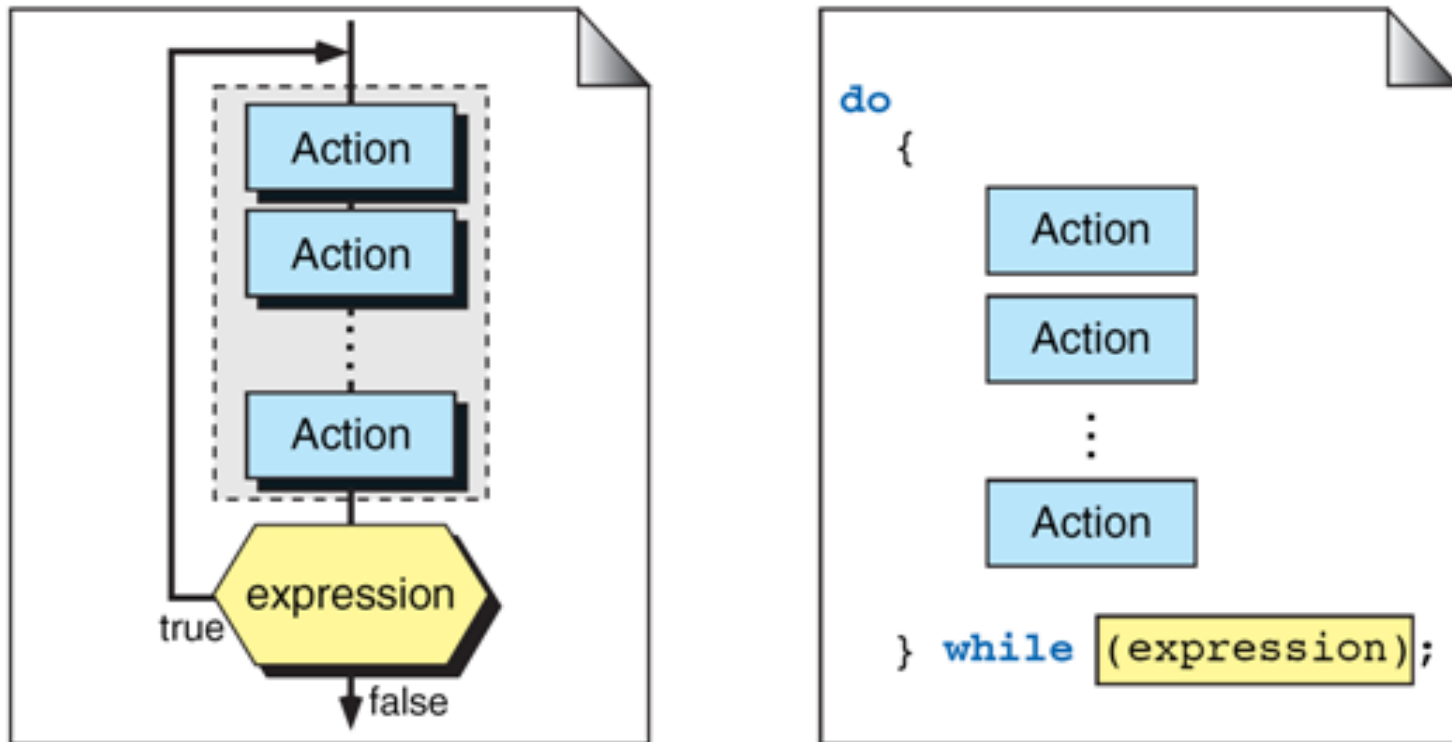
Fig : do…while Statement

While loop

- The while loop is a pretest loop.

- It test the expression before every iteration of the loop.

- No semicolon is needed at the end of the while statement.

- Hear the body of the loop must be only one statement.

Statements

- To include multiple statements in the body , we must put them in a compound statement.

For Loop

- It is a pretest loop.

- It uses three expressions.

- The first expression contains any initialization statements , the second contains the limit-test expression , and the third contains the updating expression.

Syntax:

Initialization   Updating

condition

for(exp1 ; exp2   ; exp3)

Statements

- The body of the for loop must be one and only one statement.
- To include more statements in the body we must code them in a compound statement.
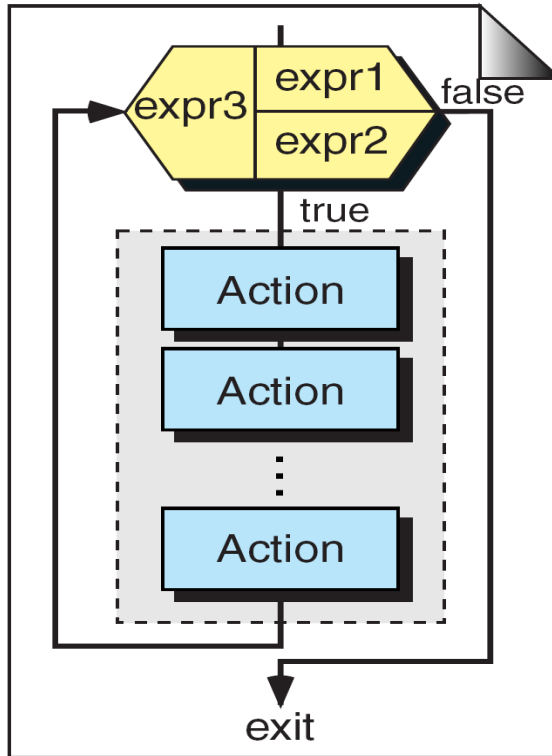
(a) Flowchart
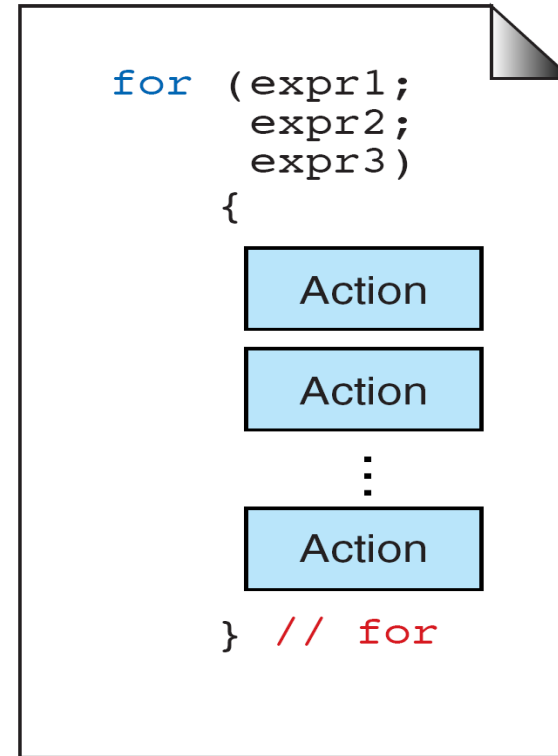
(b) C Language

Fig : Compound for Statement

- A for loop is used when a loop is to be executed a known number of times .

- The same thing can be implemented with a while loop but the for loop is easier to read and more natural for counting loops.
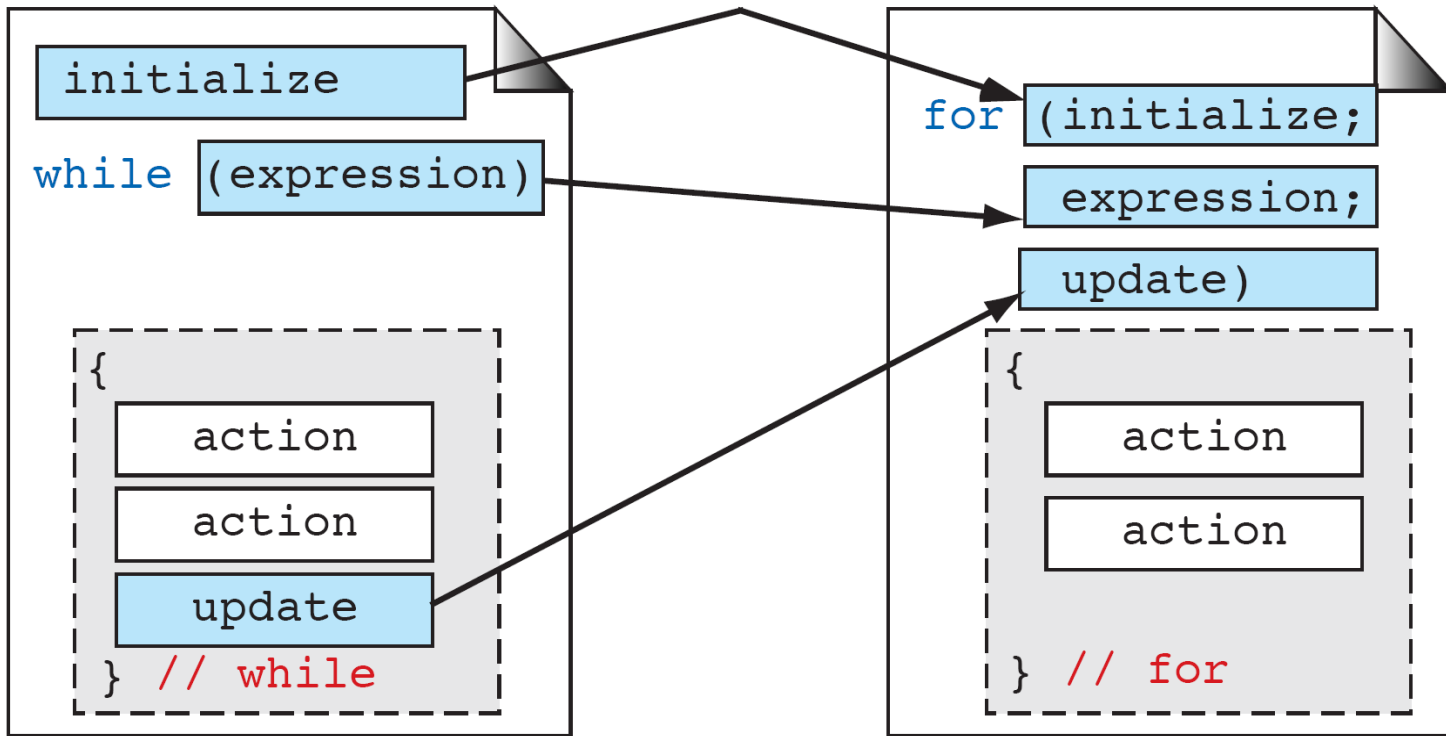
Fig : Comparing for and while Loops

Nested For Loops

- Including the for loop with in the body of another for loop.

- By using nested for loops we create looping applications.

The comma Expression

- The comma expression is a complex expression made up of two expressions separated by a comma.

- The expressions are evaluated left to right.

- The value and type of the expression are the value and type of the right expression.

- The comma expression has the lowest priority of all expressions , i.e is 1.

Ex:

```
for( sum=0,i=1; i<=20; i++)
        sum=sum+i;
```

Fig : Nested Comma Expression

Statements

Break

- The break statement causes a loop to terminate.

- The break statement can be used in any of the loop statements like while , do-while , for and in the selection switch statement.

```
while (condition)
   {
   ...
   for ( ...; ...; ... )
      {
      ...
      if (otherCondition)
          break;
      ...
      } // for

   // more while processing
   ...
   } // while
```

The *break* statement takes us out of the inner loop (the *for* loop). The *while* loop is still active.
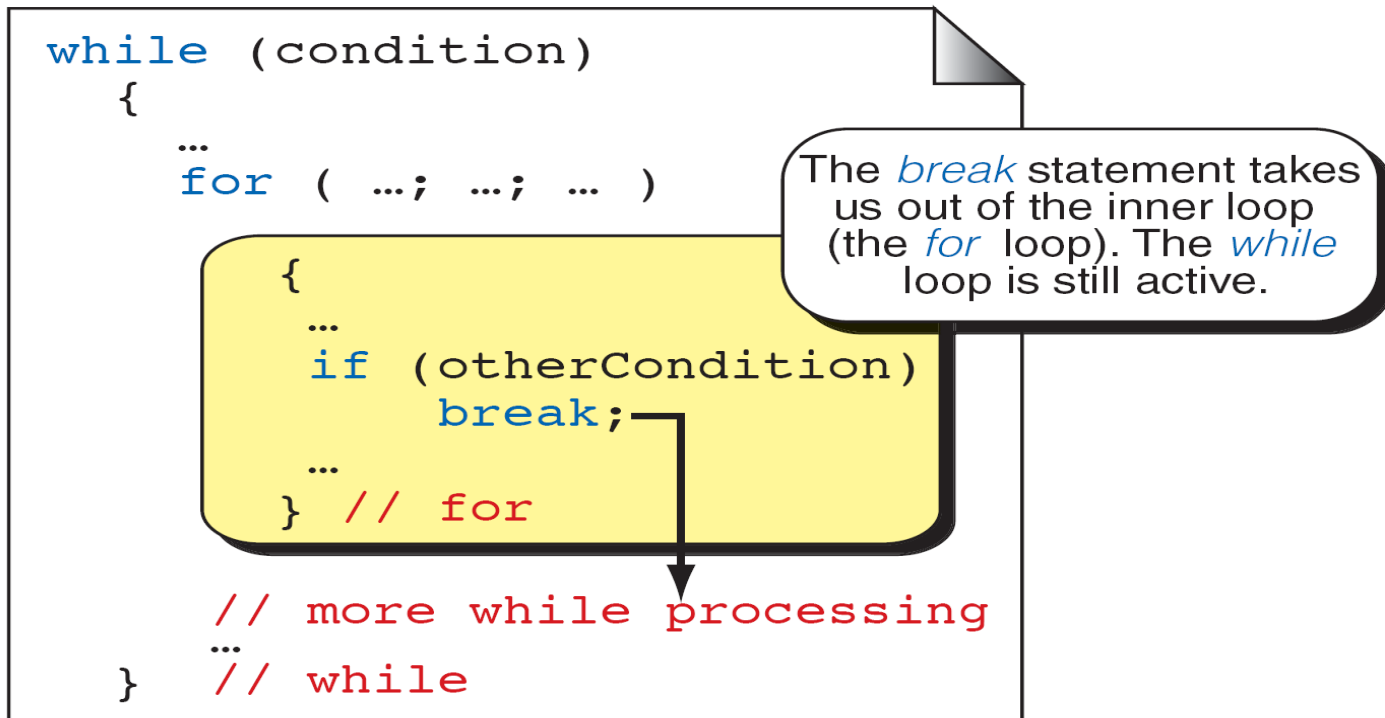
Fig : break and Inner Loops

Statements

```
1   // A bad loop style         // A better loop style
2   for ( ; ; )                  for ( ; !condition ; )
3      {                            {
4       ...                          ...
5        if (condition)            } // for
6           break;
7      } // for
```

```
1   while (x)                    while   (x && !condition)
2      {                            {
3       ...                          ...
4        if (condition)              if (!condition)
5           break;                       ...;
6        else                     } // while
7           ...
8      } // while
```

The for and while as Perpetual Loops

Statements

```
1  breakFlag = 0;
2  while (!breakFlag)
3    {
4    ...
5    if (x && !y || z)              // Complex limit test
6        breakFlag = 1;
7    else
8        ...;
9    } // while
```

Using a break Flag

Continue

- The continue statement does not terminate the loop.
- It simply transfer to the testing expression in while and do-while statements and transfer to the updating expression in a for statement.
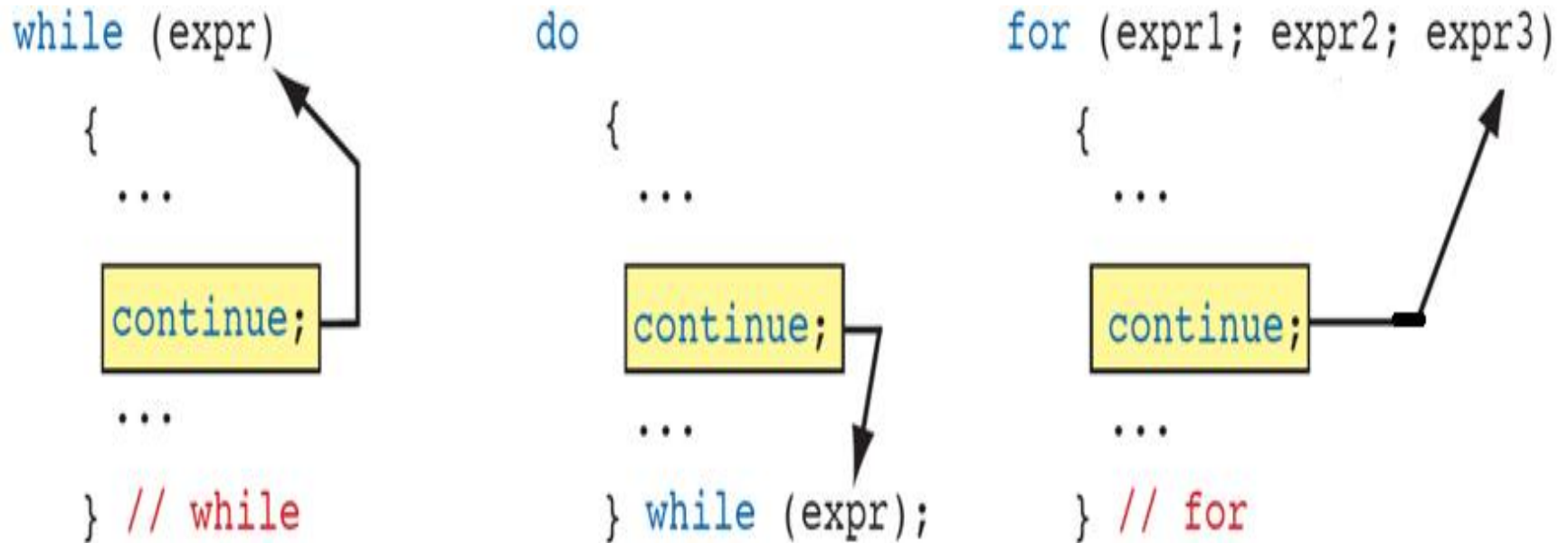

Fig : The continue Statement

goto

- The goto statement is used to transfer control to a specified label.

- Label is an identifier that specifies the place where the branch is to be made. Label can be any valid variable name that is followed by a colon (:).

- label can be placed anywhere in the program either before or after the goto statement. Whenever the goto statement is encountered the control is immediately transferred to the statements following the label.

- If the label is placed after the goto statement then it is called a forward jump and in case it is located before the goto statement, it is said to be a backward jump.

ARRAYS

- **A collection of objects of the *same type* stored contiguously in memory under one name.**

    —**May be type of any kind of variable**

    —**May even be collection of arrays!**

- **The elements of the array are stored in consecutive memory locations and are referenced by an index (subscript).**

- **To refer to an element, specify**

    —**Array name**

    —**Position number**

- **Syntax:**

    array_name[*position number*]

ARRAYS

Array Declaration

- **When declaring arrays**
  - **Name**
  - **Type of data elements**
  - **Number of elements**
- **Syntax**

    Data_Type   array_Name[Number_Of_Elements];

- **Examples:**

    int c[10];

    float myArray[3284];

- **Declaring multiple arrays of same type**
  - **Format similar to regular variables**
  - **Example:**

    int b[100], x[27];

ARRAYS

- **`int c[12]`**
    - An array of ten integers
    - **`c[0]`**, `c[1]`, …, `c[11]`
- **`double B[20]`**
    - An array of twenty long floating point numbers
    - **`B[0]`**, **`B[1]`**, …, **`B[19]`**
- Arrays of **`structs`**, **`unions`**, **`pointers`**, etc., are also allowed
- Array indexes *always* start at zero in *C*

Name of array (Note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

ARRAYS

Two Dimensional Array

- **Syntax**

  Data_Type array_Name[Row_Elements][Column_Elements];

- **Example**

  int D[10][20]

  - **An array of ten rows, each of which is an array of twenty integers**
  - **D[0][0], D[0][1], …, D[1][0], D[1][1], …, D[9][19]**
  - **Not used so often as arrays of pointers**

ARRAYS

Two Dimensional Array

- **Multiple subscripted arrays as**
  - **Tables with rows and columns (m×n array)**
  - **Like matrices: specify row, then column**

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

Column subscript

Array name

Row subscript

ARRAYS

Multi Dimensional Arrays

- **Array declarations read right-to-left**

- **Syntax**

  Data_Type array_Name[Size][Size][Size] … Size];

- **Example**

  int a[10][3][2];

  "an array of ten arrays of three arrays of two elements"

- **In memory**
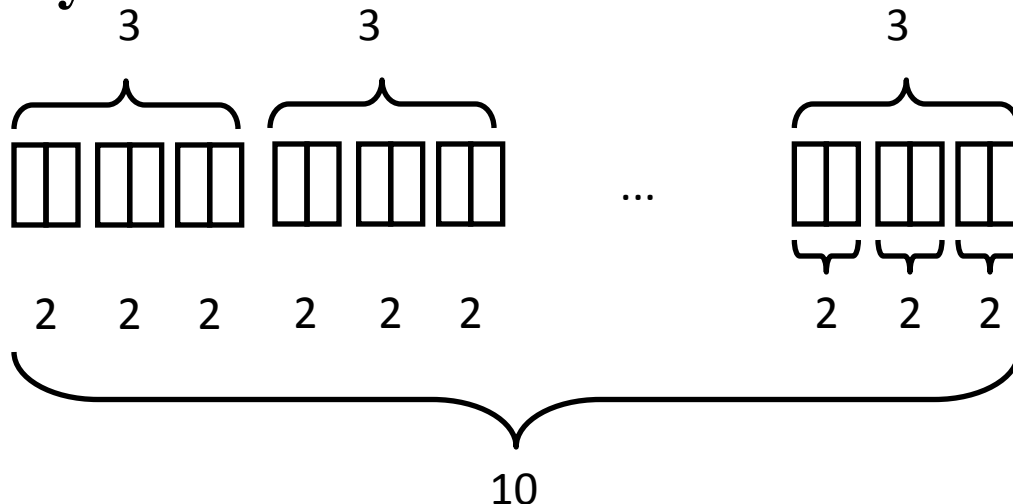
ARRAYS

Array initialization

- **Example**

  int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

  - **Values must be compile-time constants (for static arrays)**
  - **Values may be run-time expressions (for automatic arrays)**

  int A[5] = {2, 4, 8, 16, 32};

    - **Static or automatic**

  int B[20] = {2, 4, 8, 16, 32};

    - **Unspecified elements are guaranteed to be zero**

  int C[4] = {2, 4, 8, 16, 32};

    - **Error — compiler detects too many initial values**

  int D[5] = {2*n, 4*n, 8*n, 16*n, 32*n};

    - **Automatically array initialized to expressions**

ARRAYS

Array initialization

- **Example**

  int n[5] = { 1, 2, 3, 4, 5 };
  - **If not enough initializers, rightmost elements become 0**

  int n[5] = { 0 }
  - **All elements 0**

  int n[ ] = { 1, 2, 3, 4, 5 };
  - **5 initializers, therefore 5 element array**

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

1	2
3	4

  - **Initializers grouped by row in braces**
  - **If not enough, unspecified elements set to zero**

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

1	0
3	4

STRINGS

String is a series of characters treated as a unit.

All string implementations treat a string as a variable length piece of data.

Strings can vary in size.

Length of strings-Strings can be of fixed length or variable length.
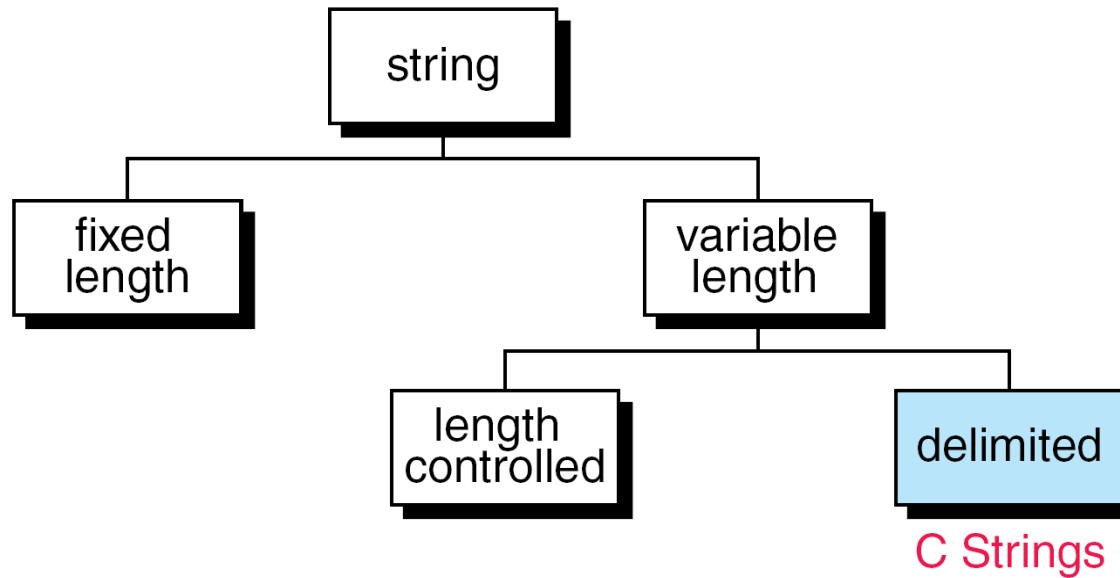
FIGURE String Taxonomy
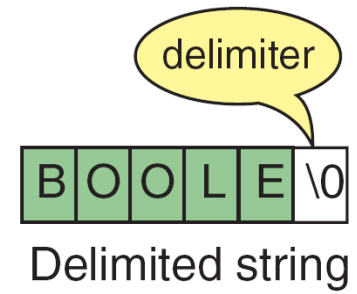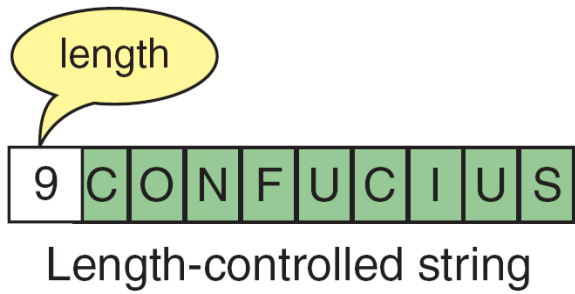
FIGURE   String Formats

Length of the Strings

Fixed length:

Fixed length strings have their length controlled. Their length fixed once cannot be changed.

There are disadvantages with fixed length because if the length is fixed small all the data may not be stored.

If the length is made big memory gets wasted.

Variable length:

The size of the string can be either by length controlled or delimited.

If it is length controlled string of variable length the number of characters in the string are represented before the string.

Length of the Strings

If it is delimiter controlled string of variable length then a '\0' is filled in after the string completes.

| C | O | N | F | U | S | I | O | N | \0 |

C strings

C String is a variable length array of characters that is delimited by the null character.

1) Storing strings:

Strings are stored in an array of characters. It is terminated by the null character(\0).

e.g. **H E L L O \0**

beginning of string

end of string delimiter

| H | e | l | l | o | \0 |
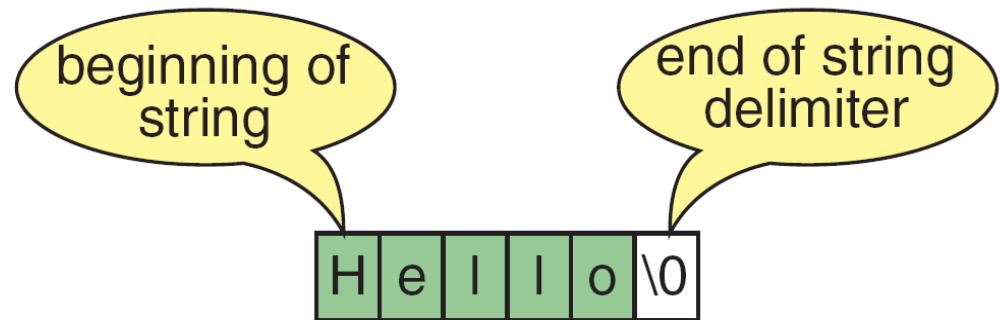
C strings

- for the above characters the memory locations required are 6.
- A character requires only one memory location.
- e.g. 'H' this requires only one memory location.
- A character string requires memory locations based on number of characters and a null character.

e.g. H \0   requires 2 memory locations.

- An empty string is shown as \0.

2) String delimiter

- A string is not a data type but is a data structure since delimiter is used.
- The physical structure of a string is an array but logically it should be stored with delimiter including.

C strings

h	e	l	l	\0

the above is a string.

h	e	l	l	o

This is a array.

g	o	o	d		b	Y	e	\0		

C strings

3) string literal:

 String literal also known as string constant.

It is a sequence of characters enclosed in double quotes.

e.g.   "C is a high-level language."

"hello" , "abcd"

A string is stored in memory just like an object is stored. It has an address.

Using pointers string literals can be referred.

C Strings

Example program

#include<stdio.h>
int main(void)
{
Printf("%c\n","hello"[1]);
Return 0;
}
O/P: E

H	HELLO[0]
E	HELLO[1]
L	HELLO[2]
L	HELLO[3]
O	HELLO[4]
\0	HELLO[5]

C strings

4)Declaring strings

C has no string type.

As strings are sequence of characters char data type is used for string declaring.

e.g. Char str[9];

The size of the string declared should be one memory space more than the data size.

If the size of the data is 6 then including delimeter it is 7. So char str[7];

C STRINGS

To declare a string pointer:

data type is followed by *p and the string name.

e.g. Char *pstr;

This allows in pointing to the first location of the string by a pointer.

Memory of the string must be allocated before the string can be used.

```
// Local Declarations
   char str[9];
```

(a) String Declaration

```
// Local Declarations
   char* pStr;
```

(b) String Pointer Declaration

FIGURE   Defining Strings

5)initializing strings:

a) Char str[9]="good day";

g	o	o	d		d	a	y	\0

str
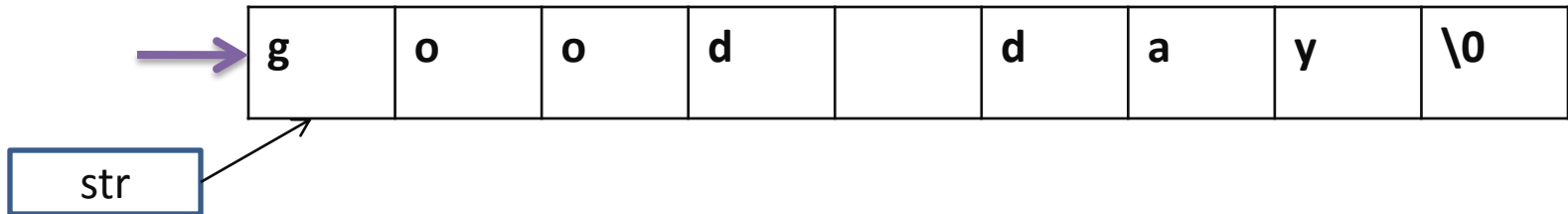
In the above counting with the characters, space and delimeter there are totally 9 characters which are been specified in the array.

C STRINGS

b) Char month[]="Januar";

In the above compiler will create an array of 8 bytes including delimeter .

c) Char *pstr="good day";

"good day"

d)Char str[9]={'G','O','O','D',' ','D','A','Y','\O'}

The method is not used too often because it is so tedious to code.

String Input/Output Functions

- C provides two basic ways to read and write strings.

  1) We can read and write strings with the formatted input/output functions, scanf/fscanf and printf/fprintf.

  2) We can use a special set of string-only functions, get string (gets/fgets) and put string (puts/fputs).
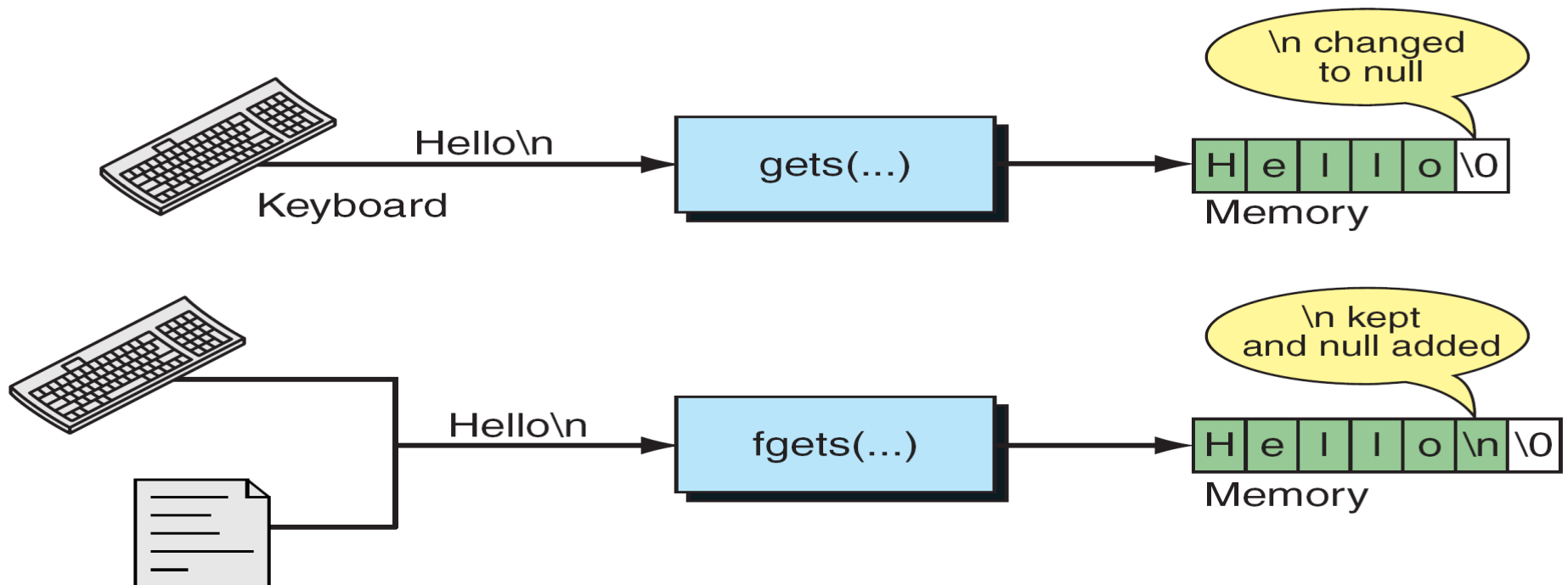
Fig : gets and fgets Functions

String Input/Output Functions
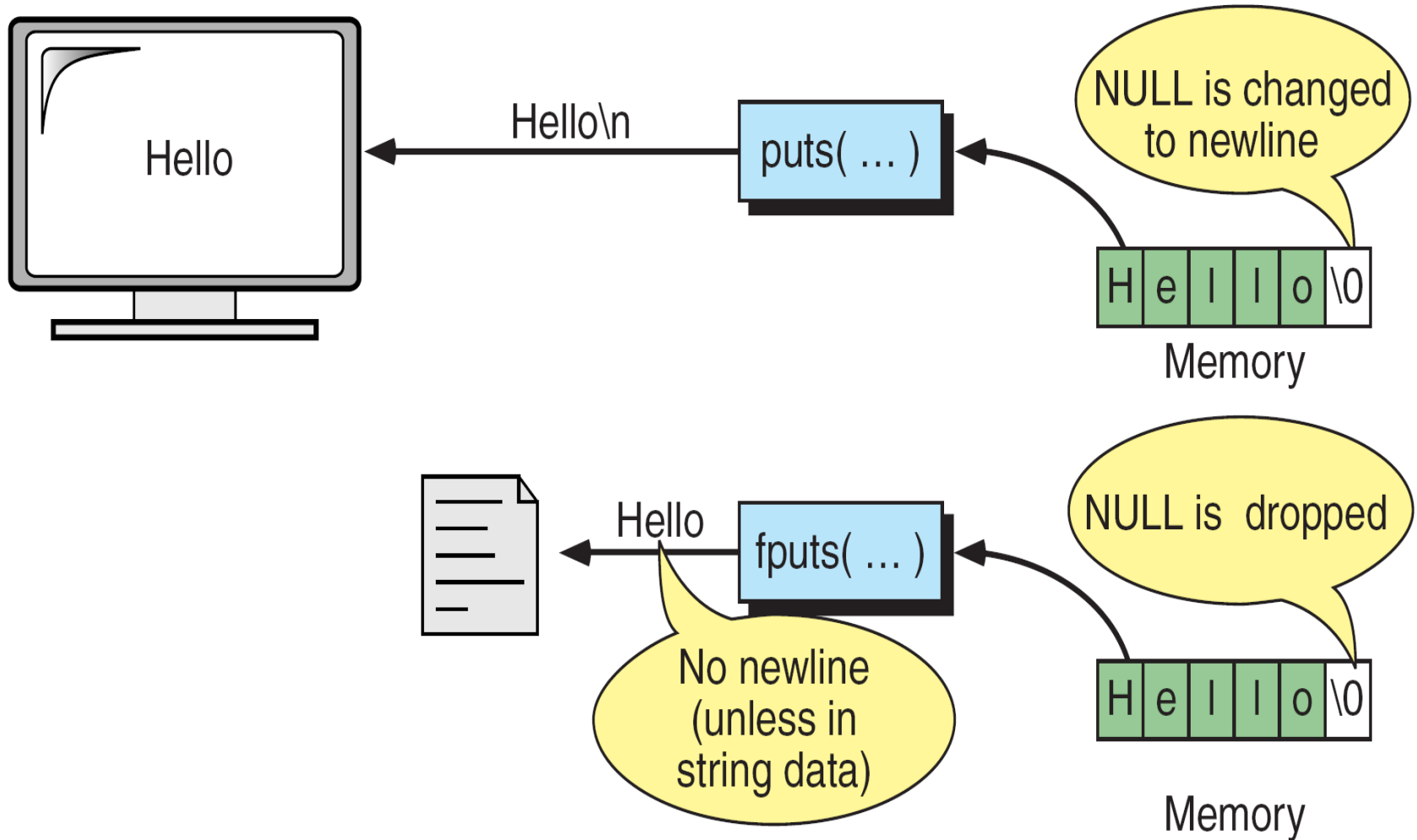
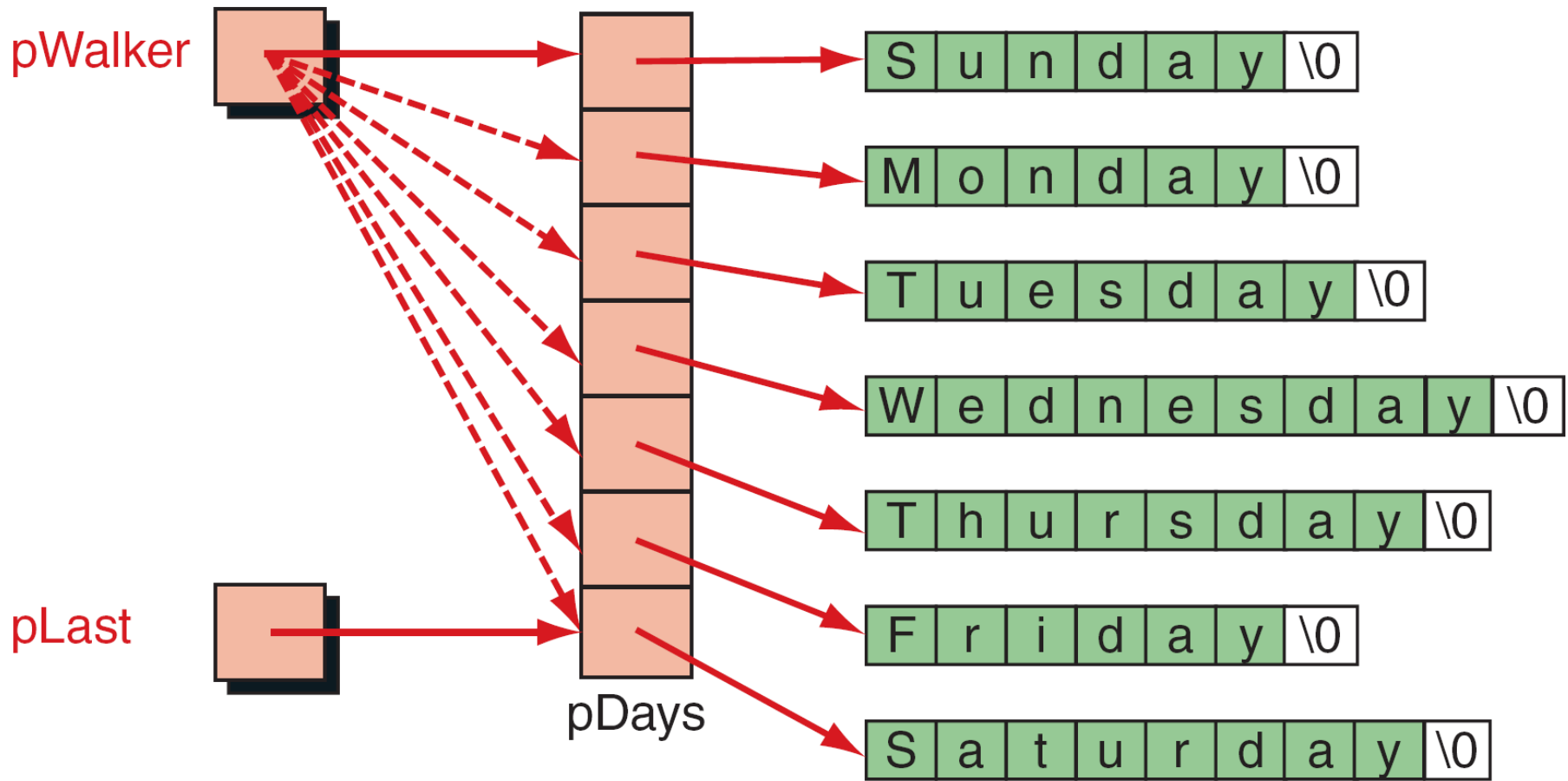Fig : puts and fputs Operations

String Input/Output Functions

Fig : Array of Strings

String Manipulation Functions
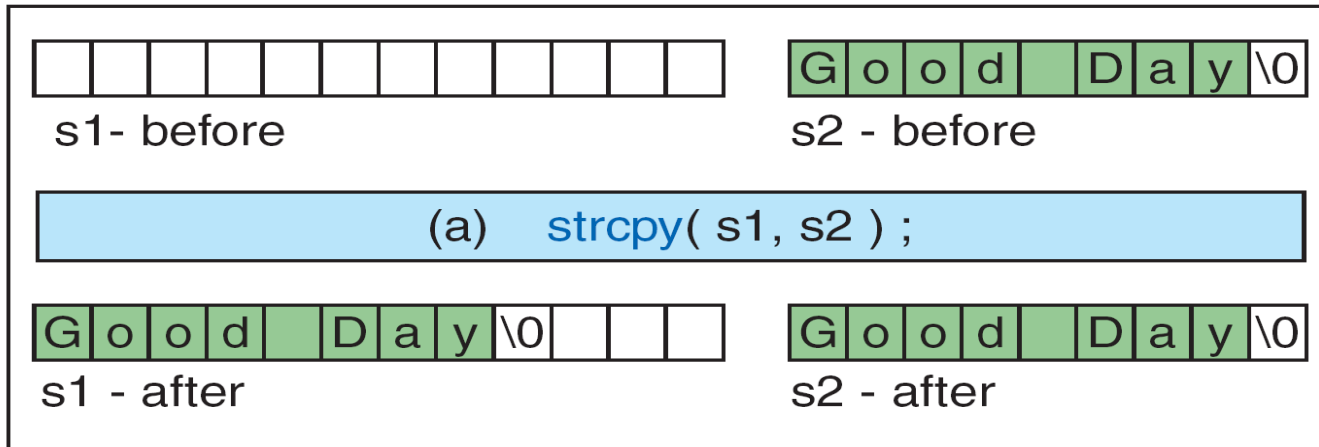
- strcpy()---- copies one string over another.

Syntax:

strcpy(destination,source);
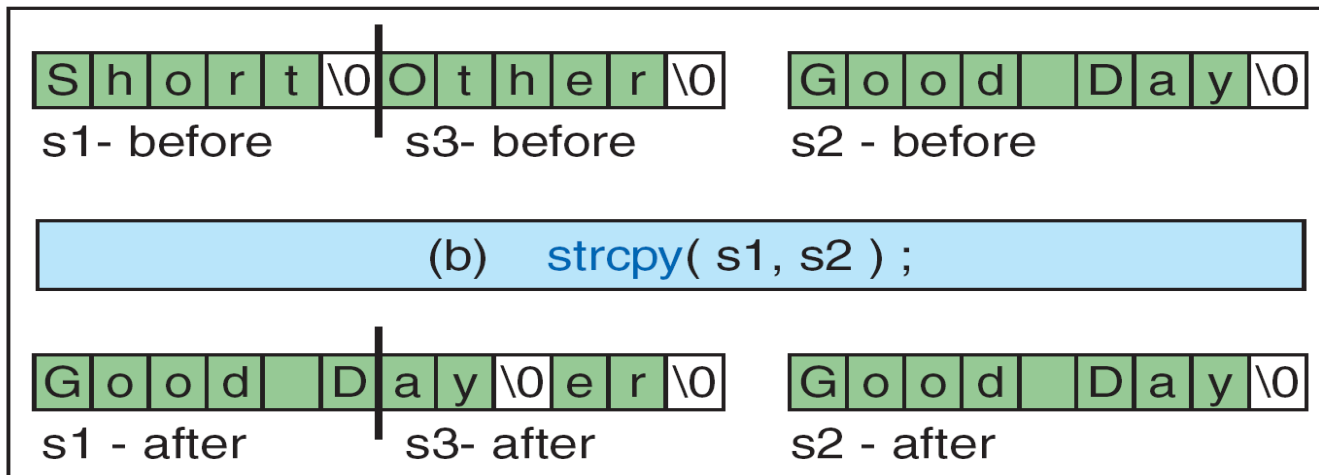
Ex: s1="bombay";   s2="delhi";
        strcpy(s1,s2);

output: s1="delhi";

- The string 1 should large enough to hold the characters of string 2 or else the no. of letters sufficient in the available string will be taken.

String Manipulation Functions

Copying Strings

Copying Long Strings

Fig : String Copy

String Manipulation Functions

- strncpy()---copy characters of a string to another string with the given number of characters in a string.
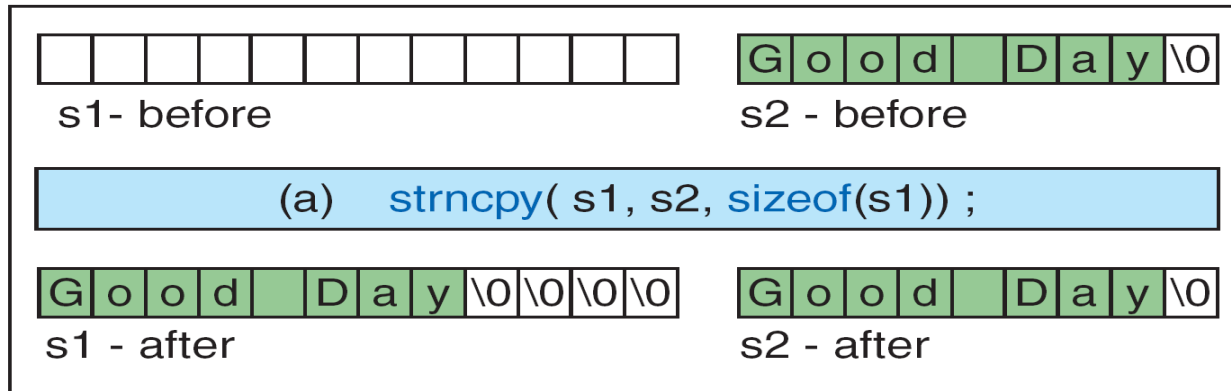

Syntax:

      strncpy(dest,source,n);

Ex:

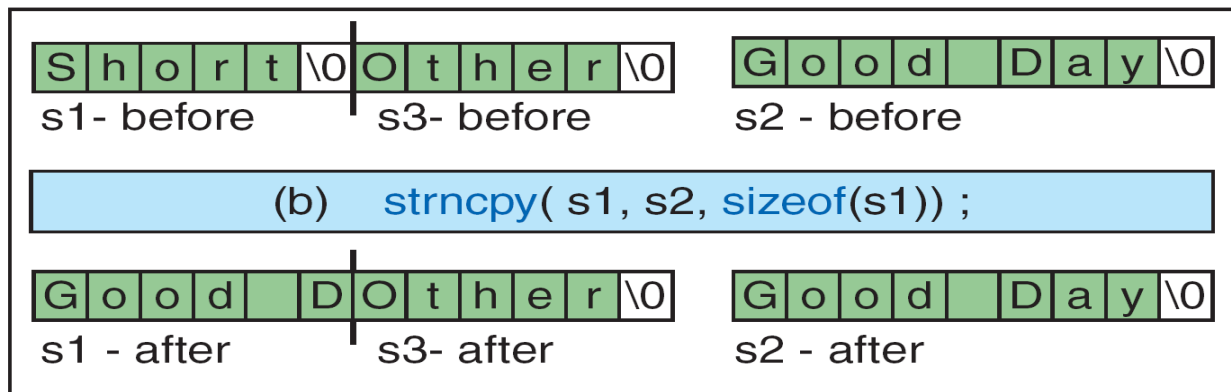   s1="come";   s2="gone";
      strncpy(s2,s1,2);

output:

      s2= "co";

String Manipulation Functions

Copying Strings

Copying Long Strings

Fig : String-number Copy

String Manipulation Functions

- strcmp()----- compares two strings

  Syntax:
  strcmp(string1,string2);

  Ex: s1="their";   s2="there";
       strcmp(s1,s2);

  Output: -9

- The result will be ASCII code of 'i' minus(-) ASCII code of 'r' = -9

- The letters are checked according to dictionary.
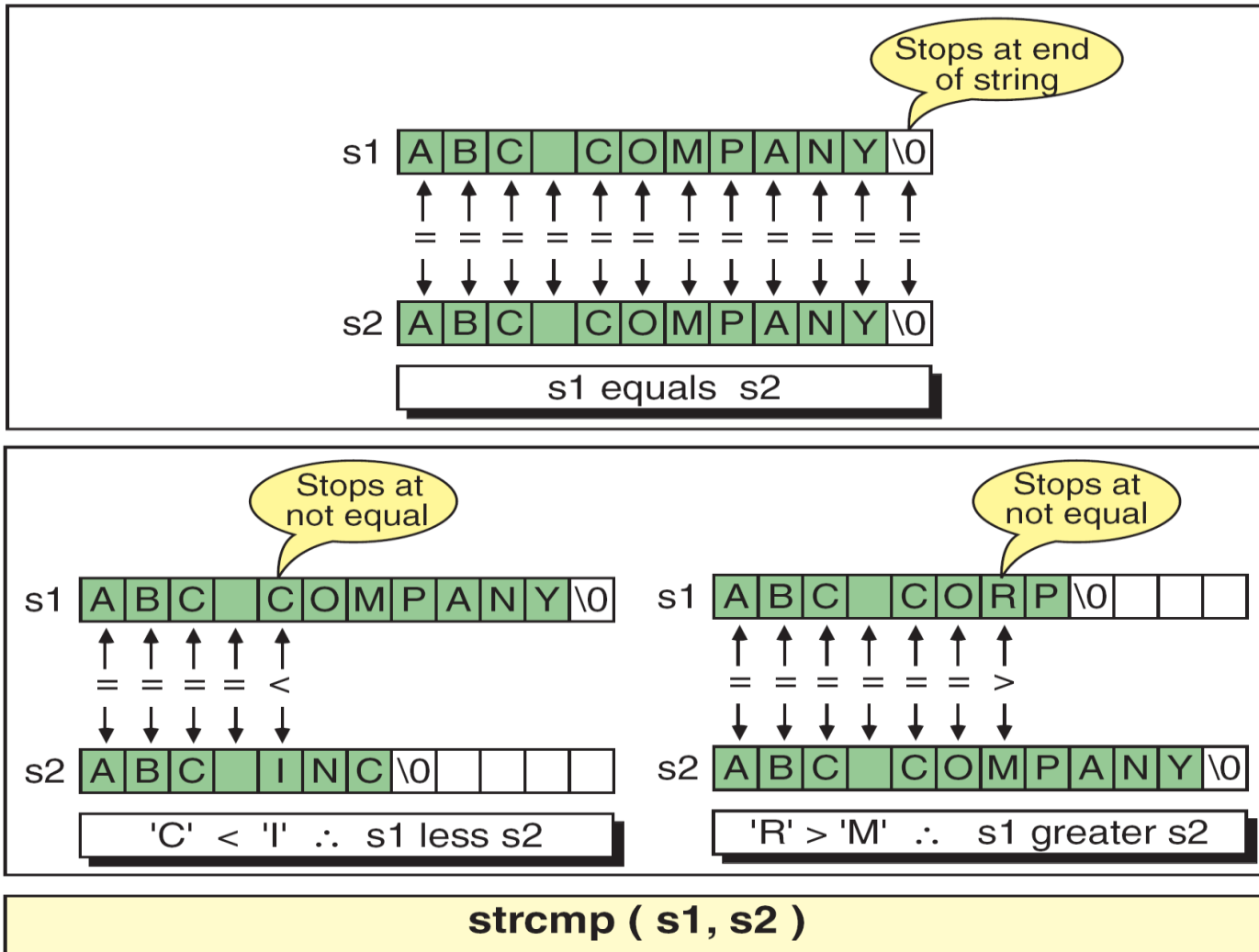
String Manipulation Functions

Fig : String Compares

String Manipulation Functions

- stricmp() ---- doesn't discriminate between small and capital letters.
  syntax:

          Stricmp(source , destination);

  Ex :

          S1="come"   s2="COME"

          Stricmp(s1,s2);

  output: 0

- Compares both string with out considering the case of the string.

- strncmp()---- compares characters of two strings up to specified length
  syntax:

          strncmp(source,destination,length);

   Ex: S1="come" s2="computer"

          Strncmp(s1,s2,2);

  Output: 1

String Manipulation Functions

- strlen()----finds the length of a string
  syntax:

         strlen(string);

  Ex: string=" computer"

             n= strlen(string1);

  Output: 8
- No. of characters in a string.
- strlwr(): converts uppercase characters of a string to lower case
  syntax: strlwr(string);

  Ex :   s2="COMPUTER"

            Strlwr(s2);

  output: s2= computer;

- strupr() --- converts lower case to characters of a string to a upper case.

  Syntax: strupr(string);

  Ex: s1="computer"

         strupr(s1);

  Output: s1="COMPUTER";

String Manipulation Functions

- C library supports a large number of string manipulation functions

- Strcat() -----concatenates two strings
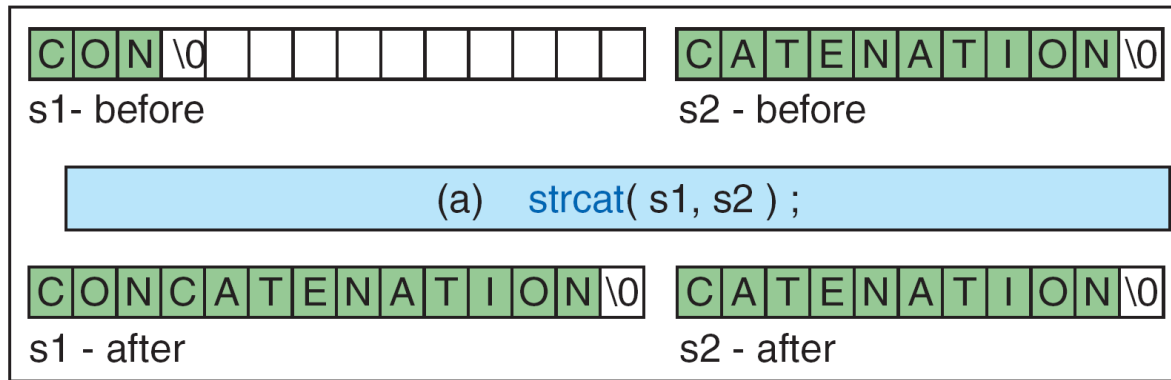
  Syntax:

  strcat(string1, string2);
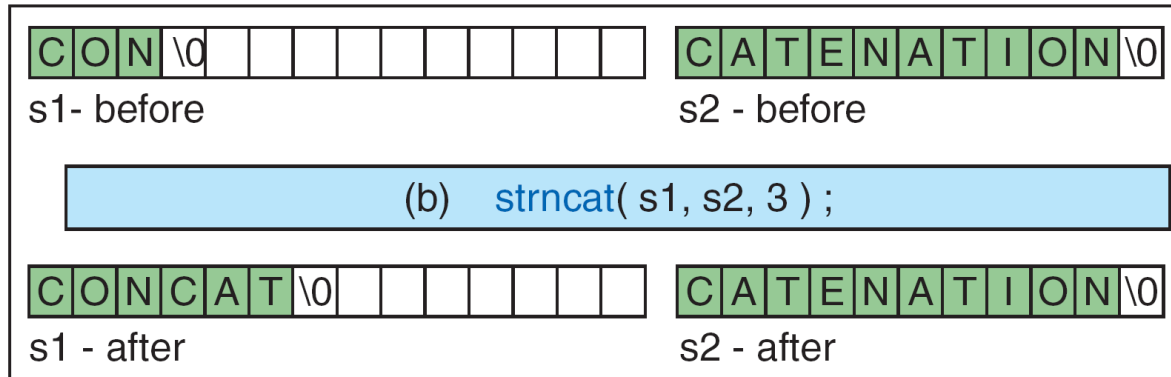
  Ex:

  string1="very";   s2="good";
  strcat(s1,s2);

  Output: s1="verygood";

- The string 1 should large enough to hold the characters of string 2.

String Manipulation Functions

C	O	N	\0															

s1- before

C	A	T	E	N	A	T	I	O	N	\0

s2 - before

(a)   strcat(s1, s2) ;

C	O	N	C	A	T	E	N	A	T	I	O	N	\0

s1 - after

C	A	T	E	N	A	T	I	O	N	\0

s2 - after

String Concatenate

C	O	N	\0													

s1- before

C	A	T	E	N	A	T	I	O	N	\0

s2 - before

(b)   strncat(s1, s2, 3) ;

C	O	N	C	A	T	\0							

s1 - after

C	A	T	E	N	A	T	I	O	N	\0

s2 - after

String N Concatenate

Fig : String Concatenation

String Manipulation Functions

- strchr() — determines the first occurrence of a given character in a string.
  Syntax:

      strchr(string,character);

  Ex:

      Char c='p';
      S1='computer';
      int i= strchr(s1,c);

  Output:

      i=3

- strrchr() ----- determines the last occurrence of the given character in string.

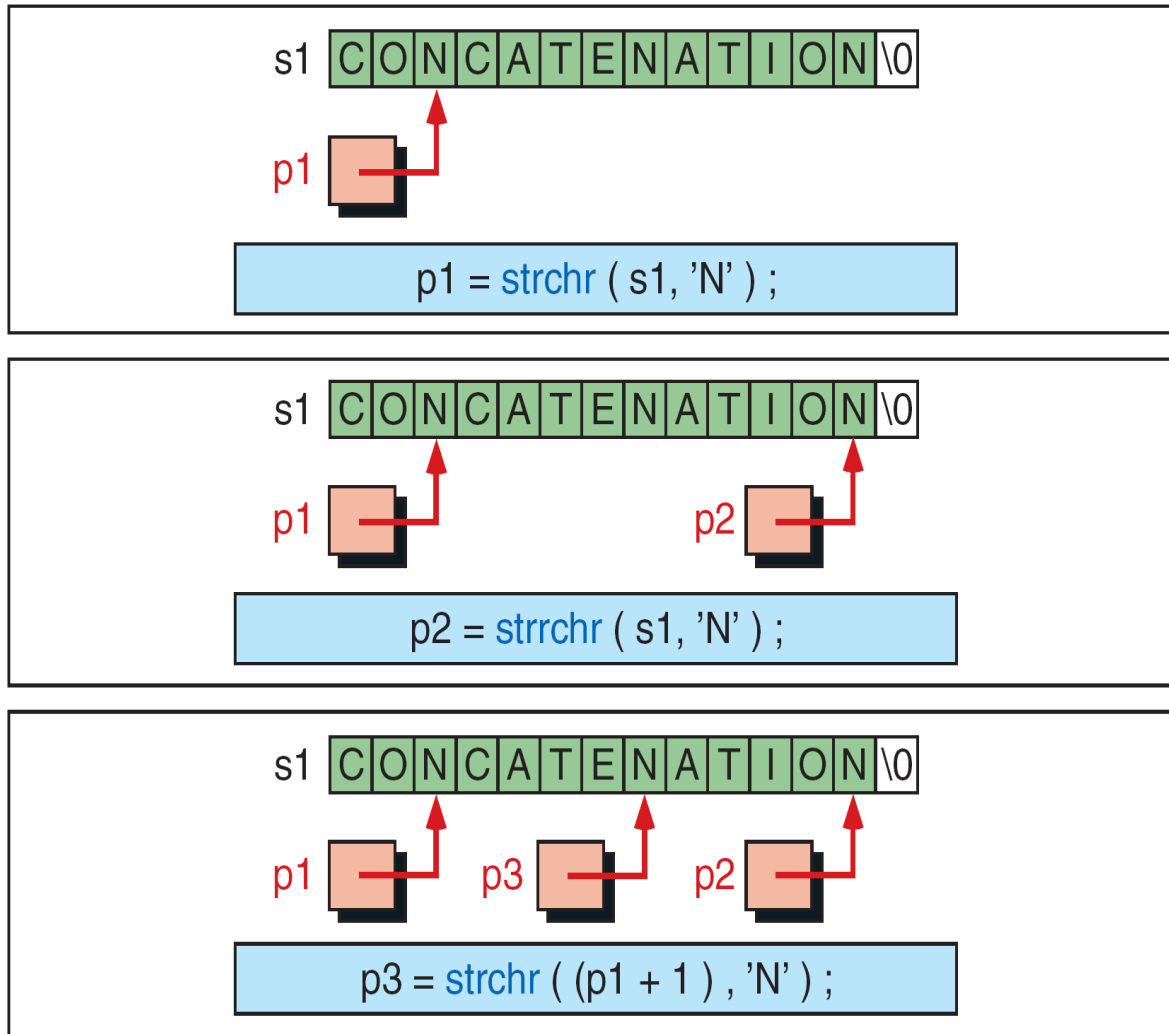   Syntax: strrchr(string, character);

String Manipulation Functions

Fig : Character in String (strchr)

String Manipulation Functions

- strstr() – determines the first occurance of a given string in another string.

    Syntax:

    strstr(string1,string2);

    Ex:

    S1="computer engg";
    S2="engg";
    strstr(s1,s2);

    Output: engg

String Manipulation Functions

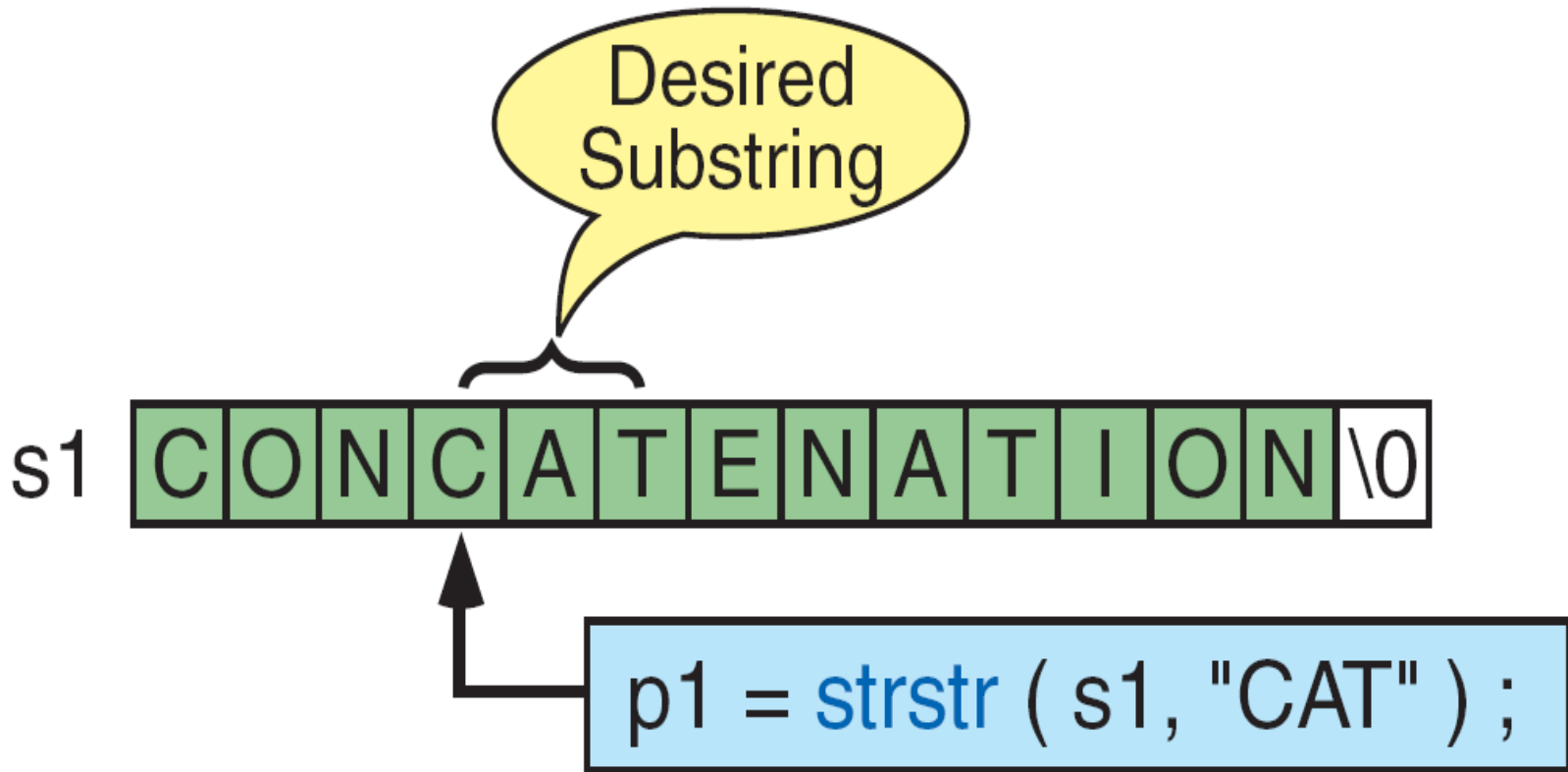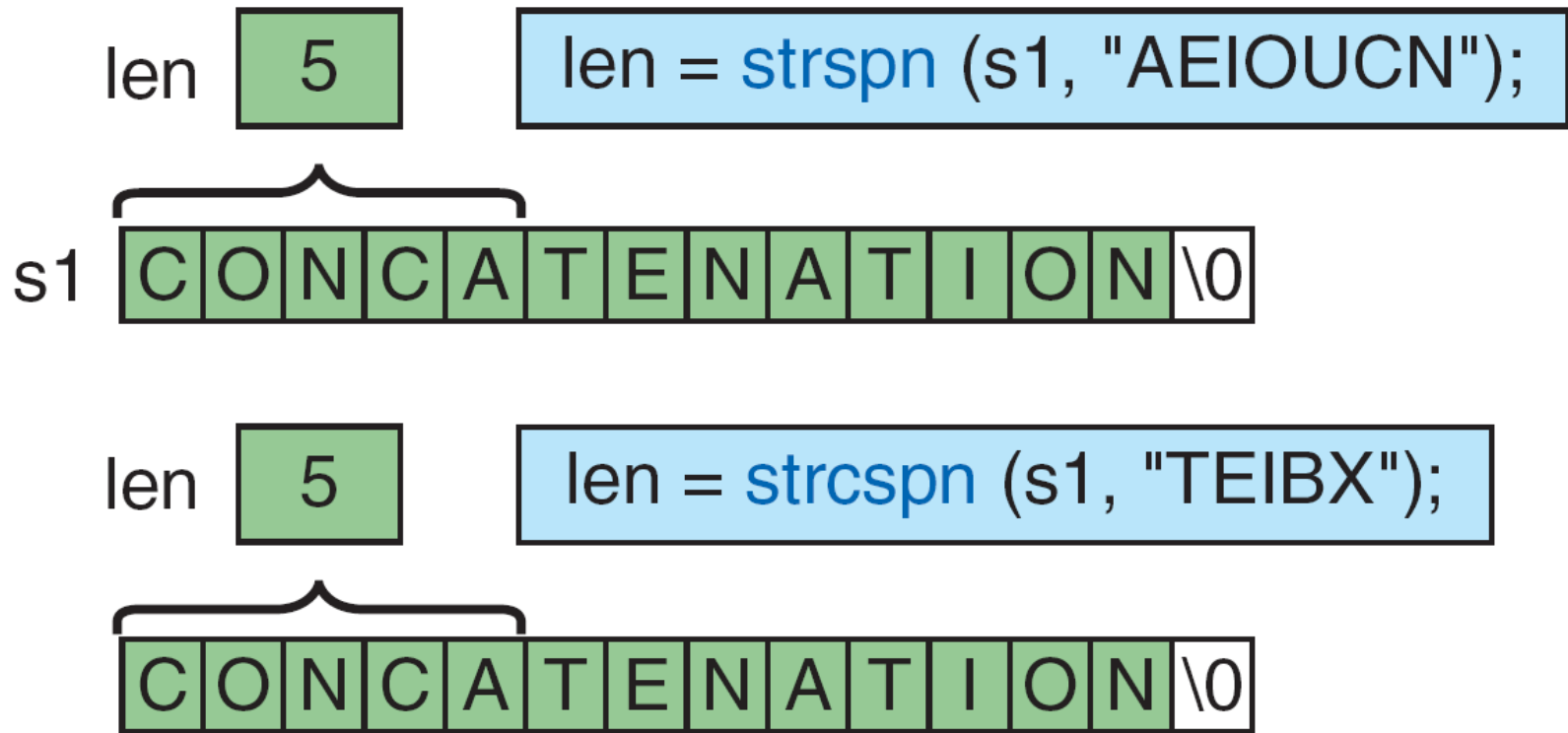Fig :   String in String

String Manipulation Functions

Fig : String Span

String Manipulation Functions

- strrev(): reverses all characters of a   string.

  Syntax: strrev(source);

  Ex :   S1="madam"

              Strrev(s1)

  Output: S1=madam

- strdup(): these function duplicates the string.

  Syntax:

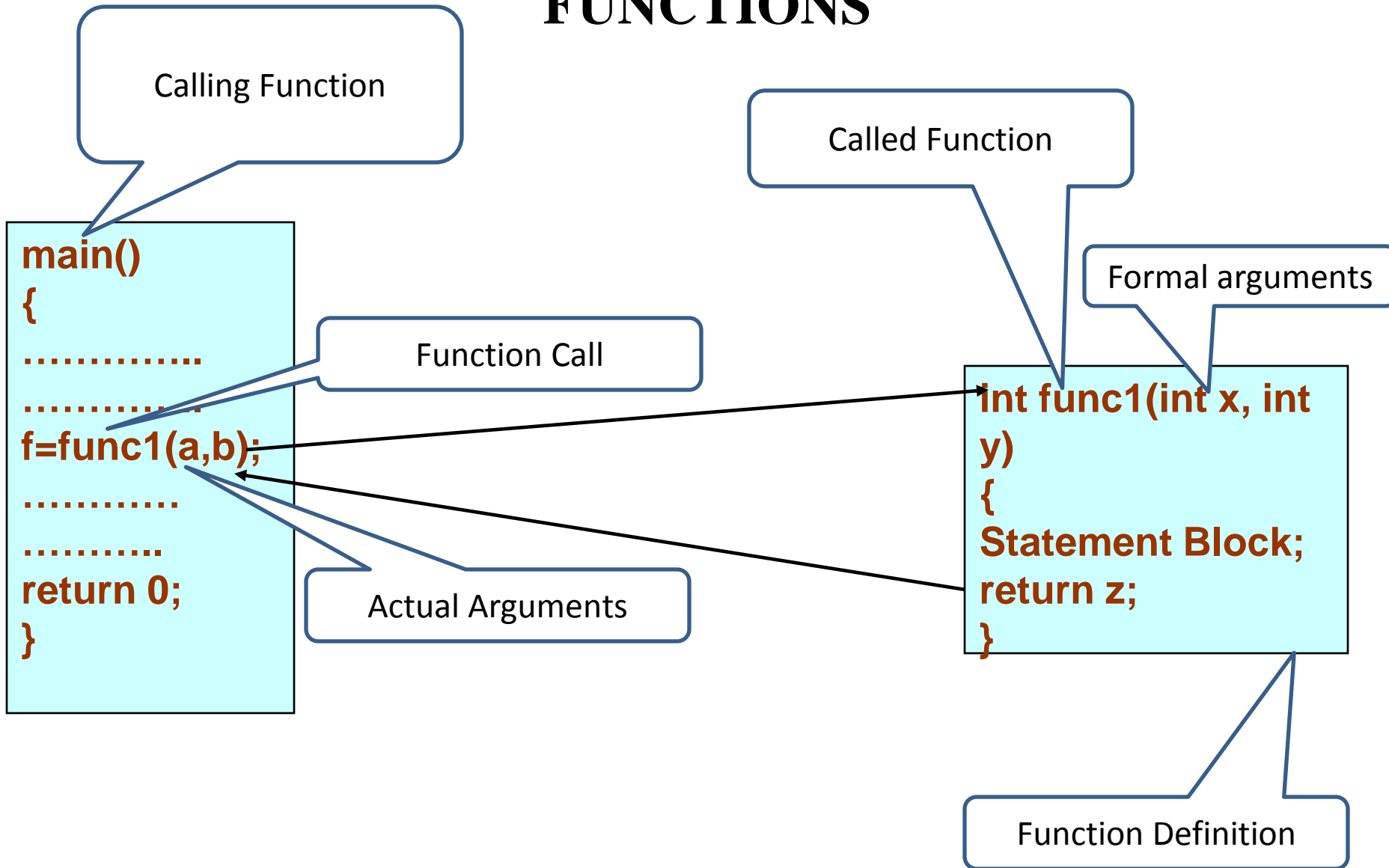   strdup(source);

  Ex: Char s1[10],c[10];

              S1="raju'

              C=strdup(s1)

  Output: C=raju

UNIT-3
FUNCTIONS AND POINTERS

- **Breaking up a program into segments commonly known as** *functions***.**

- **Each function can be written more or less independently of the others.**

- **The purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.**

- **The** *side effect* **of a function is an action that results in a change in the state of a program. If occurs, it effects while the function is executing and before the function returns.**

FUNCTIONS

Calling Function

Called Function

Formal arguments

Function Call

```
main()
{
…………..
…………..
f=func1(a,b);
…………
………..
return 0;
}
```

```
int func1(int x, int y)
{
Statement Block;
return z;
}
```

Actual Arguments

Function Definition

Fig: Terminology of Functions

FUNCTIONS

Terminology

- **main() calls another function, func1() to perform a well defined task. main() is known as the *calling function* and func1() is known as the *called function*.**

- **When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.**

- **After the called function is executed, the control is returned back to the calling program.**

- **The inputs that the function takes are known as *arguments* .**

FUNCTIONS

Terminology

- **When a called function returns some result back to the calling function, it is said to *return* that result.**

- **Main() is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.**

- **The arguments in function call referred as *actual arguments*.**

- **The arguments in function definition call referred as *formal arguments*.**

FUNCTIONS

- **main() can call as many functions as it wants and as many times as it wants.**

- **It is not that only the main() can call another functions. Any function can call any other function.**
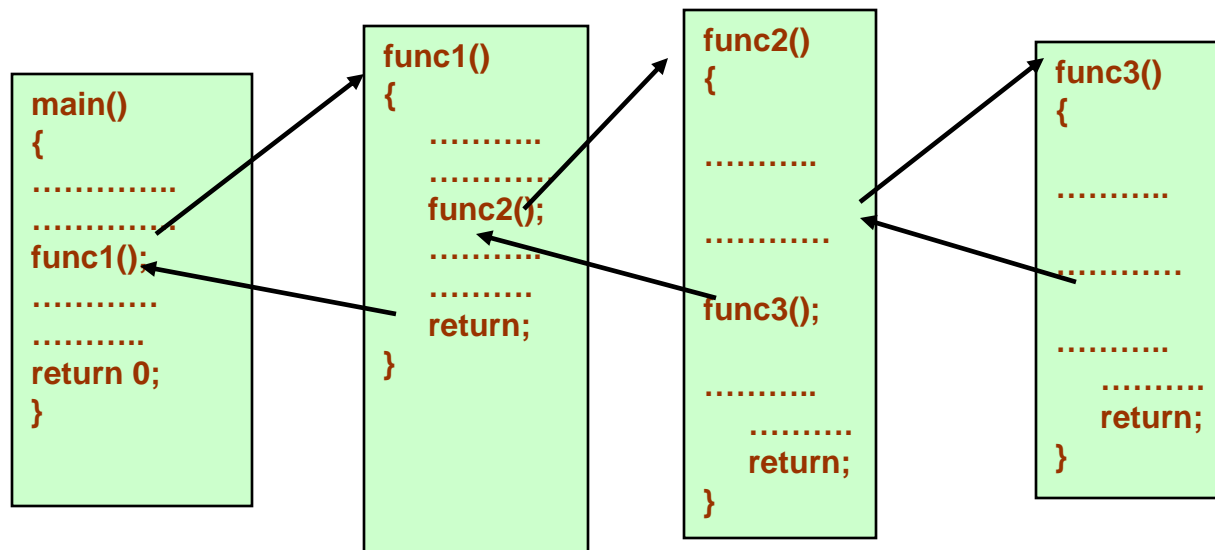
Fig: Multi function call illustration

FUNCTIONS

Advantages of functions

- **Each function to be written and tested separately. This simplifies the process of getting the total program to work.**

- **Understanding, coding and testing multiple separate functions are easier than one huge function.**

  - **Provide a way to *reuse* code that is required in more than one place in a program.**

  - **A library of functions can be built to carry out *repetitive work* like math library, standard I/O library.**

FUNCTIONS

- *Advantages of functions*

  - **Functions can *protect the data*.**

  - **Functions can have local data described within a function. These data are available only to the function and only while the *function is executing*. When the function is *not active*, the data are *not accessible*.**

  - **With local data, data in one function cannot be seen or changed by a function *outside of its scope*.**

FUNCTIONS

Function Declaration / Prototype

- *Function declaration* **identifies a function with its name, a list of arguments that it accepts and the type of data it returns.**

- *Syntax:*

  return_type   function_name(data_type   variable1,   data_type variable2,..);

- **No function can be declared within the body of another function.**

  Eg: float avg(int , int);

FUNCTIONS

Function Definition

- **Function definition consists of a *function header* that identifies the function, followed by the *function body* containing the executable code for that function.**

- **Syntax:**

return_type function_name(data_type variable1, data_type variable2,..)

{   //function body

………….
statements
 ………….
return(variable);

}

FUNCTIONS

Function Definition

- **When a function defined, space is allocated for that function in the *memory.***

- **The number of and the order of arguments in the function header must be same as that given in function declaration statement.**

FUNCTIONS

Function call

- **The function call statement *invokes* the function.**

- **When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.**

- **Once the called function is executed, the program control passes back to the calling function.**

- **Syntax:**

  function_name(variable1, variable2, …);

FUNCTIONS

Function Call

- **Names (not the types) of variables in function declaration, function call and function definition may vary.**

- **Arguments may be passed in the form of expressions to the called function.**

- **In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.**

- **If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.**

variable_name = function_name(variable1, variable2, …);

FUNCTIONS

Function Call


```c
// Function Declaration
void greeting (void);
int main (void)
{
// Statements
    greeting( );    // call
    return 0;
} // main
```

Back to Operating System

```c
// Function Definition
void greeting (void)
{
    printf("Hello World!");
    return;
} // greeting
```

Hello World

Side Effect

Fig: Declaring, Calling, and Defining Functions

FUNCTIONS

Multiple Function Call

```
#include<stdio.h>
void y();
void y()
{
  printf("y");
}
void main()
{
  void a(), b(), c(), d();
  clrscr();
  y();
  a();
  b();
  c();
  d();
}
```

```
void a()
{
  printf(" a ");
  y();
}

void b()
{
  printf(" b ");
  a();
}

void c()
{
  a();
  b();
  printf(" c ");
}
```

```
void d()
{
  printf(" d ");
  c();
  b();
  a();
}
```

OUTPUT:
y a y b a y a y b a y c
d a y b a y c b a y a y

FUNCTIONS

Return Statement

- The return statement is used to terminate the execution of a function and return control to the calling function.

- When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.

- A return statement may or may not return a value to the calling function.

- Syntax:

return <expression> ;

FUNCTIONS

Return Statement

- The value of *expression*, if present, is returned to the calling function. However, in case *expression* is omitted, the return value of the function is undefined.

- Programmer may or may not place the *expression* within parentheses (,).

- By default, the return type of a function is *int*.

- Functions that has no return statement, the control automatically returns to the calling function after the last statement of the called function is executed.

FUNCTIONS

Passing Parameters To The Function

- **There are two ways in which arguments(parameters) can be passed to the called function.**
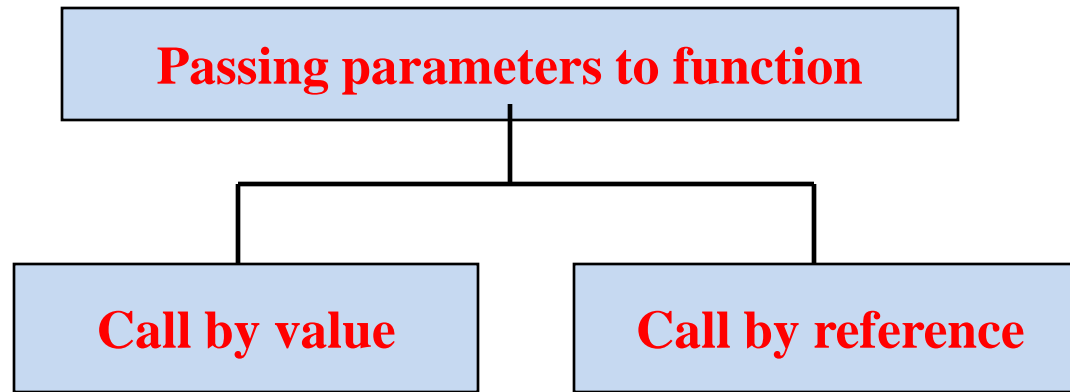
- *Call by value* **in which values of the variables are passed by the calling function to the called function.**

- *Call by reference* **in which address of the variables are passed by the calling function to the called function.**

FUNCTIONS

Passing Parameters To The Function

— **Call By Value:**

- **The called function creates new variables (formal arguments) to store the value of the arguments passed to it. Therefore, the called function uses a separate copy of the *actual arguments* (formal arguments) to perform its intended task.**

- **If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.**

FUNCTIONS

Passing Parameters To The Function – Call By Value

#include<stdio.h>
 	
void add(int n);
 	
int main()
 {
 	
   int num = 2;
   printf("\n The value of num before calling the function = %d", num);
   add(num);
   printf("\n The value of num after calling the function = %d", num);
   return 0;
 	
}
	void add(int n)
	{
	
	   n = n + 10;
	   printf("\n The value of num in the called function = %d", n);
	
	}
		OUTPUT :
		
		The value of num before calling the function = 2
		
		The value of num in the called function = 12
		
		The value of num after calling the function = 2

FUNCTIONS

Passing Parameters To The Function

— **Call By Reference**

- **In call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement.**

- **The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.**

- **In call by reference, we declare the function parameters as references rather than normal variables.**

FUNCTIONS

Passing Parameters To The Function

— **Call By Reference**

- **Any changes made by the function to the arguments it received are visible by the calling program.**

- **To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list.**

FUNCTIONS

Passing Parameters To The Function --Call By References

```c
#include<stdio.h>
void add( int *n);
int main()
{

    int num = 2;

    printf("\n The value of
    num before calling the
    function = %d", num);

    add(&num);

    printf("\n The value of
    num after calling the
    function = %d", num);

    return 0;
}
```

```c
void add( int *n)
{

    *n = *n + 10;
    printf("\n The value
    of num in the called
    function = %d", *n);
}
```

Output:

The value of num before calling the function = **2**

The value of num in the called function = **12**

The value of num after calling the function = **12**

FUNCTIONS

User Defined Functions

Five types of functions are possible:

- **Functions with no arguments and no return values.**

- **Functions with arguments and no return values.**

- **Functions with arguments and return values.**

- **Functions with no arguments and return values.**

- **Functions that return multiple values.**

FUNCTIONS

User Defined Functions

- **Functions with no arguments and no return value (void functions without parameters)**

  - **Function without any arguments means no data is passed (values like int, char, etc..) to the called function.**

  - **Similarly, function with no return type does not pass back data to the calling function.**

  - **This type of function which does not return any value cannot be used in an expression.**

  - **It can be used only as independent statement.**

FUNCTIONS

User Defined Functions

- **Functions with no arguments and no return value (void functions without parameters) — Example**

```c
#include<stdio.h>
#include<conio.h>
void printline();
void main()
{
  clrscr();
  printf("Welcome to function in C");
  printline();
  printf("Function easy to learn.");
  printline();
  getch();
}
```

```c
void printline()
{
    int i;
    printf("\n");
    for(i=0;i<30;i++)
    {
        printf("-");
    }
    printf("\n");
}
```

FUNCTIONS

User Defined Functions

- **Functions with arguments and no return value (void functions with parameters)**

  – **Function with arguments can perform much better than a function without arguments.**

  – **This type of function can accept data from calling function.**

  – **Data can be sent to the called function from calling function but you cannot send result data back to the calling function.**

  – **Output of function can be controlled by providing various values as arguments.**

FUNCTIONS

User Defined Functions

- **Functions with arguments and no return value (void functions with parameters) — Example**

```c
#include<stdio.h>
#include<conio.h>
void add(int , int );
void main()
{
  clrscr();
  add(30,15);
  add(63,49);
  add(952,321);
  getch();
}
```

```c
void add(int x, int y)
{
  int result;

  result = x+y;
  printf("Sum of %d and %d is %d.\n\n",x,y,result);
}
```

FUNCTIONS

User Defined Functions

- **Functions with no arguments but returns value**

  (non-void functions without parameters)

  - **Function does not take any argument but only returns values to the calling function**

```c#include<stdio.h>#include<conio.h>int send(){ int no1; printf("Enter a no : "); scanf("%d",&no1); return(no1);}```	```cvoid main(){ int z; clrscr(); z = send(); printf("\nYou entered : %d.", z); getch();}```

FUNCTIONS

User Defined Functions

- **Functions with arguments and return value (non-void functions with parameters)**

  - **This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function.**

  - **The data returned by the function can be used later in the program for further calculations.**

FUNCTIONS

User Defined Functions

- **Functions with arguments and return value (non-void functions with parameters) — Example**

```
#include<stdio.h>
#include<conio.h>
int add(int , int );
void main()
{
    int z;
    clrscr();
    z = add(952,321);
    printf("Result %d.\n\n",add(30,55));
    printf("Result %d.\n\n",z);
    getch();
}
```

```
int add(int x, int y)
{
    int result;
    result = x+y;
    return(result);
}
```

FUNCTIONS

User Defined Functions

- **Functions that return multiple values — Example**

#include<stdio.h>	**void main()**
#include<conio.h>	**{**
void calc(int x, int y, int *add, int *sub)	**int a=20, b=11, p,q;**
{	**clrscr();**
** *add = x+y;**	**calc(a,b,&p,&q);**
** *sub = x-y;**	**printf("Sum =%d,**
}	**Sub=%d", p, q);**
	getch();
	}

FUNCTIONS

Inter-function Communication

- **Functions have to communicate between them to exchange data**

- **The data flow between the calling and called functions can be divided into**

  - **A downward flow – from the calling to the called function**

  - **An upward flow from the called to the calling function**

  - **A bi-directional flow in both directions**
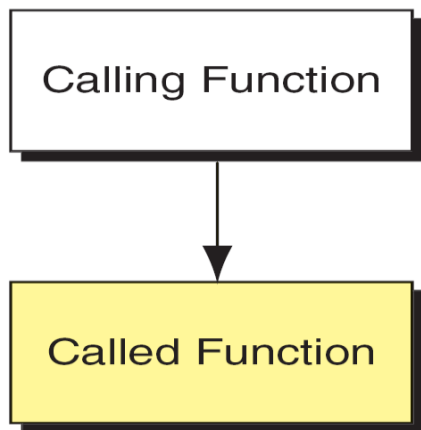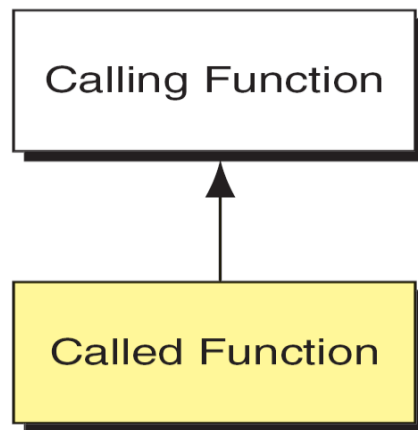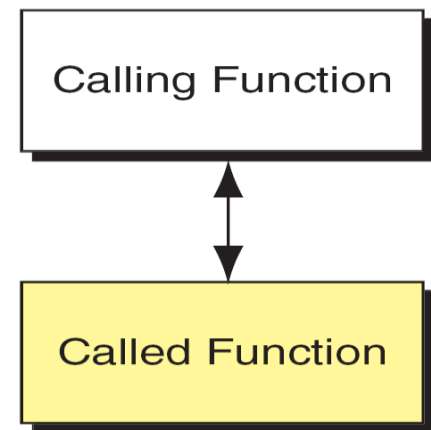
a. Downward b. Upward c. Bi-direction

FUNCTIONS

Inter-function Communication

– **Downward flow**

- **In downward communication, the calling function sends data to the called function**

- **No data flows in the opposite direction**

- **Copies of data items are passed from the calling function to the called function.**

- **The called function may change the values passed, but the original values in the calling function remain untouched**

FUNCTIONS

Inter-function Communication

Fig: Downward Communication

FUNCTIONS

Inter-function Communication

- **Upward flow**

  - **Upward communication occurs when the called function sends data back to the calling function without receiving any data from it**

FUNCTIONS

Inter-function Communication
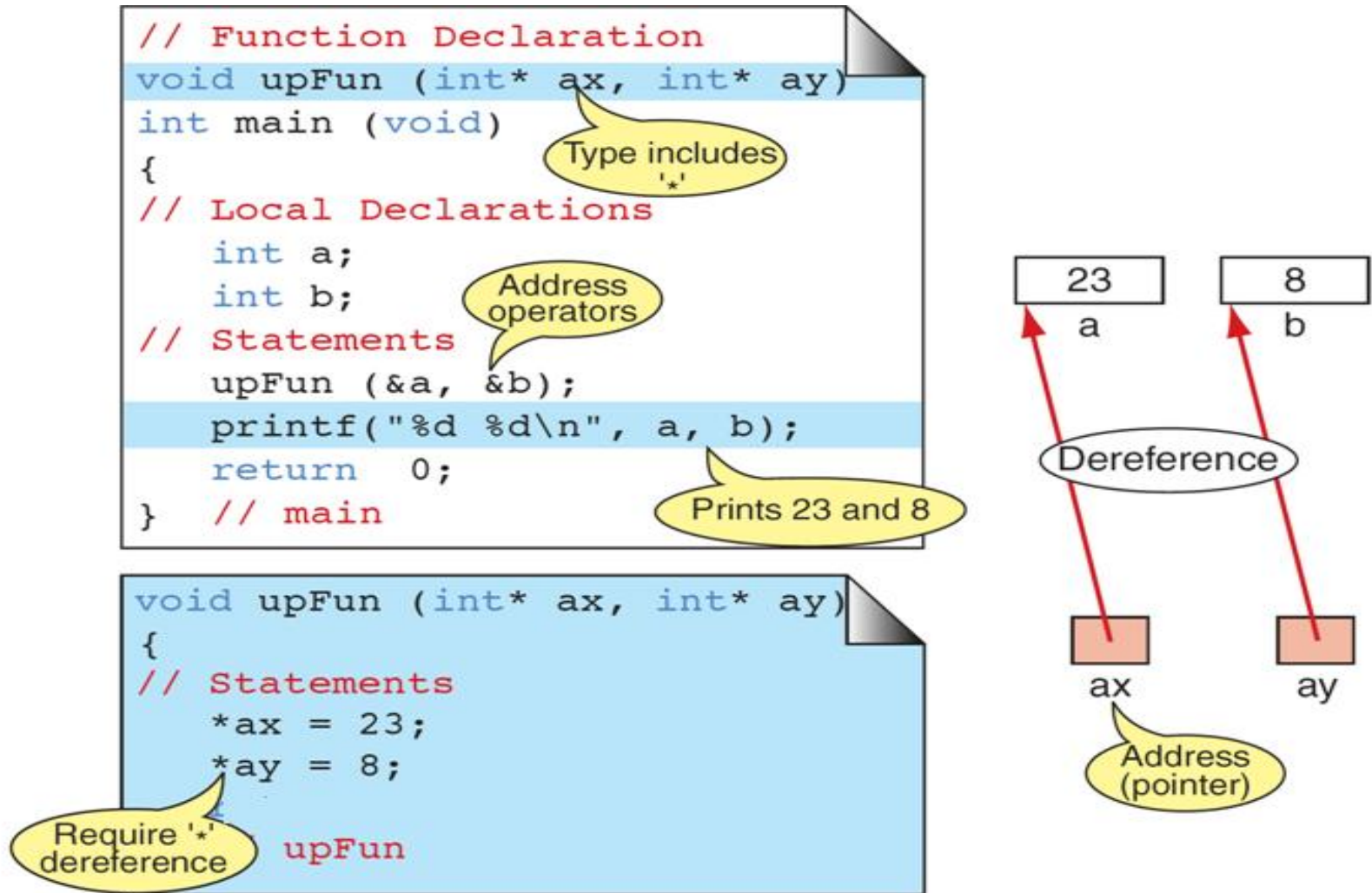
Fig: Upward Communication

FUNCTIONS

Inter-function Communication

– **Bi-directional flow**

- **Bi-directional communication occurs when the calling function sends data down to the called function.**

- **During or at the end of its processing, the called function then sends data up to the calling function**

FUNCTIONS

Inter-function Communication


```
// Function Declaration
void biFun (int* ax, int* ay);

int main (void)
{
// Local Definitions
    int   a = 2;
    int   b = 6;

// Statements

    ...
    biFun (&a, &b);

    ...
 return 0;
}   // main
```

```
void biFun (int* ax, int* ay)
{
    *ax = *ax + 2;
    *ay = *ay / *ax;

}   // biFun
```
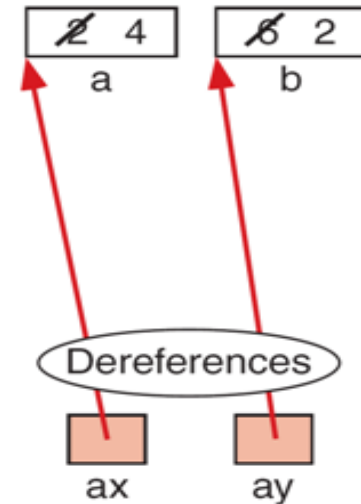
Fig: Bi-directional Communication

FUNCTIONS

Standard Functions

- C provides a large set of functions whose definitions have been written and are ready to be used in user programs.

- To use these functions, we include *(#include<math.h> or #include<stdlib.h>)* in programs.

- The function declarations of these functions are grouped together and collected in several header files.

FUNCTIONS

Standard Functions

- – **The header files are included in the program instead of adding the individual function declarations.**

- – **The *include* statement causes the header file of the function to be copied into the program.**

- – **When the program is linked, the object code for the function is combined with the program code to build the complete program.**

FUNCTIONS

Standard Functions
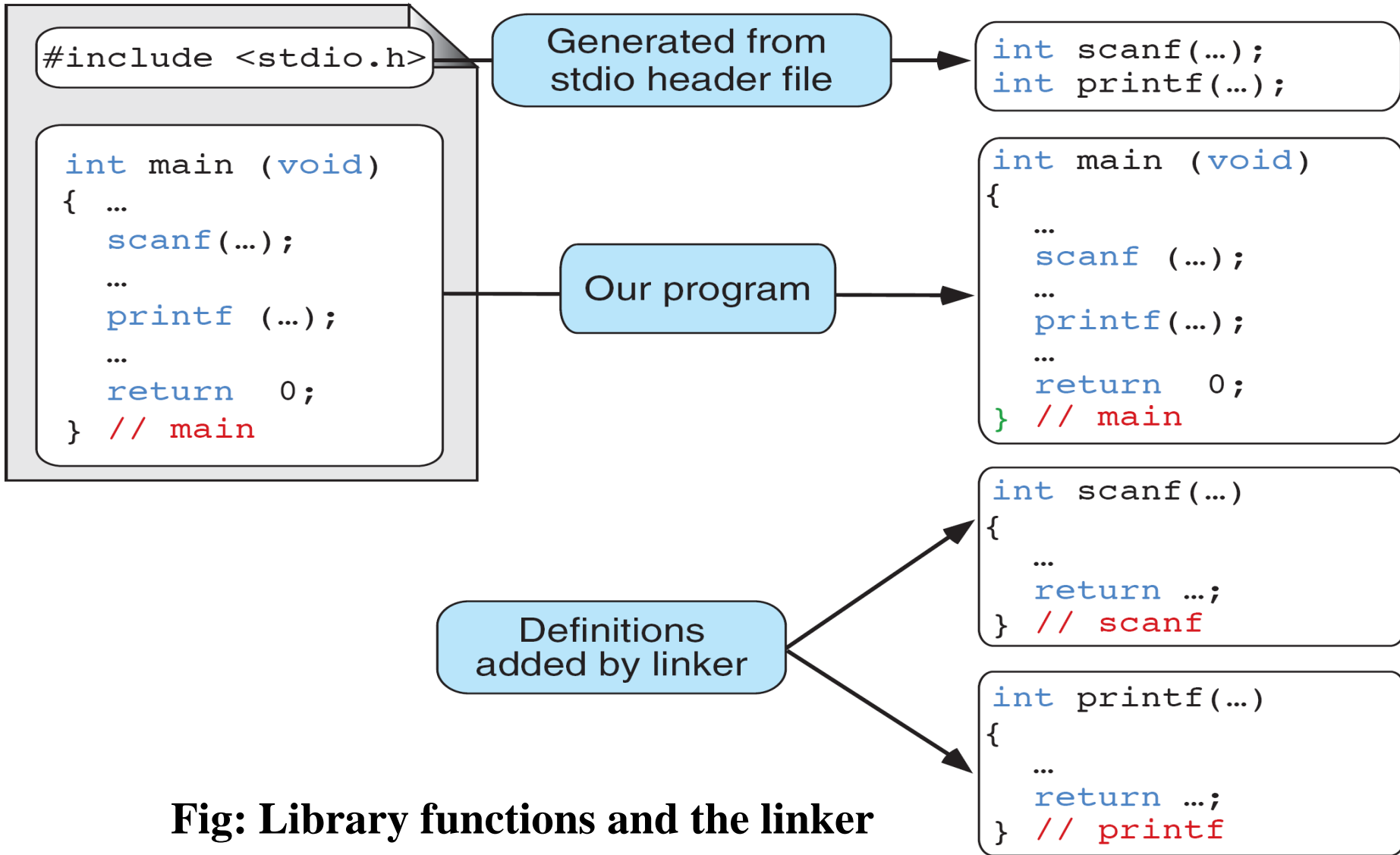
Fig: Library functions and the linker

FUNCTIONS

Standard Functions

– **Math Functions**

- **These are collections of functions for mathematical calculations. These include**

  – **Absolute value functions – abs – returns the positive value regardless of sign**

  int abs (int number);

  abs (3)

  – **Complex Number functions – cabs, creal, cimag**

  – **Ceiling functions – ceil -returns smallest integral value greater than or equal to a number**

  float ceil (float number)

  ceil (-1.9) returns -1.0

  ceil (1.1) returns 2.0

FUNCTIONS

Standard Functions

- **Math Functions**
  - **Floor functions – floor – returns the largest integral value that is equal to or less than a number**

    float floor (float number)

    floor (-1.1) returns -2.0

    floor (1.9) returns 1.0

  - **Truncate functions – trunc – returns the integral in the direction of 0**

    double trunc (double number)

    trunc (-1.1) returns -1.0

    trunc (1.9) returns 1.0

FUNCTIONS

Standard Functions

– **Math Functions**

  – **Floor functions – floor – returns the largest integral value that is equal to or less than a number**

  > **float floor (float number);**
  >
  > **floor (-1.1) returns -2.0**
  >
  > **floor (1.9) returns 1.0**

  – **Truncate functions – trunc – returns the integral in the direction of 0**

  > **double trunc (double number);**
  >
  > **trunc (-1.1) returns -1.0**
  >
  > **trunc (1.9) returns 1.0**

FUNCTIONS

Standard Functions

– **Math Functions**

– **Round functions – round – returns the nearest integral value**

> **double round (double number);**
>
> **round (-1.1) returns -1.0**
>
> **round (1.9) returns 2.0**
>
> **round (-1.5) returns -2.0**

– **Power functions – pow – returns the value of the x raised to the power y. Error occurs if the base(x) is negative and the exponent (y) is not an integer or if the base is zero and the exponent is not positive**

> **double power (double n1, double n2);**
>
> **pow (3.0, 4.0) returns 81.0**

FUNCTIONS

Standard Functions

– **Math Functions**

   – **Square root functions – sqrt – returns the non-negative square root of a number. Error occurs if the number is negative.**

   double sqrt (double n1);

   sqrt (25) returns 5.0

FUNCTIONS

Standard Functions

- **Random numbers**
    - **A random number is a number selected from a set in which all members have the same probability of being selected.**
    - **C provides two functions to build a random number series seed random (srand) and random (rand).**
    - **These functions are found in *stdlib.h***

        srand (997)

FUNCTIONS

Standard Functions

```
MATH.H

Functions

abs             acos,   acosl       asin,     asinl
atan,   atanl   atan2,  atan2l      atof,     _atold
cabs,   cabsl   ceil,   ceill       cos,      cosl
cosh,   coshl   exp,    expl        fabs,     fabsl
floor,  floorl  fmod,   fmodl       frexp,    frexpl
hypot,  hypotl  labs                ldexp,    ldexpl
log,    logl    log10,  log10l      matherr,  _matherrl
modf,   modfl   poly,   polyl       pow,      powl
pow10,  pow10l  sin,    sinl        sinh,     sinhl
sqrt,   sqrtl   tan,    tanl        tanh,     tanhl
```

STORAGE CLASSES

- *Scope*

    – Scope determines the region of the program in which a defined object is visible in the part of the program in which we can use the object's name.

    – Scope pertains to any object that can be declared like a variable or a function declaration.

    – A block is zero or more statements enclosed in a set of braces.

    – A block has a declarations section and a statement section.

266

STORAGE CLASSES

- *Scope*

  - Blocks can be nested within the body of a function and each block will be an independent group of statements with its own isolated definitions.

  - Global area of a program consists of all statements that are outside functions.

  - An objects scope extends from its declaration until the end of the its block.

  - Variables are in scope from their point of declaration until the end of their block.

STORAGE CLASSES

- *Scope* — **Example**

```
#include<stdio.h>
int fun (int , int );          //Global area
int p;                          //Global variable
int main (void)
{
    int  x;                     //main's area
    float y;                    //Local variables
        { // beginning of nested block
                float a = y /2;
                float y;        //Nested block area
                float z;

                 ….
                z =  a * b;

                 ….
        } // end of nested block
}
```

```
int fun ( int i, int j)
{
        int a;
        int y;

                //function area
        …
}  // function block
```

STORAGE CLASSES

- *Scope*

  – **Global Scope**

    - **Any object defined in the global are of a program is visible from its definition until the end of the program.**

    - **The function declaration for *fun* is a global definition.**

    - **It is visible everywhere in the program.**

STORAGE CLASSES

- *Scope*
  - **Local Scope**

  - **Variables defined within a block have local scope.**

  - **They exist only from the point of their declaration until the end of the block in which they are declared**

  - **Outside the block they are invisible**
    - **There are two blocks in main.**

    - **The first block is all of main.**

    - **The second block is nested within main.**

    - **All definitions in main are visible to the second block unless local variables with an identical name are defined.**

    - **In the inner block a local version of '*a*' is defined and its type is float.**

STORAGE CLASSES

- *Storage Class*

    – The storage class of a variable defines the scope (visibility) and life time of variables and/or functions declared within a C Program.

    – To fully define a variable it is necessary to define its 'type' and also its 'storage class'.

    – If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes.

STORAGE CLASSES

- *Storage Class*

    - A variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored.

    - There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers.

    - It is the variable's storage class that determines in which of these two locations the value is stored.

STORAGE CLASSES

- **Object Storage Attributes**

   Storage class specifiers control three attributes of an object's storage.

   – **Scope**
   – **Extent**
   – **Linkage**

STORAGE CLASSES

- **Object Storage Attributes**
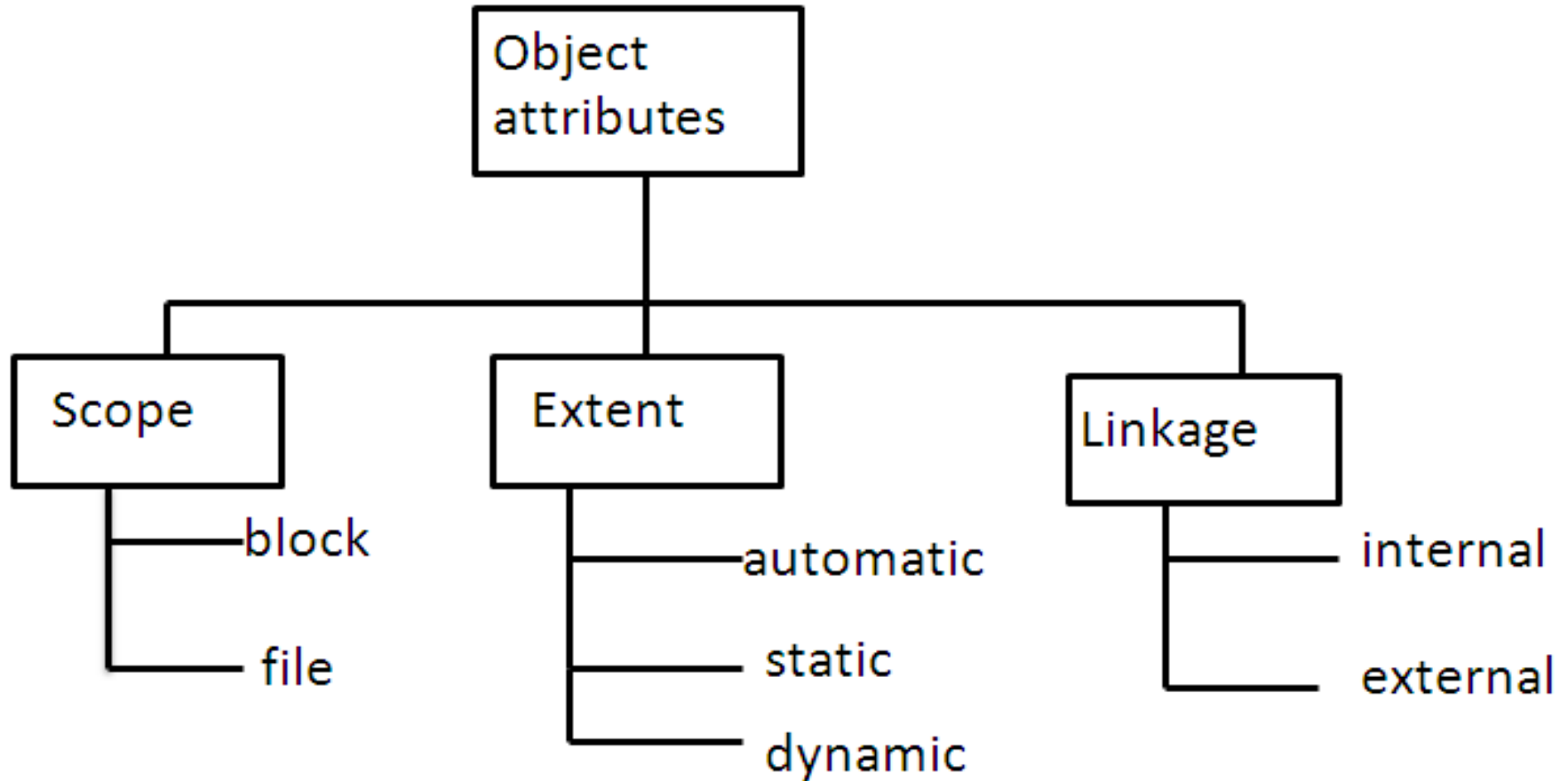
STORAGE CLASSES

- **Object Storage Attributes**
  - **Scope**
    - **Scope defines the visibility of an object.**
    - **It defines where an object can be referenced.**
    - **Scope can be**
      - **Block Scope**
      - **Global(File) Scope.**

STORAGE CLASSES

- **Object Storage Attributes**

  – **Scope**

    – **Block (Local) Scope**

      » **When the scope of an object is block, it is visible only in the block in which it is defined. This object is called a local object.**

      » **Example – A variable declared in the formal parameter list of a function has block scope.**

      » **A variable declared in the initialization section of a *for* loop also has a block scope, but only within the *for* statement.**

STORAGE CLASSES

- **Object Storage Attributes**

    – **Scope**

    – **File (Global) Scope**

        » **File scope includes the entire source file for a program, including any files included in it.**

        » **An object with file scope has visibility through the whole source file in which it is declared.**

        » **Objects within block scope are excluded from file scope unless specifically declared to have file scope i.e. block scope hides objects from file scope.**

        » **File scope includes all declarations outside a function and all function headers.**

        » **An object with file scope is referred as Global object.**

STORAGE CLASSES

- **Object Storage Attributes**

  - **Extent**

    - **The extent of an object defines the duration for which the computer allocated memory for it.**

    - **Extent can be**
      - **Automatic extent**
      - **Static Extent**
      - **Dynamic Extent**

STORAGE CLASSES

- **Object Storage Attributes**

  – **Extent**

    – **Automatic extent**

      » **An object with automatic extent is created each time its declaration is encountered and is destroyed each time its block is exited.**

      » **Example– a variable declared in the body of a loop is created and destroyed in each iteration.**

      » **Declarations in a function are not destroyed until the function is complete.**

      » **When a function calls a function, they are out of scope but not destroyed.**

STORAGE CLASSES

- **Object Storage Attributes**
  - **Extent**
    - **Static extent**
      - » **A variable with a static extent is created when the program is loaded for execution and is destroyed when the execution stops.**
    - **Dynamic extent**
      - » **Dynamic extent is created by the program through *malloc()* library functions.**

STORAGE CLASSES

- **Object Storage Attributes**

    – **Linkage**

    - **When a program is divided into modules, these module are to be linked for the whole program to function.**

    - **Linkage can be**

        – **Internal**

        – **External**

STORAGE CLASSES

- **Object Storage Attributes**

  - **Linkage**

    - **Internal**

      » **An object with internal linkage is declared and visible only in one module. Other modules cannot refer to this object.**

    - **External**

      » **An object with an external linkage is declared in one module but is visible in all other modules that declare it with a special keyword, *extern*.**

STORAGE CLASSES

- **Types of Storage Classes**

    There are four storage classes in C:

    - **Automatic** storage class

    - **Register** storage class

    - **Static** storage class

    - **External** storage class

STORAGE CLASSES

FEATURE	STORAGE CLASS			
	Auto	**Extern**	**Register**	**Static**
Accessibility	**Accessible within the function or block in which it is declared**	**Accessible within all program files that are a part of the program**	**Accessible within the function or block in which it is declared**	**_Local:_ Accessible within the function or block in which it is declared _Global:_ Accessible within the program in which it is declared**
Storage	**Main Memory**	**Main Memory**	**CPU Register**	**Main Memory**

Table: Storage Classes Features

STORAGE CLASSES

FEATURE	STORAGE CLASS			
	Auto	Extern	Register	Static
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Exists throughout the execution of the program	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	*Local:* Retains value between function calls or block entries *Global:* Preserves value in program files
Default value	Garbage	Zero	Garbage	Zero

Table: Storage Classes Features

STORAGE CLASSES

- *Automatic Storage Class*

Keyword	**-- auto**
Storage	− **Memory.**
Default value	− **An unpredictable value, which is often called a garbage value.**
Scope	− **Local to the block in which the variable is defined.**
Life	− **Till the control remains within the block in which the variable is defined.**

STORAGE CLASSES

- *Automatic Storage Class*

Example:

```
#include<stdio.h>
void call1();
void call2();
main()
{
    int v=10;
    call1();
    call2();
    printf("\n v=%d",v);
}

void call1()
{
    int v=20;
    printf("\n v=%d",v);
}
void call2()
{
    int v=30;
    printf("\n v=%d",v);
}
```

STORAGE CLASSES

- *Register storage class*

The features of a variable defined to have an automatic storage class , declaration includes a recommendation to the compiler to use a CPU register for the variable:

        Keyword   -- register

        Storage   − Register /   Memory.

        Default initial value − Garbage value.

        Scope   − Local to the block in which the variable is defined.

        Life   − Till the control remains within the block in which the variable is defined.

STORAGE CLASSES

- *Register storage class*

    Example:

    #include<stdio.h>

    main()

    {

    register int m=1;

    for(;m<=5;m++)

    printf("%d",m);

    }

STORAGE CLASSES

- *Static storage class (Block scope)*
  The features of a variable defined to have static storage class are as under:

Key word	**-- static**
Storage	**− Memory.**
Default value	**-- only one time initialization ,**
	if not initialized it will be zero .
Scope	**− Local to the block in which the variable is defined.**
Life	**− is created when the program is loaded for execution and is destroyed when the execution stops.**

STORAGE CLASSES

- *Static storage class (Block scope)*

Example:
```
#include<stdio.h>
void call1();
void call2();
main()
{
   call1();
   call2();
   call1();
   call2();
}
```

```
void call1()
{
  static int v;
   v=v+10;
    printf("\n in
           call1()v=%d",v);
}
void call2()
{
   static int v;
   v=v+15;
   printf("\n in call2()
          v=%d",v);
}
```

STORAGE CLASSES

- *Static storage class (File scope)*
  The features of a variable defined to have static storage class are as under:

Key word	**-- static**
Storage	**− Memory.**
Default value	**-- only one time initialization ,**
	if not initialized it will be zero .
Scope	**− visible to the whole source file in which the variable is defined.**
Life	**− is created when the program is loaded for execution and is destroyed when the execution stops.**

STORAGE CLASSES

- *Static storage class (File scope)*

Example:

```
#include<stdio.h>
void call1();
void call2();
 static int v;
main()
{
   call1();
   call2();
   call1();
   call2();
}
```

```
void call1()
{
  v=v+10;
  printf("\n in call1()v=%d",v);
}
void call2()
{
  static int v;
  v=v+15;
  printf("\n in call2() =%d",v);
}
```

STORAGE CLASSES

- **External storage class**
  The features of a variable defined to have extern storage class are as under:

  Key word   --   **extern**

  Storage   −   **Memory**

  Default value   --   **initialized with zero**

  Scope   −   **visible to the whole source file in which the variable is defined**

  Life   −   **is created when the program is loaded for execution and is destroyed when the execution stops**

STORAGE CLASSES

- **External storage class**

Example:

```
void  call1();                          void call2()
void  call2();                          {
int v=10;                                  extern int v;
main()                                     printf("in call2() =%d",v);
{                                       }

   call1();
   call2();
   printf("in main() =%d",v);
}
void call1()
{

   int v=20;
   printf("in call1() =%d",v);
   }
```

RECURSION

- A **recursive function** is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

- Every recursive solution has two major cases, they are:

    – Base Case

    – Recursive Case

Base case:

- *T*he problem is simple enough to be solved directly without making any further calls to the same function.

RECURSION

Recursive case:

- first, the problem is divided into simpler sub parts.

- Second, the function calls itself but with sub parts of the problem obtained in the first step.

- Third, the result is obtained by combining the solutions of simpler sub-parts.

- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem.

- In recursive function, complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

RECURSION

Factorial of A Number Using Recursion

PROBLEM	SOLUTION
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

- **Base case is when n=1, because if n = 1, the result is known to be 1**

- **Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as**

$$\text{factorial(n)} = n \times \text{factorial (n-1)}$$
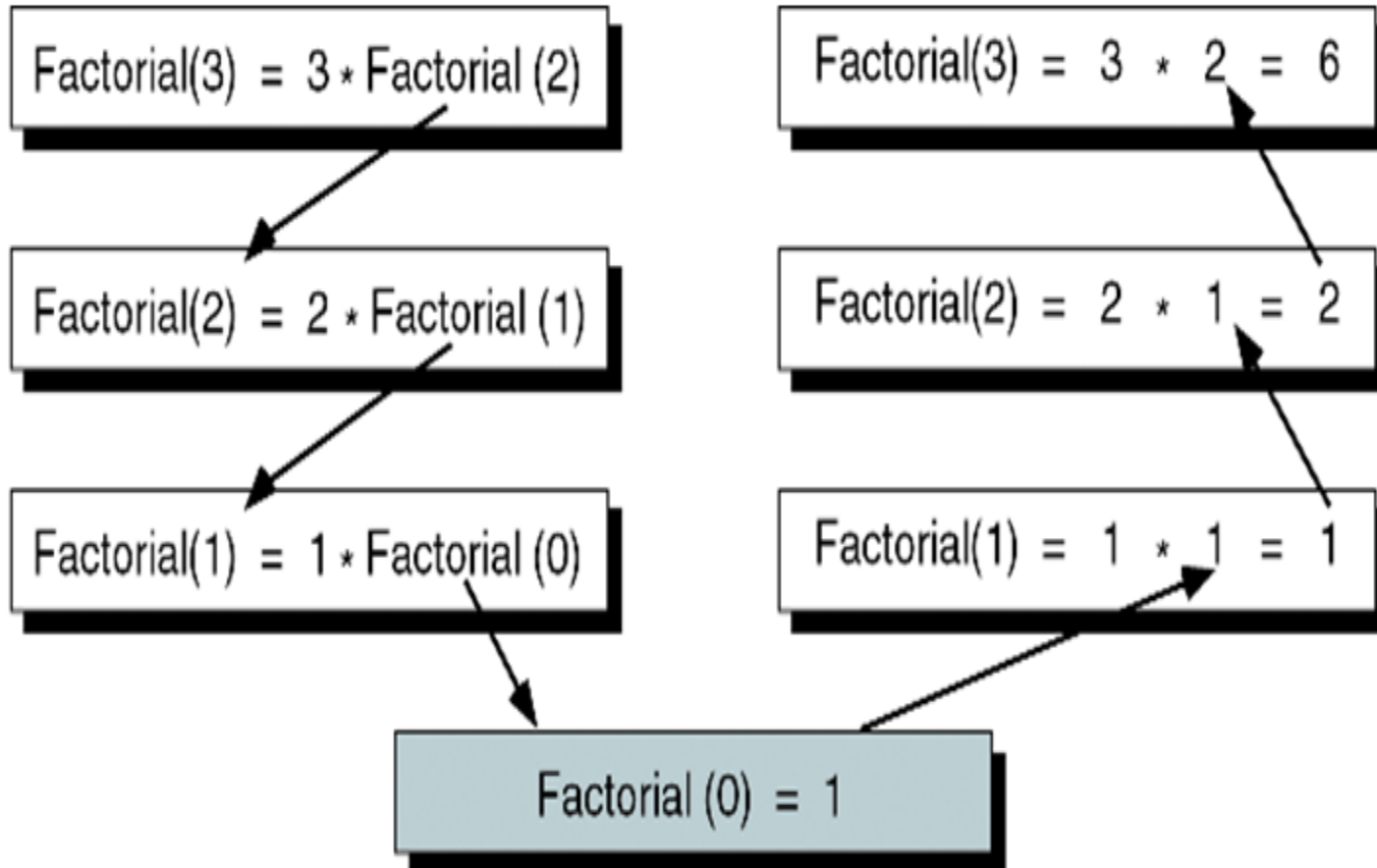
RECURSION

Factorial of A Number Using Recursion — **Example**

```c
#include<stdio.h>
int Fact(int)
{if(n==1)
    retrun 1;
 return (n * Fact(n-1));
}
main()
{int num;
 scanf("%d", &num);
 printf("\n Factorial of %d = %d", num, Fact(num));
 return 0;
}
```

RECURSION

Factorial of A Number Using Recursion — **Illustration**

Factorial(3) = 3 * Factorial (2)

Factorial(2) = 2 * Factorial (1)

Factorial(1) = 1 * Factorial (0)

Factorial (0) = 1

Factorial(3) = 3 * 2 = 6

Factorial(2) = 2 * 1 = 2

Factorial(1) = 1 * 1 = 1

RECURSION

Fibonacci Series Using Recursion

- **The Fibonacci series can be given as:**

  0 1 1 2   3   5   8   13   21   34   55……

- **The third term of the series is the sum of the first and second terms.**

- **On similar grounds, fourth term is the sum of second and third terms, so on and so forth.**

- **Now we will design a recursive solution to find the nth term of the Fibonacci series. The general formula to do so can be given as**

$$
\text{FIB}(n) =
\begin{cases}
1, & \text{if } n \le 2 \\
\text{FIB}(n-1) + \text{FIB}(n-2), & \text{otherwise}
\end{cases}
$$

RECURSION

*Fibonacci Series Using Recursion*

RECURSION

Fibonacci Series Using Recursion — Program

```
int Fibonacci(int num)
{   if(num <= 2)
        return 1;
    return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
main()
{
    int n;
    printf("\n Enter the number of terms in the series : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    printf("\n Fibonacci (%d) = %d", i, Fibonacci(i));
}
```

RECURSION

Types Of Recursion

- **Any recursive function can be characterized based on:**
  - **whether the function calls itself directly or indirectly (direct or indirect).**

  - **whether any operation is pending at each recursive call (tail-recursive or not).**

  - **the structure of the calling pattern (linear or tree-recursive).**

```
                    Recursion
    ┌──────────┬──────────┼──────────┬──────────┐
  Direct    Indirect    Linear      Tree       Tail
```
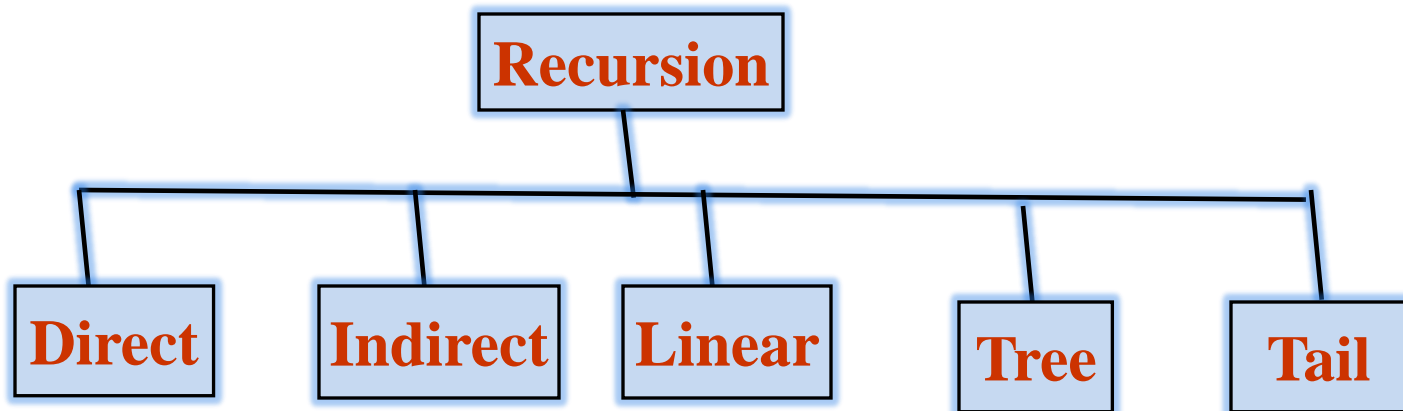
RECURSION

Types of Recursion

– **Direct Recursion**

- **A function is said to be *directly* recursive if it explicitly calls itself.**

- **Example**

```
int Func( int n)

{

  if(n==0)

  retrun n;

  return (Func(n-1));

}
```

RECURSION

Types of Recursion

– **Indirect Recursion**

- **A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.**

- **Example**

int Func1(int n)	int Func2(int x)
{ if(n==0) return n; return Func2(n); }	{ return Func1(x-1); }

- **These two functions are indirectly recursive as they both call each other.**

RECURSION

- *Types of Recursion*
- **Tail Recursion**

  - **A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.**

  - **That is, when the called function returns, the returned value is immediately returned from the calling function.**

  - **Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.**

RECURSION

- *Types of Recursion*
- Tail Recursion
  - **Example**

int Fact(n)
{
  return Fact1(n, 1);
}
	int Fact1(int n, int res)
	{
	      if (n==1)
	       return res;
	      return Fact1(n-1, n*res);
	}

RECURSION

- *Types of Recursion*
- Linear Recursion
  - A recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function.

  - For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.

```
int Fact(int)
{     if(n==1)
          retrun 1;
      return (n * Fact(n-1));
}
```

RECURSION

- *Types of Recursion*
- – **Tree Recursion**
    - **A recursive function is said to be *tree recursive* (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.**

    - **For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.**

```
int Fibonacci(int num)
{
    if(num <= 2)
            return 1;
    return ( Fibonacci (num - 1) +
    Fibonacci(num – 2));
}
```

RECURSION

Pros and Cons of Recursion

Pros:

- **Recursive solutions often tend to be shorter and simpler than non-recursive ones.**

- **Code is clearer and easier to use**

- **Recursion represents like the original formula to solve a problem.**

- **Follows a divide and conquer technique to solve problems**

- **In some (limited) instances, recursion may be more efficient**

RECURSION

Pros and Cons of Recursion

Cons:

- **Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.**

- **Aborting a recursive process in midstream is slow and sometimes nasty.**

- **Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.**

- **It is difficult to find bugs, particularly when using global variables**

RECURSION

Limitations Of Recursion

- **Recursive solutions may involve extensive overhead because they use function calls.**

- **Each function call requires push of return memory address, parameters, returned results, etc. and every function return requires that many pops.**

- **Each time we make a call we use up some of our memory allocation. If the recursion is deep that is, if there are many recursive calls then we may run out of memory.**
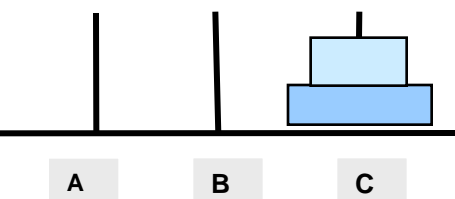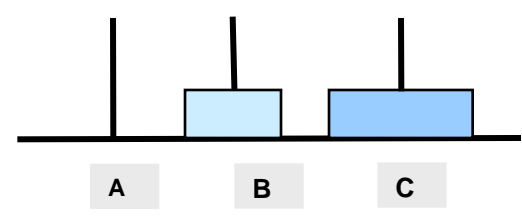
RECURSION

Towers Of Hanoi

- **Tower of Hanoi is one of the main applications of a recursion. It says, "if you can** solve ***n-1* cases, then you can easily solve the *nth* case?"**
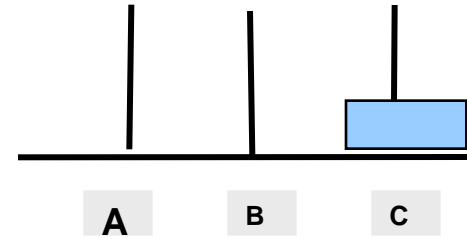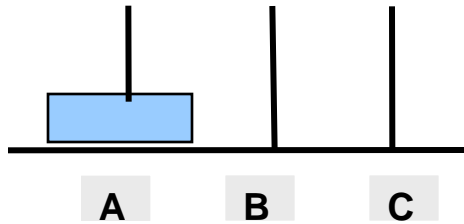
If there is only one ring, then move the ring from source to the Destination
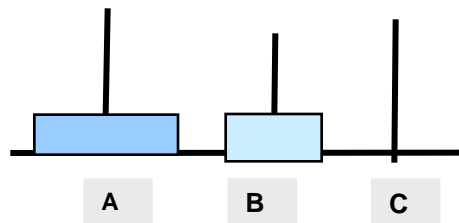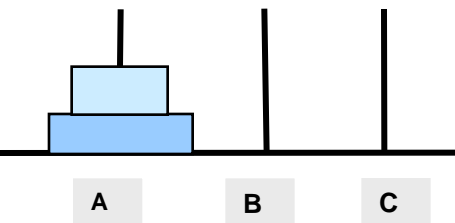
If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from the source to the destination

RECURSION

Towers Of Hanoi

- **Consider the working with three rings.**
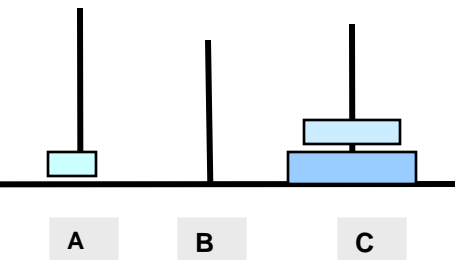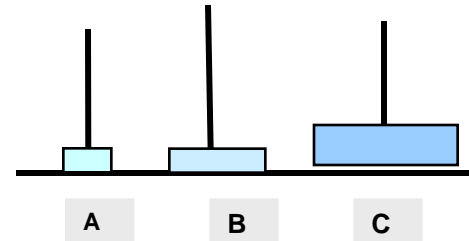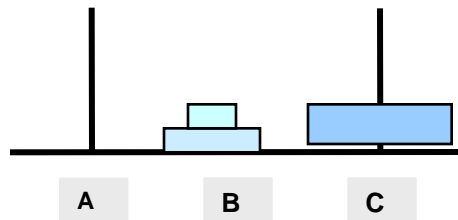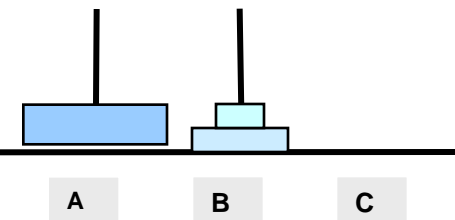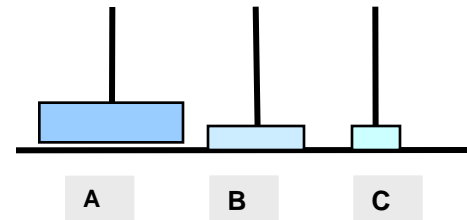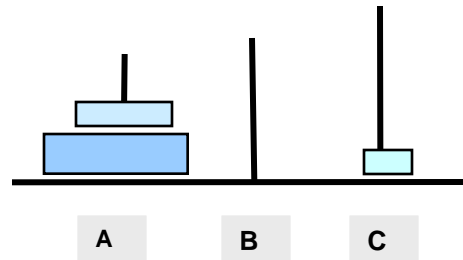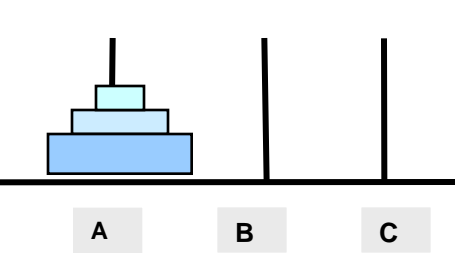
PRE-PROCESSOR

- There are many steps involved in turning a C program into an executable program. The first step is called *pre-processing*.
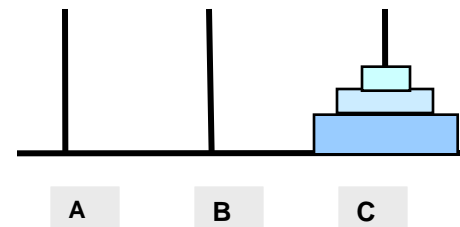
- The pre-processor performs textual manipulation on the source code before it is compiled. There are a number of major parts to this:

  1. Deleting comments

  2. Inserting the contents of files mentioned in #include directives

  3. Defining and substituting symbols from #define directives

  4. Deciding which code should be compiled depending on conditional compiler directives

  5. To act on recognized #pragma statements, which are implementation dependent.

PRE-PROCESSOR

Predefined Symbols

_ _DATE_ _ provides a string constant in the form "mm dd yyyy"

_ _FILE_ _ provides a string constant containing the name of

the source file.

_ _LINE_ _ provides a string constant containing the current

statement number in the source file.

_ _TIME_ _ provides a string constant in the form hh:mm:ss

_ _STDC_ _ provides a integer constant with value 1 if and only

if the   compiler confirms with ISO implementation.

PRE-PROCESSOR

Macro Substitution

- **Definition:**

  #define *name replacement text*

  This causes a simple macro substitution, each occurrence of *name* **is replaced by** *replacement text***.**

- **Example:**

  #define XDIM 10

  causes all occurrences of XDIM to be replaced by the literal 10.

PRE-PROCESSOR

Macro Substitution

- **The replacement text spreads over more than one line a forward slash(\) is used to indicate continuation.**

- **Example:**

    **#define WHERE_AM_I printf("In file %s at line %d", **

    __FILE__,__LINE__)

PRE-PROCESSOR

Macro Substitution

- **Example**

    #define TWO_PLUS_J   2+j

  Such substitutions should be undertaken with care as there may be more than one j in scope.

  If the following were to appear in the program:

    int j=100;

    i= TWO_PLUS_J;

  After pre-processing, it would be:

    int j=100;

    i= 2+j;

PRE-PROCESSOR

Macro Substitution

• **Example**

    #define TWO_PLUS_J   2+j

Such substitutions should be undertaken with care as there may be more than one j in scope.

The programmer must realize that the replacement text is used exactly.

        5* TWO_PLUS_J

will after pre-processing be:

        5* 2+j

which might not be what the programmer expected?

PRE-PROCESSOR

Macros

- **A macro allows parameters to be used in substitution text.**

- **Syntax**

  #define   name(parameters)   code

  The bracket of the parameter list must be next to the *name*, otherwise it will be part of the code substituted.

- **Example**

  #define   SQUARE(x) x*x

  x= SQUARE(5); →   x= 5*5;

  y= SQUARE(x); →   y= x*x;

  z= SQUARE(2.5);→   z= 2.5*2.5;

  10+SQUARE(5); →   10+5*5

PRE-PROCESSOR

Conditional Compilation

- **Conditional compilation provides a way where by code can be selectively included into the compilation - depending on values available at pre-processing.**

- *Syntax:*

    #if constant expression

       …..

       statements

       ......

    #endif

PRE-PROCESSOR

Conditional Compilation:

— **The pre-processor evaluates the *constant expression* if it is zero (false) the *statements* are deleted from the code passed to the compiler, if the *constant expression* is non-zero (true) the statements are passed to the compiler.**

— **The *constant expression* is made up of literals and variables that have been defined using a #define.**

— **It is illegal to use anything where the value will not be known until execution time, as the compiler is unable to predict the values.**

PRE-PROCESSOR

Conditional Compilation

- A simple example is to bracket the code used for debugging in the following manner:

```
#if DEBUG
    printf("At line %d: a=%d, b=%d\n",__LINE__,a,b);
#endif
```

- Therefore, the listing may contain many instances of this conditional inclusion, protecting the printing of interesting variables.

PRE-PROCESSOR

Conditional Compilation

— **If the following is present:**

    #define DEBUG 1

then at compile time, all the debugging print statements will be included in the object code produced.

— **Whereas if the definition is:**

    #define DEBUG 0

none of the debug statements will be included in the object code generated.

PRE-PROCESSOR

Conditional Compilation

- **Conditional compilation is also useful if developing a software product that has different functionality depending on whether the user has purchased the full version, the economy version or is trying a cover disc sample.**

- **A single set of code can exist for all versions. Where there is functionality, that is available, differs between versions then the code can be delimited within a conditional inclusion.**

- **This can be achieved using this enumeration and definition.**

PRE-PROCESSOR

Conditional Compilation

- **Example**

    enum VERSION {FULL, ECONOMY, SAMPLE};

    #define VERSION   FULL

in conjunction with conditional compilations of the following sort:

#if (VERSION == FULL)   *statements for full implementation*

#elif(VERSION == ECONOMY) *statements for economy implementation*

#else   *statements for sample implementation*

#endif

PRE-PROCESSOR

Conditional Definitions

- **The #ifdef command conditionally includes code if a symbol is defined.**

- **If the symbol is not defined, the code is not included.**

- **The opposite command is #ifndef which includes code only if the symbol is not defined.**

- **For example, if the program includes a library file that, in some implementations, does not define a symbol.**

PRE-PROCESSOR

Conditional Definitions

• **MAXLINES, then the program may have a fragment like this:**

> **1: #include <somelib.h>**
>
> **2: #ifndef MAXLINES**
>
> **3: #define MAXLINES 100**
>
> **4: #endif**

– **Line 1 includes the library, which may vary between machines.**

– **Line 2 checks if MAXLINES is already defined. If it isn't defined then Line 3 defines it.**

– **Line 4 is the end of the conditional definition.**

PRE-PROCESSOR

Conditional Definitions

- **This is necessary as it is not possible to define the same symbol twice.**

- **An alternative would be to undefined the symbol and then redefine it, for example:**

> **#undef MAXCHARS**
>
> **#define MAXCHARS 60**

PRE-PROCESSOR

Pragma

- The #pragma command is a mechanism that supports implementation dependent directives.

- An environment may provide pragma to allow special options. Where a pragma is not recognized, it is ignored.

- Program with pragma will run on different machines.

- The actions of the pragma may not be the same across machines, so the program is not truly portable.

- Pragma is used in a number of ways by C++ Builder.

PRE-PROCESSOR

Pragma

- **Example:**

  when generating forms, the corresponding unit will contain code like:

  #pragma resource "*.dfm"

  This is equivalent to: {$R *.DFM}

  Error command

  The error command is of the form   # error message

  It is used to print the message detected by the preprocessor.

Pointer

- **A pointer is a constant or variable that contains an address that can be used to access data.**

    (or)

    A pointer is a variable that contains the memory location of another variable.

- **Pointers deals with memory address, it can be used to access and   manipulate data stored in memory.**

'*' And '&' Operators

- **When is * used?**
  *→"dereferencing operator" which provides the contents in the memory location specified by a pointer.

- **when is & used?**

  &→"address operator" which gives or produces the memory address of a data variable.

- **verify the following declaration**
  int x=10;
- **In the above statement the c compiler reserve the memory space for the integer value**

Use of & and *

x ——————— Memory location name.

10 ——————— Value at memory location.

65325 ——————— Memory location address.
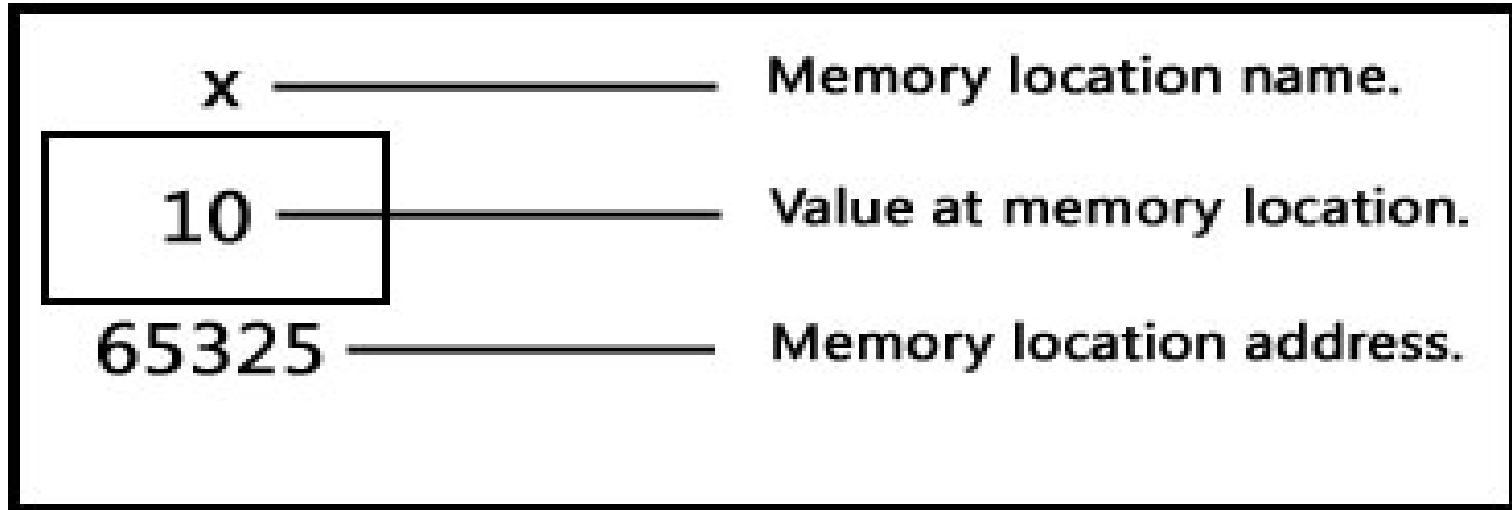
Name this memory location as x.

Store the value 10 at the location 65325.

Use of & and *

- **When we declared an integer variable x and assigned value 10 then compiler occupied a 2 byte memory space at memory address 65325 and stored value 10 at that location.**

- **Compiler named this address x so that we can use x instead of 65325 in our program**

address of variable example

```
Line 1:     #include<stdio.h>
Line 2:     #include<conio.h>
Line 3:     void main()
Line 4:     {
Line 5:     int i=9;
Line 6:     clrscr();
Line 7:     printf("Value of i : %d\n",i);
Line 8:     printf("Address of i : %u",&i);
Line 9:      getch();
Line 10:    }
```

Use of & and *

- **This is a very simple c program which prints value and address of an integer.**

- **But did you notice line no. 8 in above program?**

- **This line output the address of i variable and to get address of i variable we have used ampersand (&) operator.**

- **This operator is known as "*Address of*" operator and we already used this operator many times in our program, just recall scanf statement which is used to accept input from computer keyboard.**

- **So when we use ampersand operator (&i) before any variable then we are instructing c compiler to return its address instead of value.**

- **Another operator is "*" called "*Value at address*" operator. It is the same operator which we use for multiplication of numbers.**

Use of & and *

- As the name suggest, "*value at address*" operator returns value stored at particular address.

- The "*value at address*" operator also called indirection operator.

- Following example extends above C program and puts "*value at address*" operator in action.

Value at address (*) example:

```
#include<stdio.h>
#include<conio.h>
 void main()
{
   int i=9;
   clrscr();
    printf("Value of i : %d\n",i);
   printf("Address of i : %u\n",&i);
   printf("Value at address of i : %d",*(&i));
   getch();
}
```

Benefits of using Pointer in C Program

- Pointer is one of the most exciting features of C language and it has added power and flexibility to the language.

- Pointer is in C language because it offers following benefits to the programmers:

1.Pointers can handle arrays and data table efficiently.

2.Pointers support dynamic memory management.

3.Pointer helps to return multiple values from a function through function argument.

4.Pointer increases program execution speed.

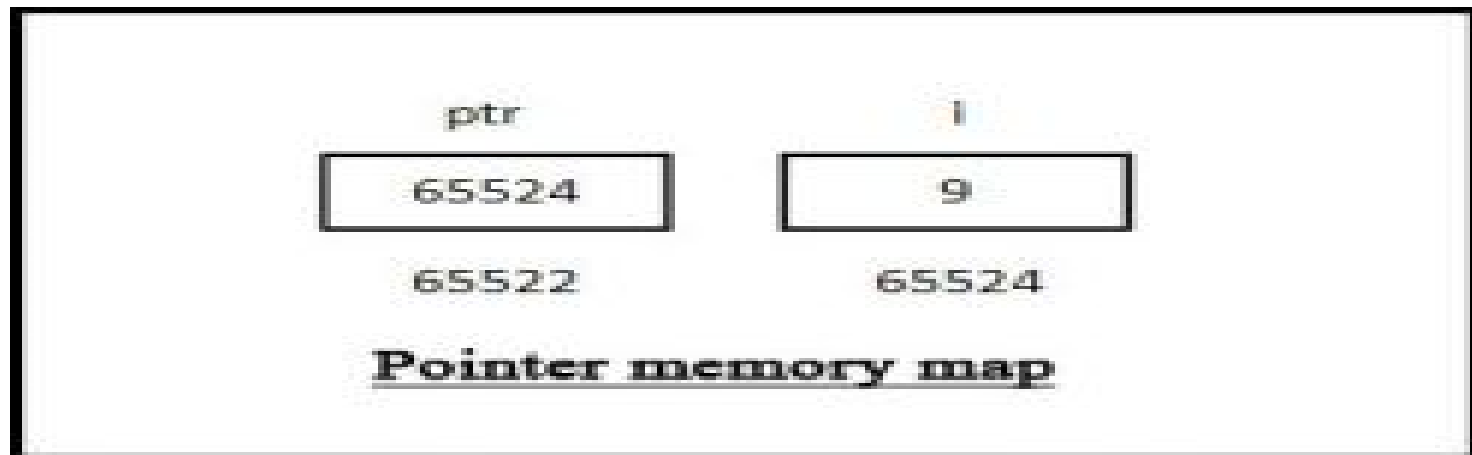5.Pointer is an efficient tool for manipulating structures, linked lists, queues stacks etc

Pointer Declaration

Example on how we can use pointer in our C program.

Syntax: datatype *pointer_name;

Example :   int *iPtr; f**loat *fPtr;**

ptr	i
65524	9
65522	65524

Pointer memory map

Pointers for inter function communication

There are two ways to be discussed with pointers, for inter function communication among pointers.
1. call by value
2. call by reference

Call By Value:

In this mechanism a variable is declared and defined in the called function for each value to the called function.
It means it is one –way communication.
The calling function can send data to the called function, ut the called function cannot send data to the calling function.

Explanation:

- In the above program, two data items are passed from main to the down function.

- One data value is a literal, the other is the value of a variable.

- Downward communication or the call by value is the one way communication.

Call by reference:

- This method is also called as pass by reference or upward communication.

- Here in this method instead of passing the values of the variables to the called

- Function.

- We pass their addresses so that the called function can change the values stored

- In the calling routine.

- This is known as "call by reference" since we are referencing the variables.

```c
// Function Declarations
void exchange (int x, int y);

int main (void)
{
   int a = 5;
   int b = 7;
   exchange (a, b);
   printf("%d %d\n", a, b);
   return 0;
} // main
```
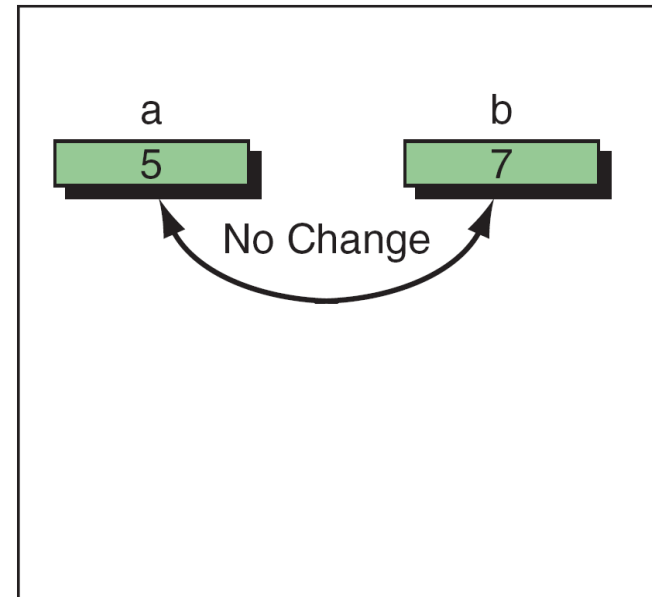
```c
void exchange (int x, int y)
{
   int temp;

   temp = x;
   x    = y;
   y    = temp;
   return;
} // exchange
```

a   b

5   7

No Change

5   7

x   y

As Passed

7   5

x   y

When Exchanged

Fig : An Unworkable Exchange

```c
// Function Declaration
void exchange (int*, int*);

int main (void)
{
    int a = 5;
    int b = 7;

    exchange (&a, &b);
    printf("%d %d\n", a, b);
    return 0;
} // main
```
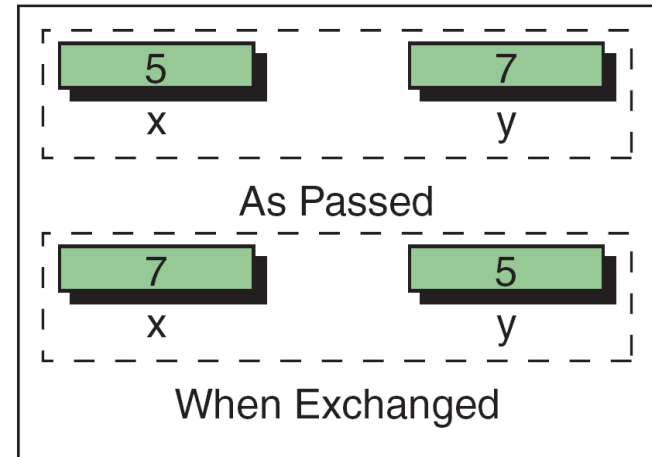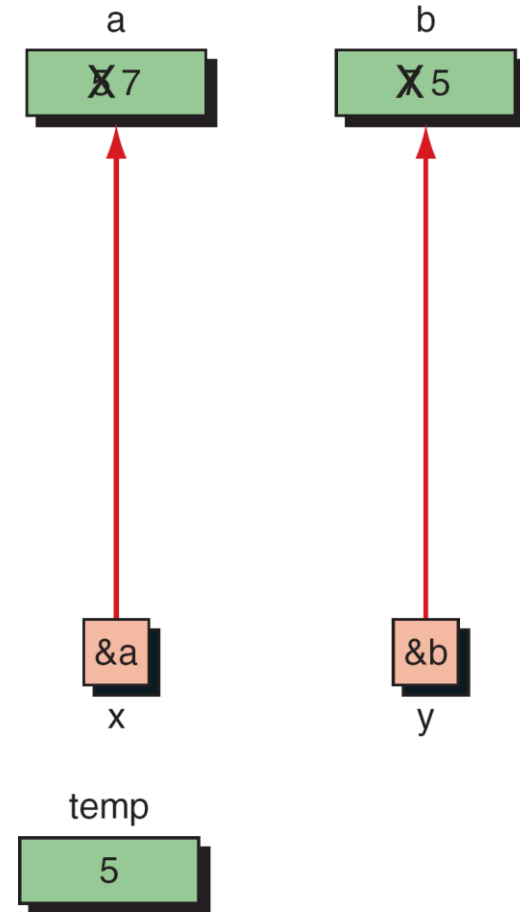
```c
void exchange (int* px, int* py)
{
    int temp;

    temp = *px;
    *px  = *py;
    *py  = temp;
    return;
} // exchange
```

a
X 7

b
X 5

&a
x

&b
y

temp
5

Fig: Exchange Using Pointers

Pointers to Pointers

Pointers to pointers is using of pointers that point to other pointers

```
// Local Declarations
int    a;
int*   p;
int**  q;
```

```
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",     a);
printf(" %3d",   *p);
printf(" %3d", **q);
```
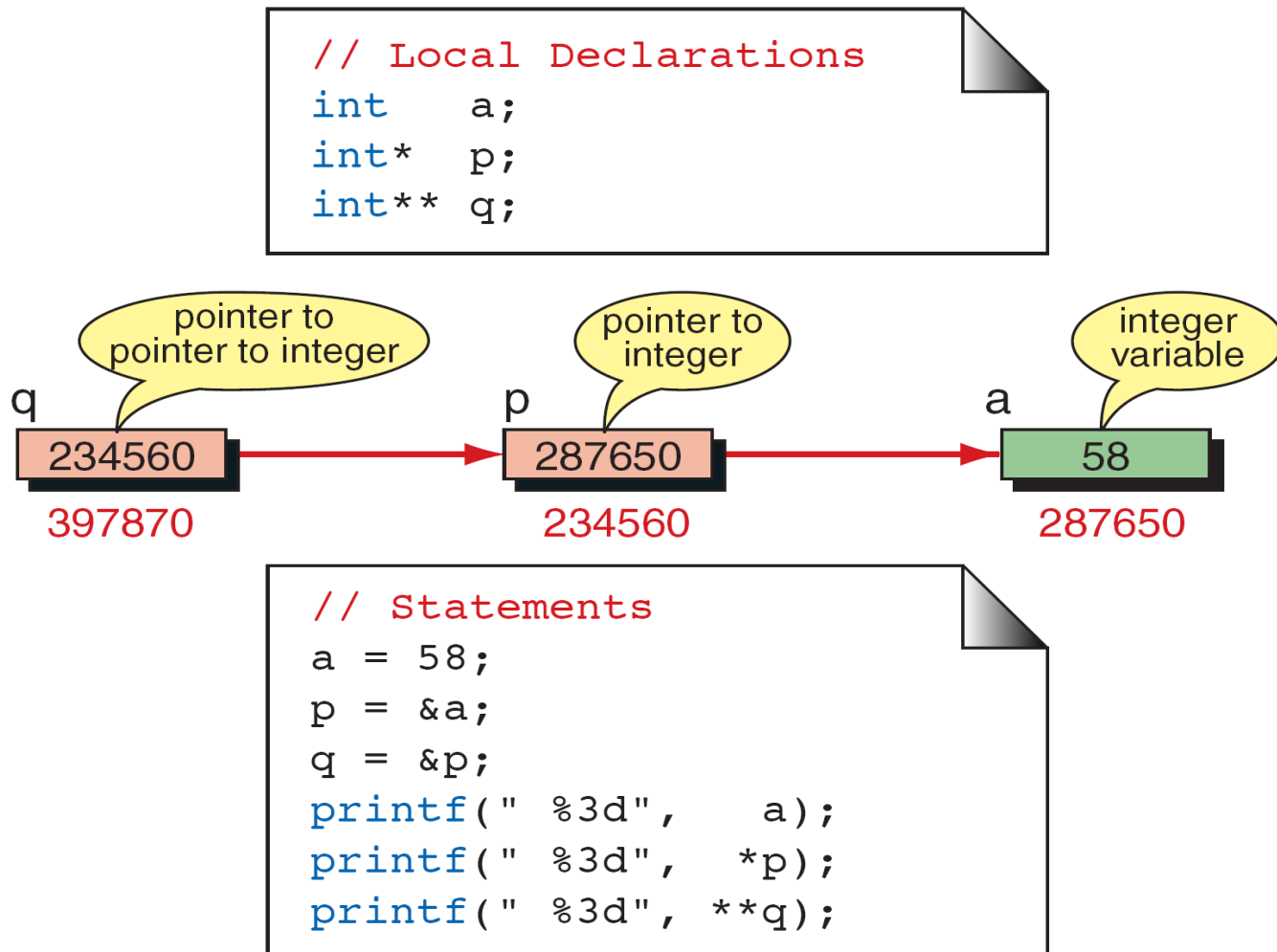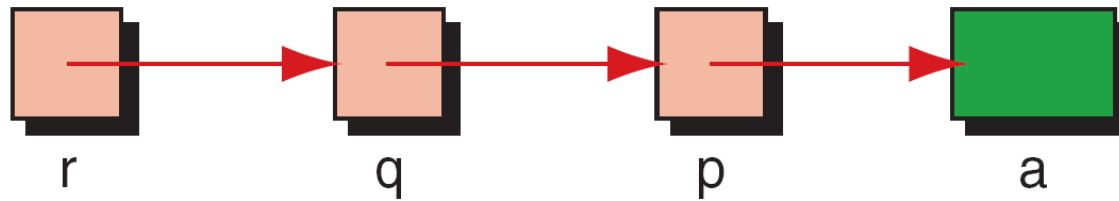
Fig: Pointers to Pointers

Fig: Using Pointers to Pointers

Explanation:

- Each level of pointer indirection requires a separate indirection operator when it is dereference.

- In the above example, to refer to variable a using the pointer p, we have to dereference it once. i.e.*p.

- To refer to variable a using the pointer q, we have to dereference it twice to get the integer a because

- There are two levels of indirection (pointers) invoked. i.e.**q.

Compatibility:

- Pointer have a type associate with it.

- They are not just pointer types but rather are pointers to a specific type

- Such as character.

- Each pointer therefore takes on the attributes of the type to which it refers

- in addition to its own attributes.

Types Of Compatibility:

- **Pointer Size Compatibility.**
- **Déréférence Type Compatibility.**
- **Dereference Level Compatibility**.

Pointer size compatibility:

The size of all pointers is same.
Every pointer holds the address of one memory location in the computer.
Size of the variable that the pointer references can be different.

Trace the following example:

```
int  a;
int  *p;
printf("%d",sizeof(a));
printf("%d",sizeof(p));
printf("%d",sizeof(*p));
```

Dereference type compatibility:

- The dereference type compatibility is the type of the variable that the pointer is referencing.
- In C, we can't use the assignment operator with pointers to different types;
- If we try to, we get a compile error.
- We cannot assign one type of pointer to another type of pointer.
- A pointer to a char is only compatible with a pointer to a char
- And a pointer to an int is only compatible with a pointer to an int.
- We cannot assign a pointer to a char to a pointer to an int.

Construct an example in which we have two variables:

one int and one char

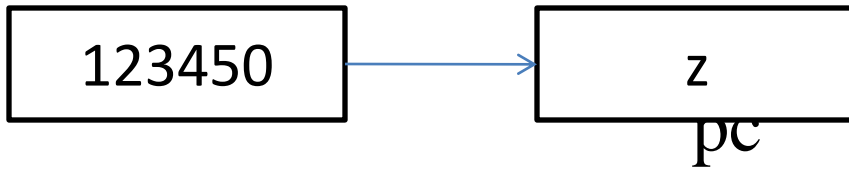**We also define one pointer to char and one pointer to int as shown in the below Figure.**

```
char  c;
char*  pc;

int  a;
int *  pa;

pc=&c;      //are valid
pa=&a;      //are valid
```

type: pointer to char

123450

| 123450 | → | z |

pc   c

type: pointer to int

| 234560 | → | 58 |

234560

pa   a

pc=&a;   //error: different types
pa=&a;   //error: different levels

The first pair of assignments are valid, we store the address of a character variable in a pointer to character variable.

In the second assignment, we store the address of an integer(int) variable in a pointer to an integer(int) variable.

There is an error in the third assignment because we try to store the address of a character variable into a pointer variable whose type is pointer to pointer to integer(int).

We also get an error in the fourth assignment.

Dereference level compatibility

Compatibility also includes dereference level compatibility.

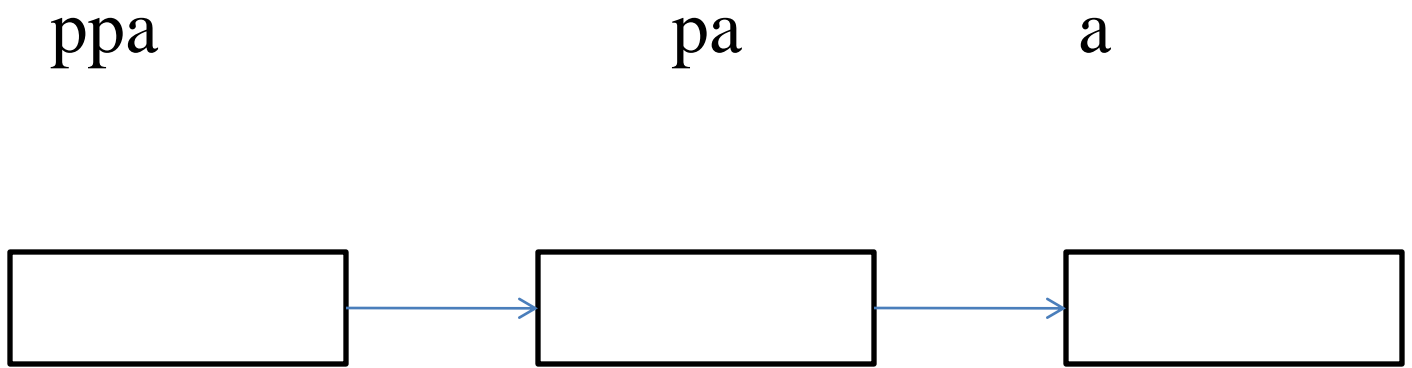For example, a pointer to int is not compatible with a pointer-to-pointer to int.

The pointer to int has a reference type of int, while a pointer-to-pointer to int has a reference type of pointer to int.

- The following figure shows two pointers declared at different levels.

  The pointer pa is a pointer to int;

-
  The pointer ppa is a pointer-to-pointer to int.

ppa   pa   a

- int a; //type int

- int b;   //type int

- int* pa; //type pointer to int

- int**   ppa;// type pointer to   pointer to int

  pa=&a;//valid : same level

- ppa=&pa; //valid : same level

- b=**pa; //valid : same level

  pa=&a;//invalid: different level

- ppa=pa ;//invalid: different level

- b=*ppa; //invalid: different level

Pointer to void

The exception to the reference type compatibility rule is the pointer to void.

A pointer to void is a generic type that is not associated with a reference type; that is, it is not the address of a character , an integer, a real, or any other type.

One restriction, void pointer has no object type, it cannot be dereferenced unless it is cast.

The following declaration shows how we can declare a variable of pointer to void type.

```
void*  pvoid;
```

It is important to understand the difference between a null pointer and a variable pointer to void.

A null pointer is a pointer of any type that is assigned the constant NULL.

The reference type of the pointer will not change with the null assignment.

A variable of pointer to void is a pointer with no reference type that can store only the address of any variable.

The following example shows the difference

void* pvoid;   //pointer to void type

int* pint=NULL;   //NULL pointer of type int

char* pchar=NULL;   //NULL pointer of type char

A **void pointer** cannot be dereferenced.

- ## **Casting pointers**

-

  The problem of type incompatibility can be solved by using casting.

-

  For example, if we need to use the char pointer, pc in the previous example, to point to an int(a), we could cast it as shown below

-

  int* pc;

-

  pc=(char*)&a;

- **Lvalue and rvalue**
- **Every expression has a value.**
- **The value of the expression after evaluation can be used in two ways:**
  (i) lvalue
  (ii) rvalue
- **Lvalue: It is used whenever the object is receiving a value, it is being modified.**
- **Rvalue: it is used to supply a value for further use.**
- **What does=(equal to) really mean?**

```
int  f(void)
{
int  s=1;
int t=1;
t=s;
t=2;
```

- **}**

- Left side of = is an "lvalue"
- it evaluates to a location(address)!
- Right side of= is an "rvalue"
- it evaluates to a value
- There is an implicit * when a variable is used as an rvalue!

Arrays and Pointers

- **The name of an array is a pointer constant to the first element.**

- **Because the array's name is a pointer constant, its value cannot be changed.**

- **Since the array name is a pointer constant to the first element,**

- **the address of the first element and the name of the array both represent the same location in memory.**

Pointers to Arrays
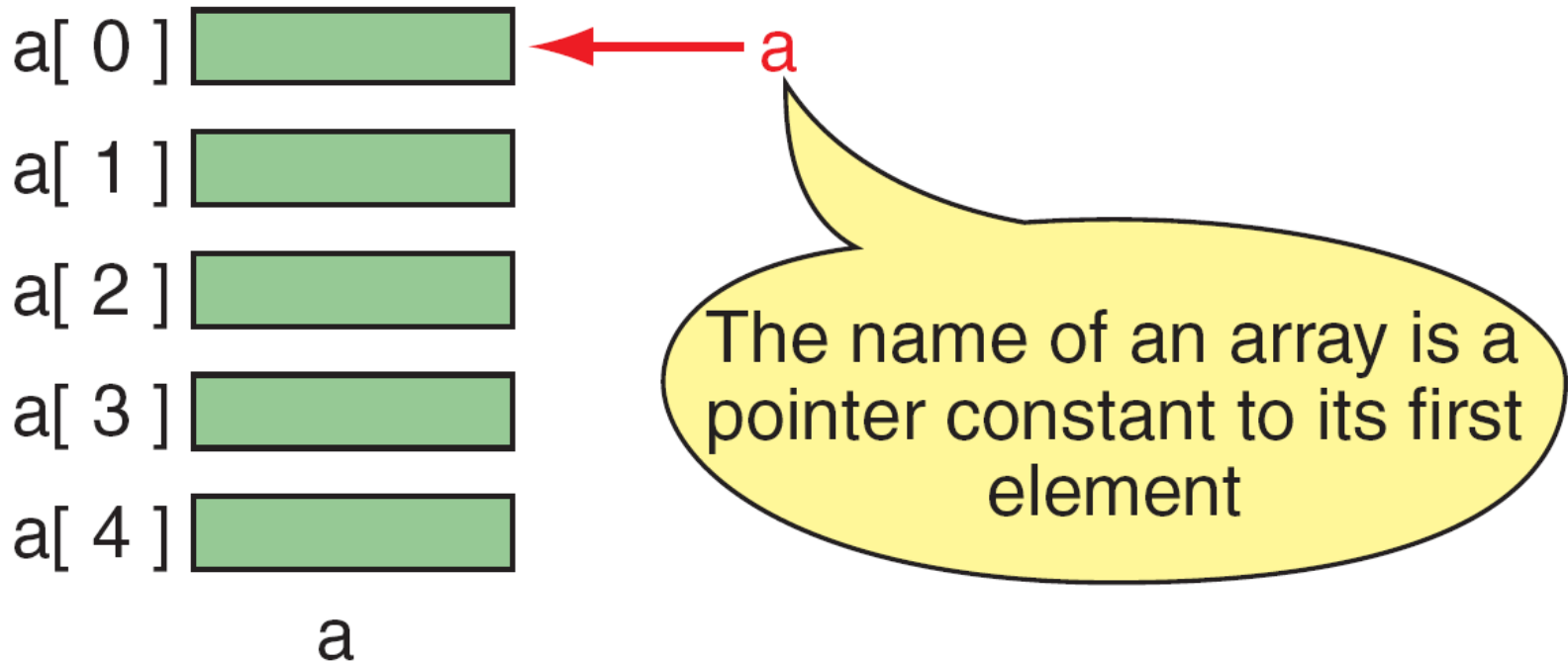
FIGURE   Pointers to Arrays
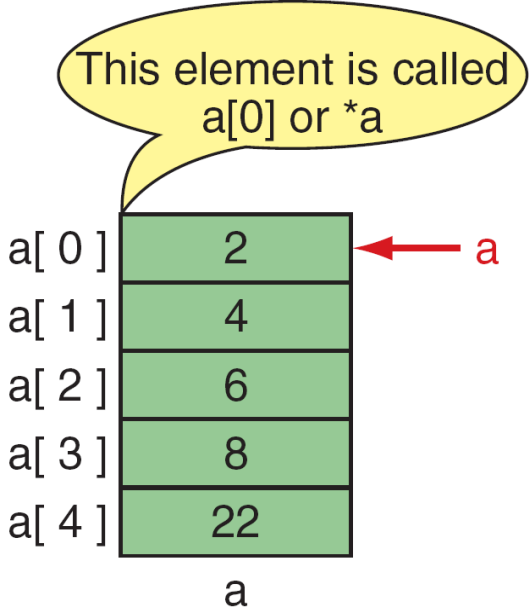
Pointers to Arrays

same

a ←———— &a[0]

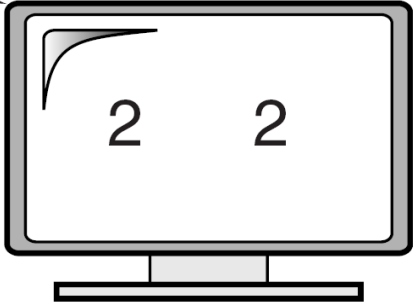- *a* is a pointer only to the first element—not the whole array.

The name of an array is a pointer constant;   it cannot be used as an *lvalue*.
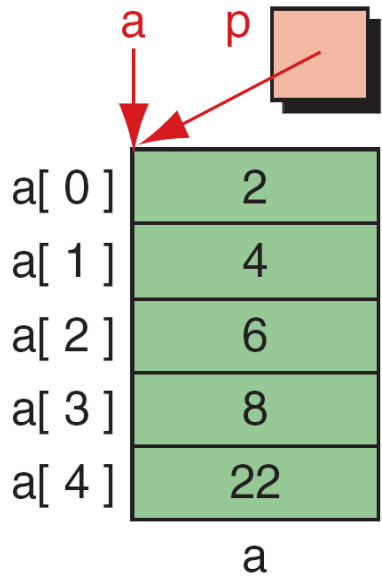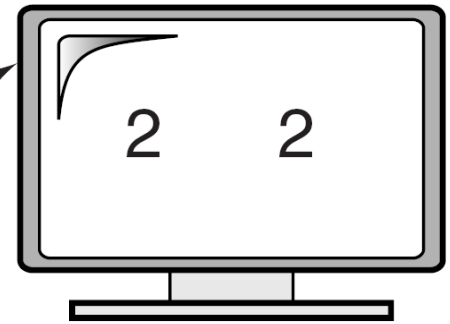
Dereference of Array Name

Dereference of Array Name

Array Names as Pointers


```c
#include <stdio.h>
int main (void)
{
int  a[5] = {2, 4, 6, 8, 22};
int* p    =  a;
   …
   printf("%d %d\n", a[0], *p);
   …
   return 0;
} // main
```

Array Names as Pointers

Multiple Array Pointers
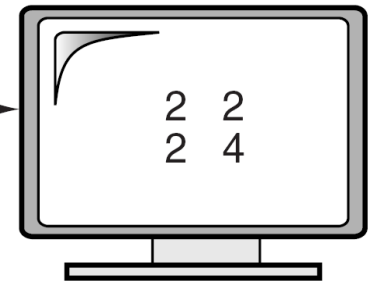
Multiple Array Pointers

To access an array, any pointer to the first element can be used instead of the name of the array.

Pointer Arithmetic and Arrays

- Given pointer, p, p ± n is a pointer to the value n elements away.
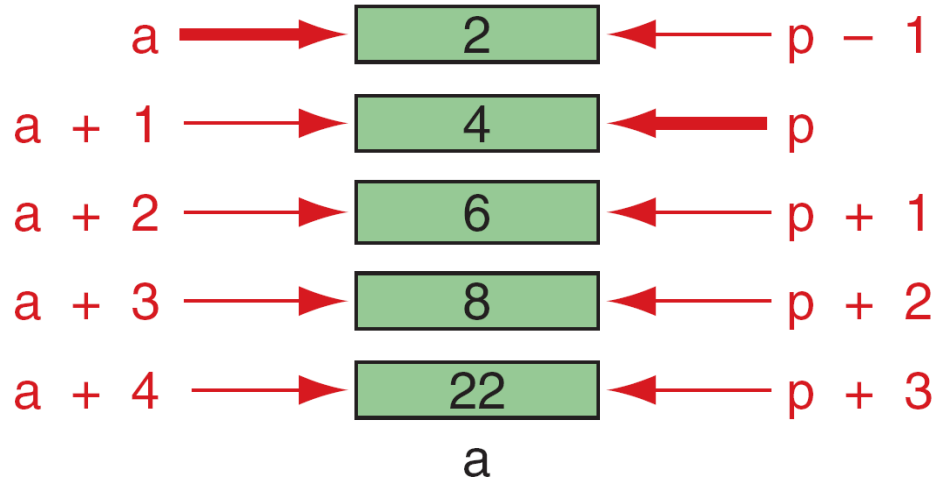
Pointer Arithmetic

Pointer Arithmetic

a + n

$$a + n * (sizeof (one element))$$

a → 100
a + 1 → 101
a + 2 → 102

memory addresses

b → 100
b + 1 → 104
b + 2 → 108

c → 100
c + 1 → 106
c + 2 → 112
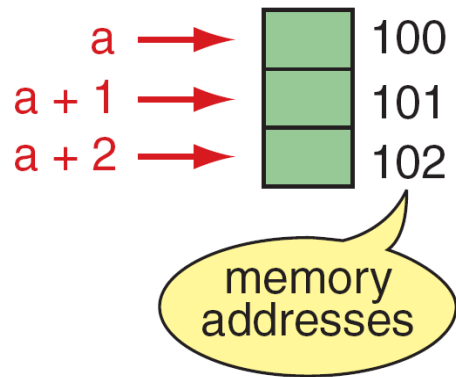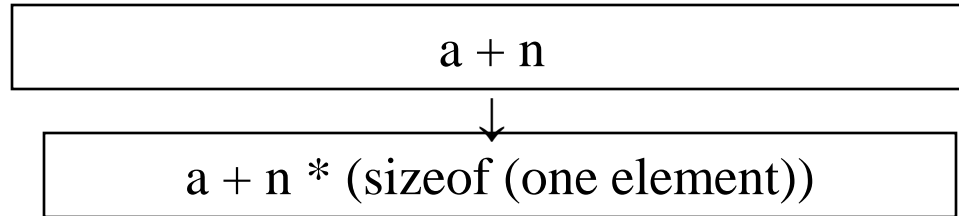
```
char   a[3];
int    b[3];
float  c[3];
```
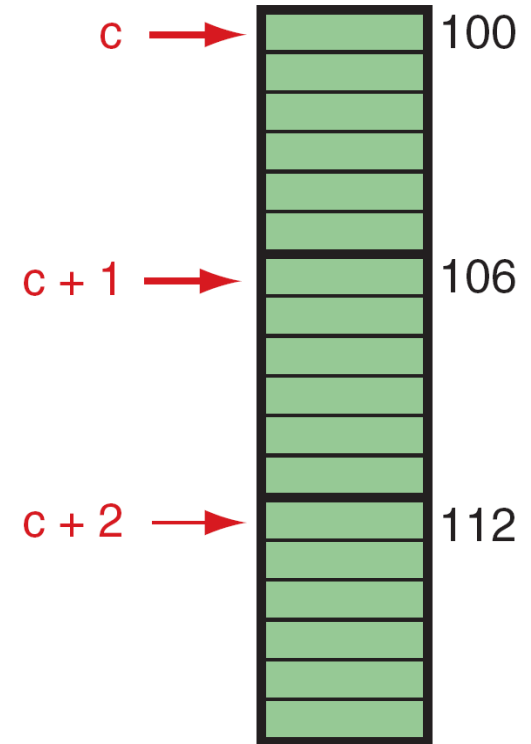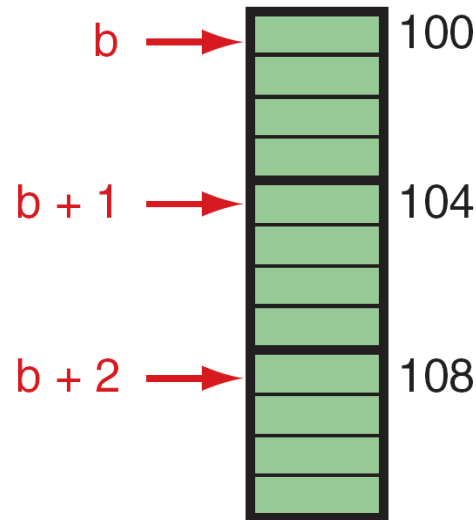
Pointer Arithmetic and Different Types

Dereferencing Array Pointers

a [0]	or	*(a + 0)
a [1]	or	*(a + 1)
a [2]	or	*(a + 2)
a [3]	or	*(a + 3)
a [4]	or	*(a + 4)

2 ← a
4 ← a + 1
6 ← a + 2
8 ← a + 3
22 ← a + 4

a
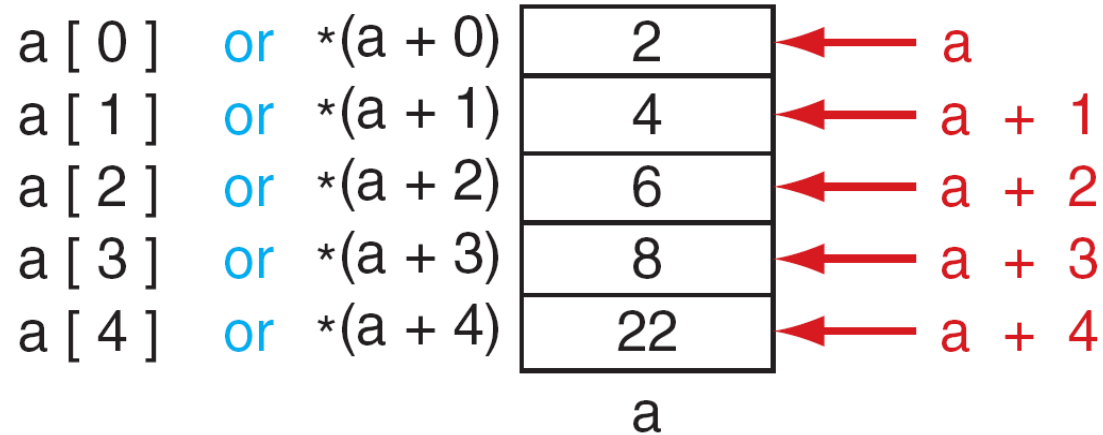
Dereferencing Array Pointers

The following expressions are identical.*(a + n)   and   a[n]

Arithmetic Operations on Pointers

Long Form	Short Form
if (ptr == NULL)	if (!ptr)
if (ptr != NULL)	if (ptr)

Pointers and Relational Operators
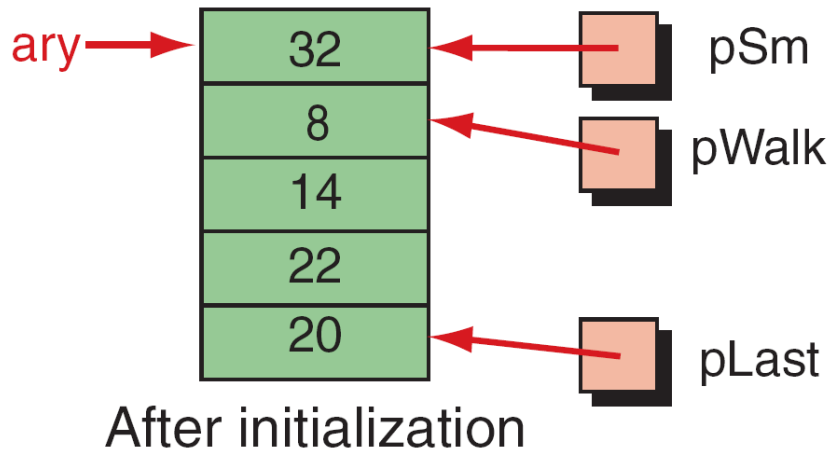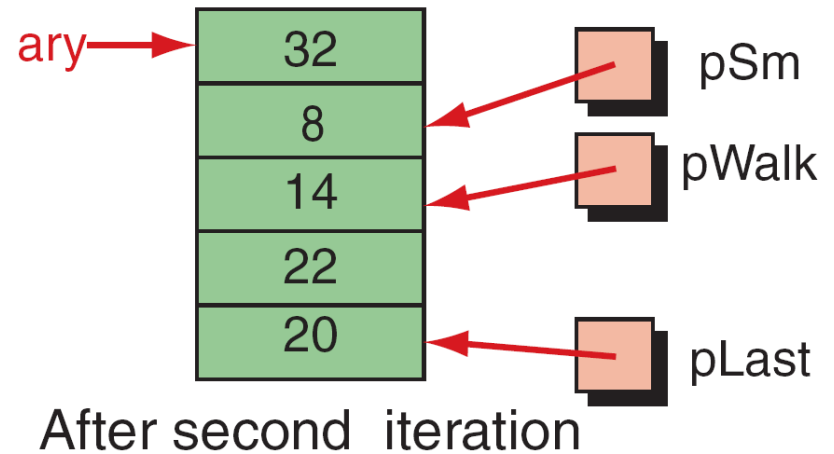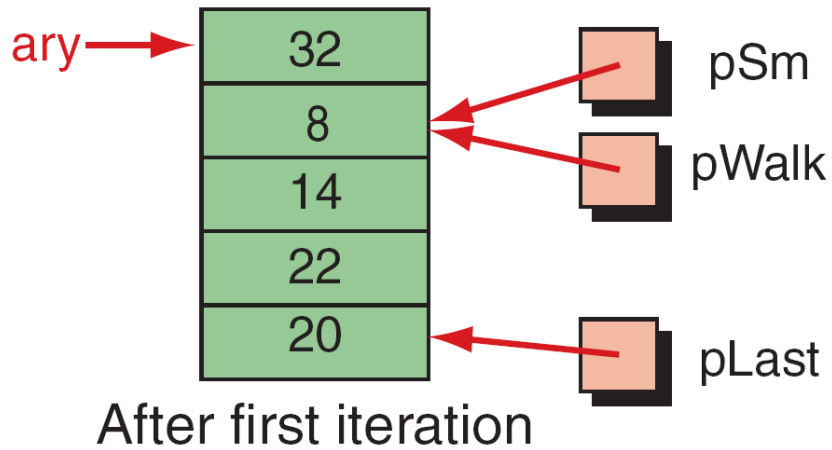
FIGURE (Part I) Find Smallest
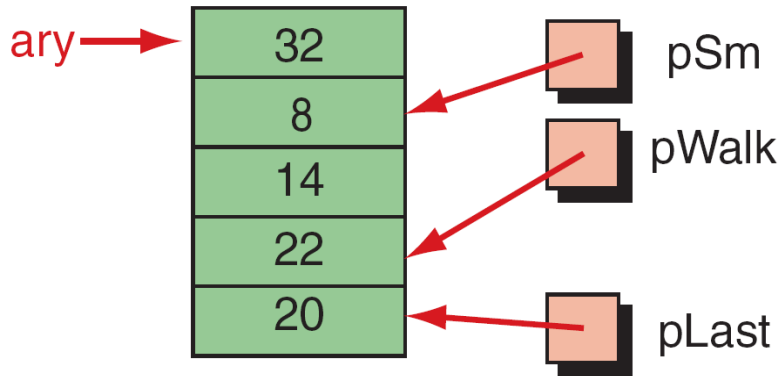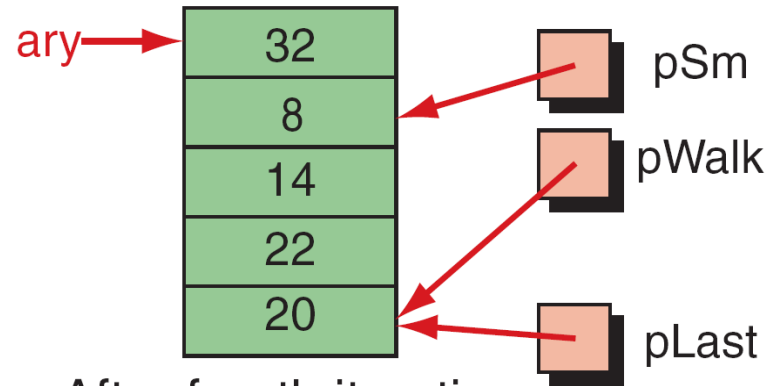
```
pLast = ary + arySize - 1;
for (pSm = ary, pWalk = ary + 1;
     pWalk <= pLast;
     pWalk++)
   if (*pWalk < *pSm)
       pSm = pWalk;
```

FIGURE (Part II) Find Smallest

Pointers And Two-dimensional Arrays

table →

table [0]   or   *(table + 0)

table+1 →

table[1]   or   *(table + 1)

table+2 →

table[2]   or   *(table + 2)
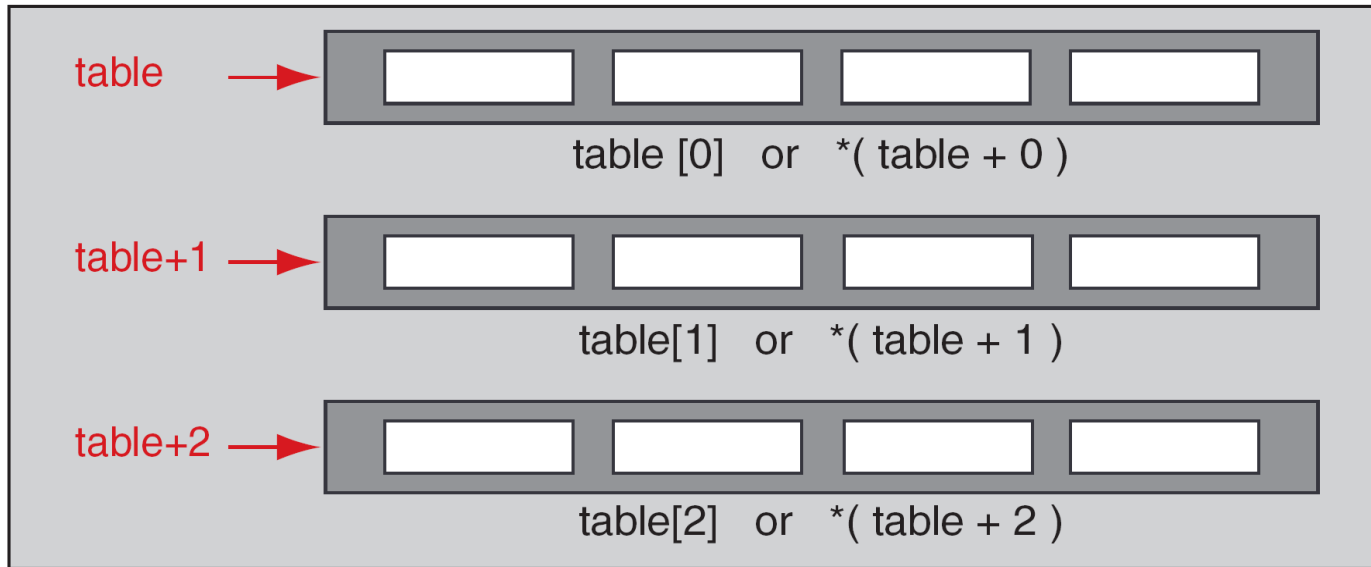
int table[3][4] ;

```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } // for i
```

Print Table

POINTERS AND TWO DIMENSIONAL ARRAY

- **Individual elements of the array mat can be accessed using either: mat[i][j] or *(*(mat + i) + j) or*(mat[i]+j);**

- **See pointer to a one dimensional array can be declared as,**

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

  int *parr;

  parr=arr;

- **Similarly, pointer to a two dimensional array can be declared as,**

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

  int (*parr)[2];

  parr=arr;

- **Look at the code given below which illustrates the use of a pointer to a two dimensional array.**

- ```c
  #include<stdio.h>
  main()
  {      int arr[2][2]={{1,2}.{3,4}};
         int i, (*parr)[2];
         parr=arr;
         for(i=0;i<2;i++)
         {        for(j=0;j<2;j++)
                         printf(" %d", (*(parr+i))[j]);
         }
  }
  ```
  OUTPUT
  1 2 3 4

Passing an Array to a Function

- **The name of an array is actually a pointer to the first element, we can send the array name to a function for processing.**

- **When we pass the array, we do not use the address operator.**

- **Remember, the array name is a pointer constant, so the name is already the address of the first element in the array.**
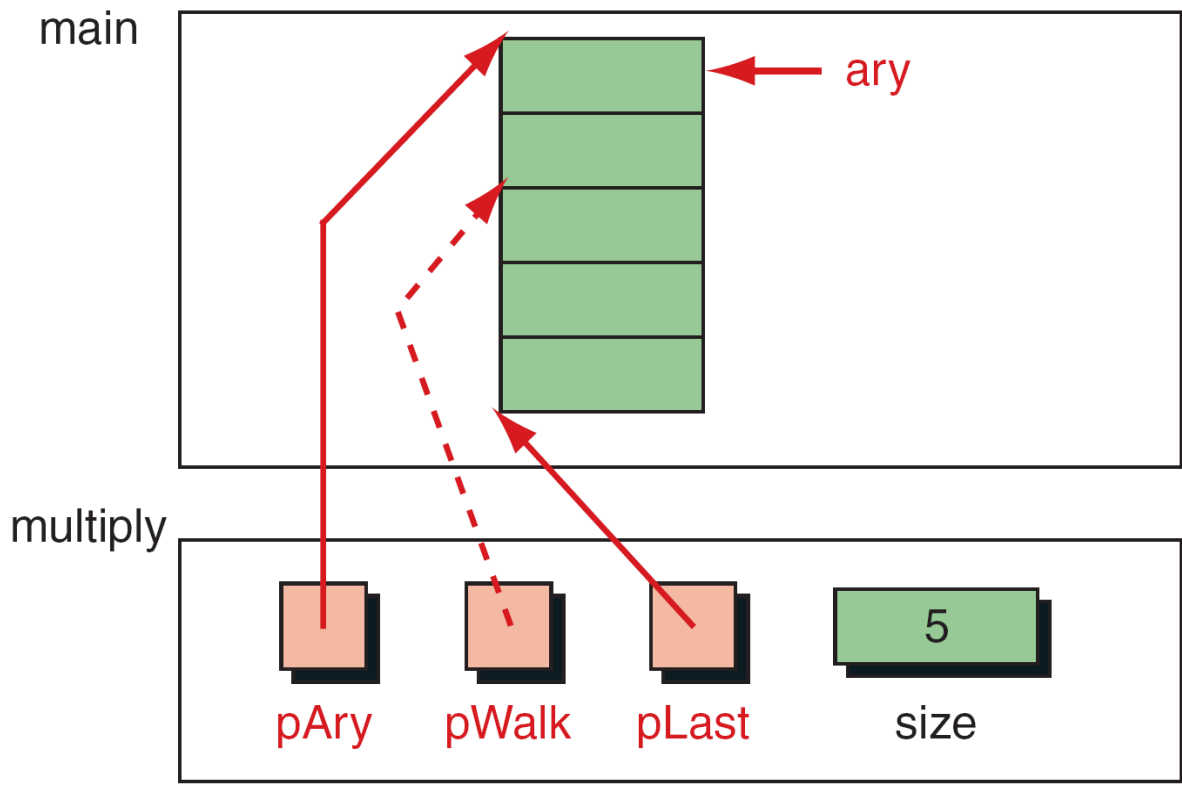
FIGURE   Variables for Multiply Array Elements By 2

Memory Allocation Functions

C gives us two choices when we want to reserve memory locations for an object: static allocation and dynamic allocation.

static memory allocation:

It requires that the declaration and definition of memory be fully specified in the source program.
The number of bytes reserved cannot be changed during runtime.

Dynamic memory allocation:

It uses predefined function to allocate and release memory for data while the program is running.

It effectively postpones the data definition, but not the data declaration, to run time.

(a) Static Memory Allocation
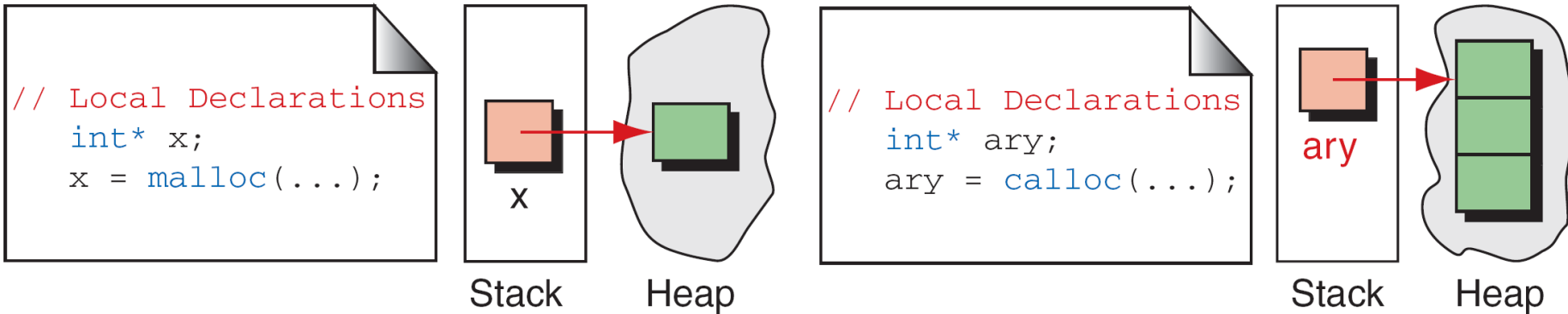
(b) Dynamic Memory Allocation

FIGURE Accessing Dynamic Memory

MEMORY USAGE

We can refer to memory allocated in the heap only through a pointer.

Dynamic memory allocation has no identifier associated with it; it has only an address that must be used to access it.

To access data in dynamic memory, therefore, we must use a pointer.
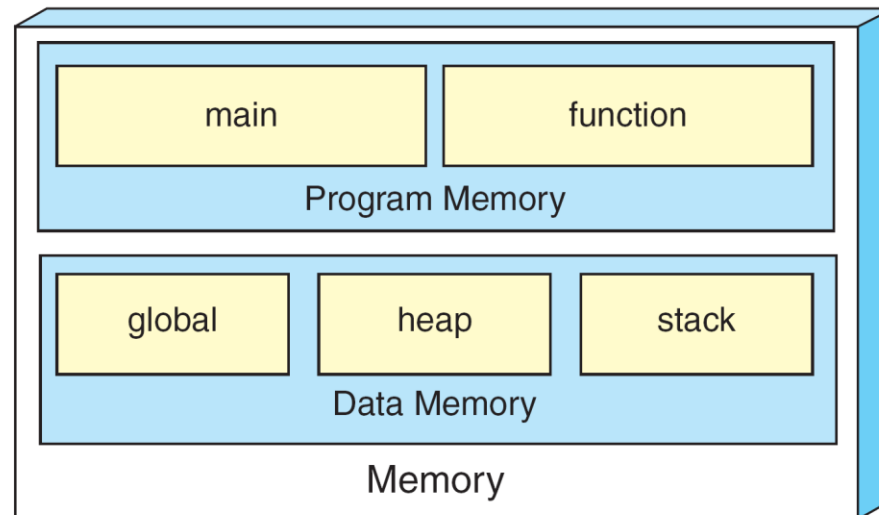
FIGURE   A Conceptual View of Memory

Memory management functions

There are four memory management functions used with dynamic memory.

They are:

(i)malloc()

(ii)calloc()

(iii)realloc()

 (iv)free()

Malloc(): Block memory allocation is Malloc function.

The Malloc function allocates a block of memory that contains the number of bytes specified in its parameter.

It returns a void pointer to the first byte of the allocated memory.

The allocated memory is not initialized.

```
if (!(pInt = malloc(sizeof(int))))
    // No memory available
    exit (100) ;
// Memory available
...
```


pInt

Stack   Heap
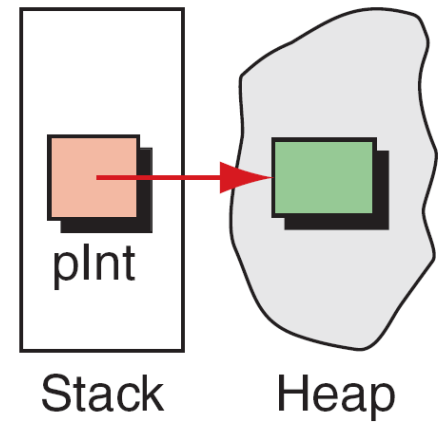
FIGURE **malloc**

The malloc function declaration is as shown below.

    Void* malloc(size_t size);

the type, size_t, is defined in several header files including stdio.h.
The processed data finds a place in a heap rather than stack.
The contents of the heap should be free immediately after operation.

For example:

```
void*  malloc(size_t  size);
main()
{
int  *p;
 p=(int*)malloc(sizeof(int));
*p=60;

 printf("*p=%d",*p);
 printf("*p=%u",p);
 printf("*p=%u",&p);
 free(p);
 }
```

calloc(): It is contiguous memory allocation function.
It is primarily used to allocate memory for arrays.

ptr

200 integers
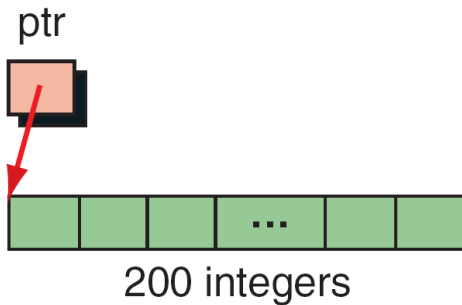
```c
if (!(ptr = (int*)calloc (200, sizeof(int))))
    // No memory available
    exit (100) ;

// Memory available
…
```

- It differs from malloc only in that it sets memory to null characters. The calloc function declaration is shown below.
  void *calloc(size_t element_count,size_t element_size);

- The result is the same for both malloc() and calloc() when overflow occurs and when a zero size is given.

- Realloc(): realloc is the reallocation of memory .
  The operation of realloc() is as shown below:
  void *realloc(void* ptr,size_t,newsize);
  realloc() is highly inefficient.
  It changes the size of the block by deleting or extending the memory at the end of the block.

- If the memory cannot be extended because of other allocations, realloc() allocates a completely new block, copies the existing memory allocation to the new allocation, and deletes the old allocation.
- For example:
ptr=realloc(ptr,15*sizeof(int));

Free():

it is used for releasing memory.

It is an error:

(1)To free memory with a NULL pointer.

(2)A pointer to other than the first element of an allocated block.

(3)A pointer that is different type than the pointer that allocated the memory.

(4)Referring to a memory after releasing it, which is a logical error.

void free(void* ptr);

Releasing memory doesn't change the value in a pointer.

It still contains the address in a heap.

Immediately after freeing the memory, the pointer should be cleared by setting it to NULL.

The pointer used to free memory must be of the same type as a pointer used to allocate the memory.

- **Array of pointers**
  Array name itself is an address or pointer.

- Name of an array indicates the address of the first cell.

- The address of the first byte is often known as base address. Arrays are stored in contiguous memory location.

- This structure is especially helpful when the number of elements in the array is variable.
  Syntax declaration:
  <type> *variable_name[size];

-

  Example:
  int *pa[5];

  Here, pa is a 5 element array of pointers to integer quantities.
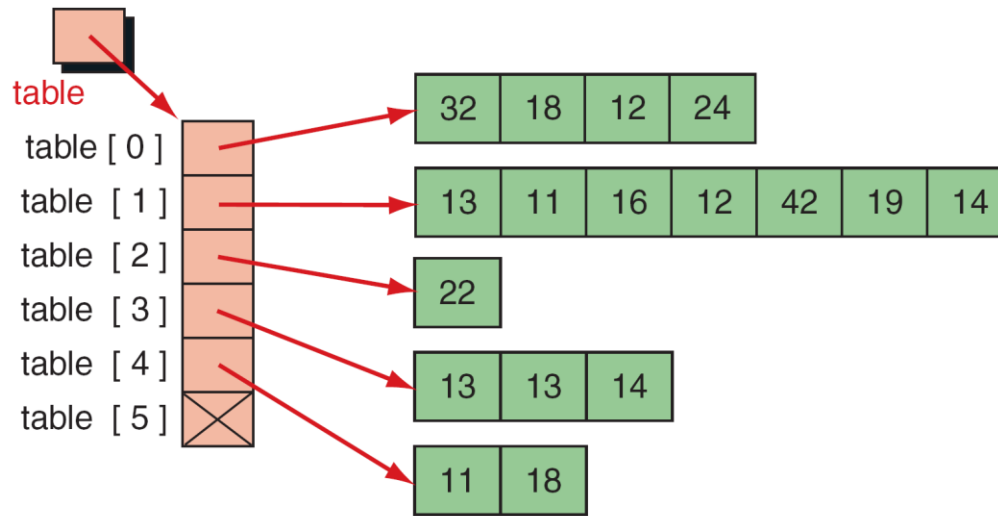
FIGURE   A Ragged Array

- include<stdio.h>
```c
#include<conio.h>
void main()
{
double a,b,c;
double  *pa[5];
clrscr();
a=2.3;
b=6.7;
c=1.3;
pa[0]=&a;
pa[1]=&b;
pa[2]=&c;
printf("\n a=%lf",a);
printf("\n b=%lf",b);
printf("\n c=%lf",c);
getch();
}
```

PROGRAMMING APPLICATIONS

pointers can be used with:
(i)arrays
(ii)functions
(iii)pointers
(iv)structures
(v)dynamic memory allocation

POINTER TO VOID

- The exception to the reference type compatibility rule is the pointer to void

- A pointer to void is a generic type that is not associated with a reference type; that is, it is not the address of a character , an integer, a real, or any other type

- One restriction, void pointer has no object type, it cannot be dereferenced unless it is cast.

- The following declaration shows how we can declare a variable of pointer to void type.

       void* pvoid;

It is important to understand the difference between a null pointer and a variable pointer to void.

- A null pointer is a pointer of any type that is assigned the constant NULL.

The reference type of the pointer will not change with the null assignment.

A variable of pointer to void is a pointer with no reference type that can store only the address of any variable.

- The following example shows the difference

  void* pvoid;   //pointer to void type

  int* pint=NULL;   //NULL pointer of type int

  char* pchar=NULL;   //NULL pointer of type char

  A void pointer cannot be dereference.

UNIT-4
Structures and Unions

▪Definition

A structure is a collection of related elements, possibly of different types, having a single name.

Syntax:

```
struct tag_name
{
     data type var_name1;
     data type var_name2;
     data type var_name3;
};
```

Structures

Example:
```
struct student
{
        int  id;
        char name[10];
        float gradepoint;
};
```

- `struct` introduces the definition for structure student

- `student` is the *structure name* and is used to declare variables of the *structure type*

- student contains three members of different types id ,name,gradepoint

Structures

Note

Elements in a structure can be of the same or different types. However, all elements in the structure
should be logically related.

Structures


```
struct TAG
    {

      field list

    } ;
```
Format

```
struct STUDENT
    {
      char id[10];
      char name[26];
      int  gradePts;
    } ; // STUDENT
```
Example

FIGURE Tagged Structure Format

Structures

- Initializing a Structure

Example:
struct student
{
       int mark;
       char name[10];
       float average;
} struct student report;

struct student report = {100, "Mani", 99.5};

Structures

❖Accessing structures

•Dot operator is used to access members of a structure.

Example
struct student
{
        int mark;
        char name[10];
        float average;
} struct student report;

struct student report = {100, "Mani", 99.5};
report.mark
report.name
report.average

Structures
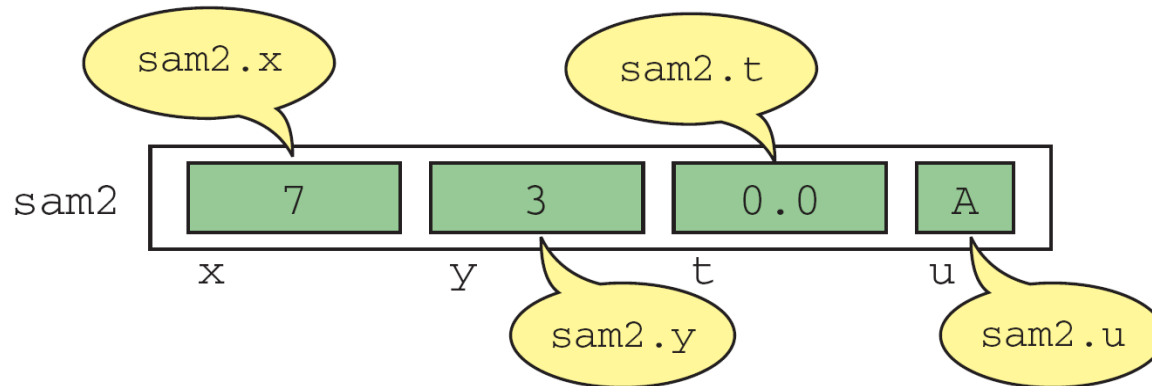
Structure Direct Selection Operator

Structures

Example
```c
#include <stdio.h>
#include <string.h>
 struct student
{        int id;
        char name[20];
        float percentage;
} record;
 int main()
{          record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
         printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

PROGRAM 12-2   Multiply Fractions

```c
1   /* This program uses structures to simulate the
2      multiplication of fractions.
3         Written by:
4         Date:
5   */
6   #include <stdio.h>
7
8   // Global Declarations
9      typedef  struct
10        {
11          int numerator;
12          int denominator;
13        } FRACTION;
14
15  int main (void)
16  {
17  // Local Declarations
18     FRACTION   fr1;
19     FRACTION   fr2;
20     FRACTION   res;
21
```

Structures

- Operations on Structures

  - Assigning a structure to a structure of the same type

  - Taking the address (`&`) of a structure

  - Accessing the members of a structure

  - Using the `sizeof` operator to determine the size of a structure

Structures

- **Nested structures**

• Nested structure in C is nothing but structure within structure.

• One structure can be declared inside other structure as we declare structure members inside a structure.

•The structure variables can be a normal structure variable or a pointer variable to access the data.

1.Structure within structure in C using normal variable

2.Structure within structure in C using pointer variable

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

struct student_college_detail
{
    int college_id;
    char college_name[50];
};
 struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;
```

```c
int main()
{
    struct student_detail stu_data = {1, "Raju", 90.5, 71145,
                            "Anna University"};
    printf(" Id is: %d \n", stu_data.id);
    printf(" Name is: %s \n", stu_data.name);
    printf(" Percentage is: %f \n\n", stu_data.percentage);

    printf(" College Id is: %d \n", stu_data.clg_data.college_id);
    printf(" College Name is: %s \n", stu_data.clg_data.college_name);
    return 0;
}
```

Structures

- Pointers in a structures

  - Structure may contain the pointer variable as member

  - Pointer are used to store address of memory location

  - They can be derefenced by the '*' operator

Structures

- Pointers in a structures
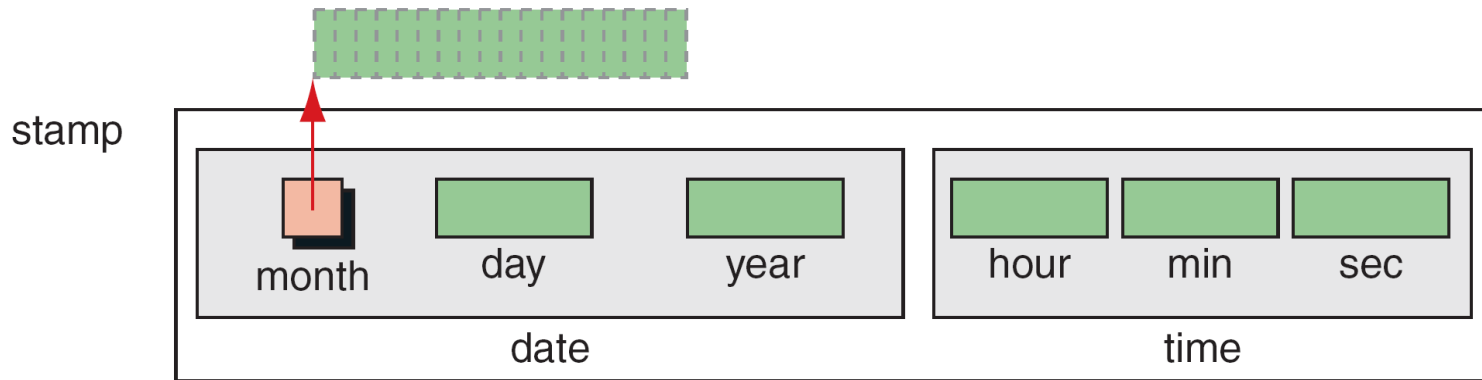
FIGURE Pointers in Structures

Structures

- Structures and functions

  - Structure can be passed to a function as a parameter

  - Functions can also have a structure as return type

Structures

- Structures and functions


```
…
res.numerator =
    multiply(fr1.numerator, fr2.numerator)
res.denominator =
    multiply(fr1.denominator, fr2.denominator);
…
```
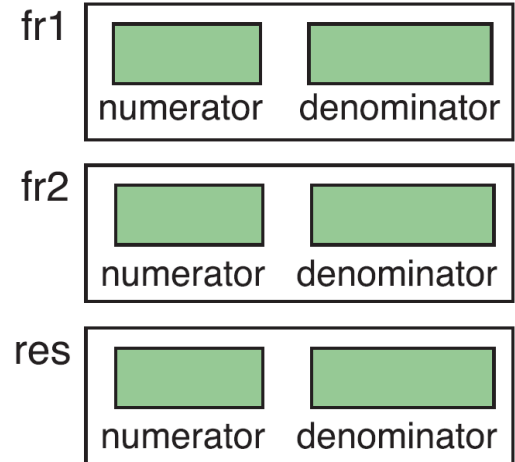
```
// ============== multiply ==============
multiply (int x, int y)
{
    return x * y;
}  // multiply
```

FIGURE   Passing Structure Members to Functions

PROGRAM 12-5 Passing and Returning Structures

```c
1   /* This program uses structures to multiply fractions.
2          Written by:
3          Date:
4   */
5   #include <stdio.h>
6
7   // Global Declarations
8   typedef   struct
9          {
10          int numerator;
11          int denominator;
12         } FRACTION;
13
14  // Function Declarations
15  FRACTION getFr     (void);
16  FRACTION multFr  (FRACTION fr1, FRACTION fr2);
17  void       printFr (FRACTION fr1, FRACTION fr2,
18                        FRACTION result);
19
```

```
20  int main (void)
21  {
22  // Local Declarations
23      FRACTION fr1;
24      FRACTION fr2;
25      FRACTION res;
26
27  // Statements
28      fr1 = getFr  ();
29      fr2 = getFr  ();
30      res = multFr (fr1, fr2);
31      printFr (fr1, fr2, res);
32      return 0;
33  }  // main
34
```

PROGRAM 12-5 Passing and Returning Structures

```c
35  /* ===================== getFr =====================
36     Get two integers from the keyboard, make & return
37     a fraction to the main program.
38        Pre    nothing
39        Post   returns a fraction
40  */
41  FRACTION getFr (void)
42  {
43  // Local Declarations
44     FRACTION fr;
45
46  // Statements
47     printf("Write a fraction in the form of x/y: ");
48     scanf ("%d/%d", &fr.numerator, &fr.denominator);
49     return fr;
50  }  // getFraction
51
```

PROGRAM 12-5 Passing and Returning Structures

```
52  /* ================== multFr ====================
53     Multiply two fractions and return the result.
54        Pre    fr1 and fr2 are fractions
55        Post   returns the product
56  */
57  FRACTION multFr (FRACTION fr1, FRACTION fr2)
58  {
59  // Local Declaration
60     FRACTION res;
61
62  // Statements
63     res.numerator   = fr1.numerator   * fr2.numerator;
64     res.denominator = fr1.denominator * fr2.denominator;
65     return res;
66  } // multFr
67
```

```
68   /* ==================== printFr ====================
69      Prints the value of the fields in three fractions.
70         Pre    two original fractions and the product
71         Post   fractions printed
72   */
73   void  printFr  (FRACTION fr1,  FRACTION fr2,
74                      FRACTION res)
75   {
76   // Statements
77      printf("\nThe result of %d/%d * %d/%d is %d/%d\n",
78                  fr1.numerator, fr1.denominator,
79                  fr2.numerator, fr2.denominator,
80                  res.numerator, res.denominator);
81      return;
82   } // printFractions
83   // ================== End of Program ==================
```

```
Results:
Write a fraction in the form of x/y: 4/3
Write a fraction in the form of x/y: 6/7

The result of 4/3 * 6/7 is 24/21
```

Structures

• Structures and functions
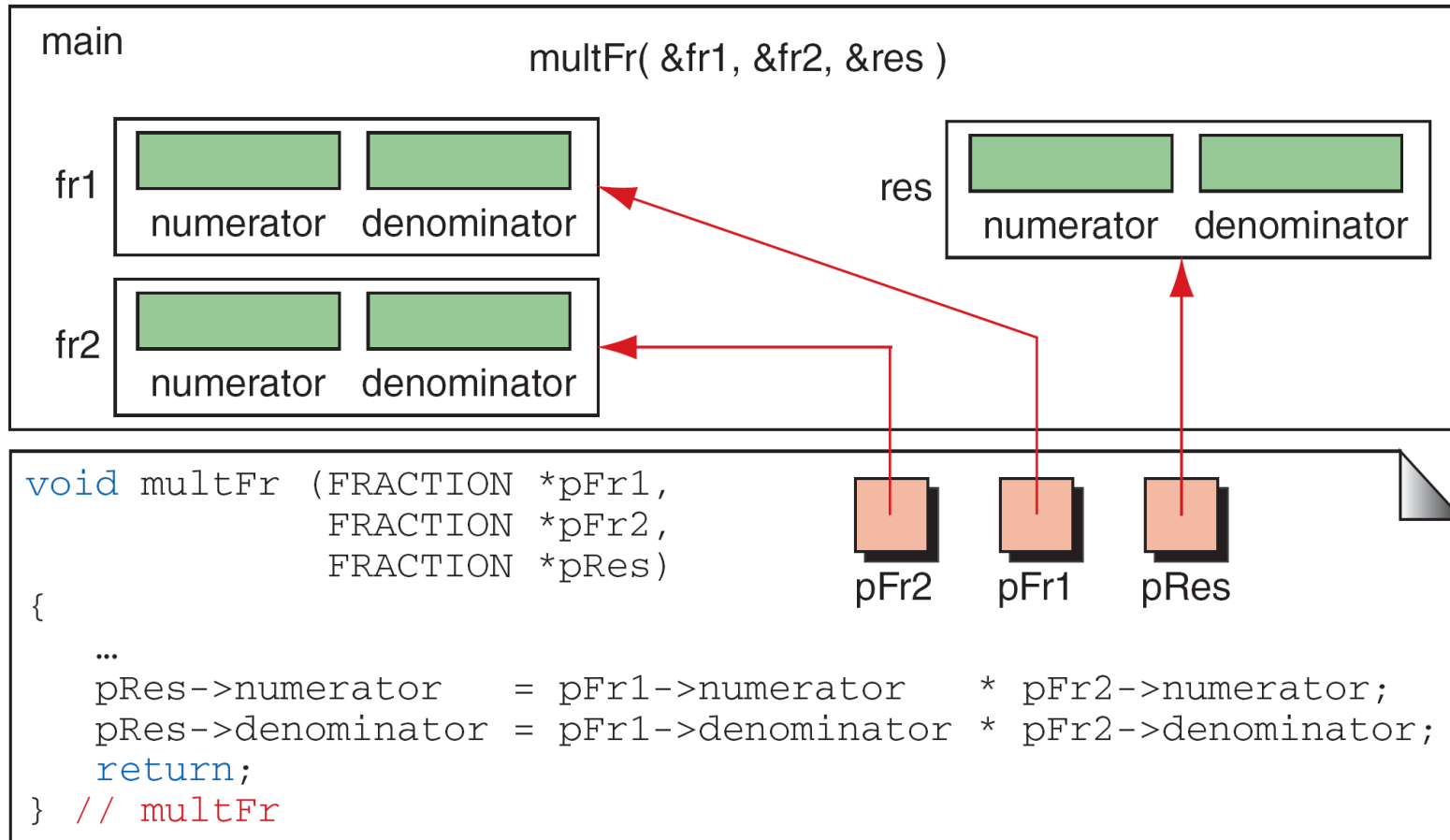
FIGURE   Passing Structures Through Pointers

Structures

- Pointer s to structures

Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable.

Note

(*pointerName).fieldName ⟷ pointerName->fieldName.

Structures

- Pointer s to structures

  Example:
  struct student
  {
         int mark;
         char name[10];
         float average;

  };

struct student *report, rep;
struct student rep = {100, "Mani", 99.5};
report = &rep;
report -> mark
report -> name
report -> average
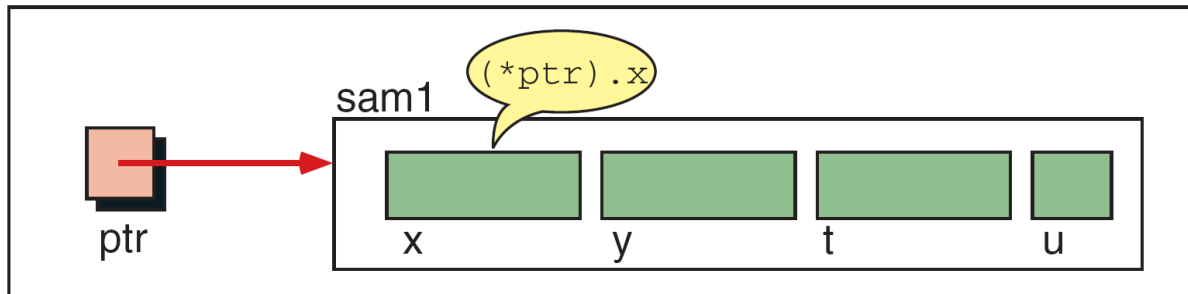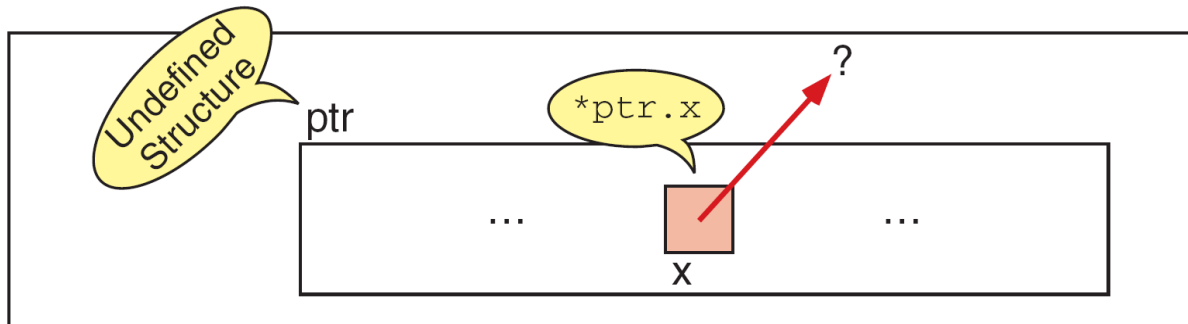
FIGURE Interpretation of Invalid Pointer Use

```c
#include <stdio.h>
#include <string.h>
 struct student
{
    int id;
    char name[30];
    float percentage;
};
 int main()
{int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;
     ptr = &record1;
        printf("Records of STUDENT1: \n");
      printf("  Id is: %d \n", ptr->id);
      printf("  Name is: %s \n", ptr->name);
      printf("  Percentage is: %f \n\n", ptr->percentage);
     return 0;
}
```

```
1    /* This program uses a structure to simulate the time.
2          Written by:
3          Date:
4    */
5    #include <stdio.h>
6
7    typedef struct
8        {
9          int hr;
10         int min;
11         int sec;
12        } CLOCK;
13
14   // Function Declaration
15   void increment (CLOCK* clock);
16   void show       (CLOCK* clock);
17
18   int main (void)
19   {
```

```
40   // Statements
41       (clock->sec)++;
42       if (clock->sec == 60)
43           {
44             clock->sec = 0;
45             (clock->min)++;
46             if (clock->min == 60)
47                 {
48                   clock->min = 0;
49                   (clock->hr)++;
50                   if (clock->hr == 24)
51                       clock->hr = 0;
52                 } // if 60 min
53           } // if 60 sec
54       return;
55   }   // increment
56
```

PROGRAM   Clock Simulation with Pointers

```
57  /* ===================== show =====================
58     Show the current time in military form.
59        Pre    clock time
60        Post   clock time displayed
61  */
62  void show (CLOCK* clock)
63  {
64  // Statements
65     printf("%02d:%02d:%02d\n",
66           clock->hr, clock->min, clock->sec);
67     return;
68  }  // show
```

```
Results:
14:38:57
14:38:58
14:38:59
14:39:00
14:39:01
14:39:02
```

Complex Structures

- Nested Structures
- Self referential structures
- A structure may have
    - Data variables
    - Internal structures/unions
    - Pointer links
    - Function pointers

Structures

Self-Referential Structures

- Self-referential structures contain a pointer member that points to a structure of the same structure type.

Example:

```
struct node {
    int data;
    struct node *nextPtr;
}
```

- nextPtr
  - is a pointer member that points to a structure of the same type as the one being declared.
  - is referred to as a link. Links can tie one node to another node.
- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

Unions

Definition:

The union is a construct that allows memory to be shared by different types of data. This redefinition can be as simple as redeclaring an integer as four characters or as complex as redeclaring an entire structure.

Syntax:
```
union tag_name
{
     data type var_name1;
     data type var_name2;
     data type var_name3;
};
```

Unions

```
typedef struct
  {
   char  first[20];
   char  init;
   char  last[30];
  } PERSON;
typedef struct
  {
   char  type;
   union
     {
      char    company[40];
      PERSON person;
     } un;
  } NAME;
```
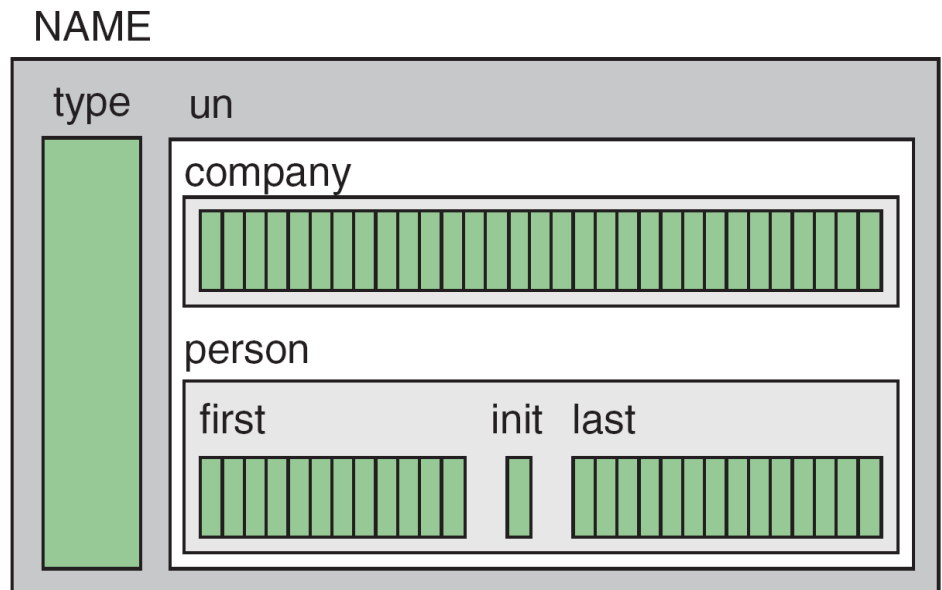

FIGURE A Name Union

PROGRAM 12-7   Demonstrate Effect of Union

```
19   // Statements
20      data.num = 16706;
21
22      printf("Short: %hd\n", data.num);
23      printf("Ch[0]: %c\n",  data.chAry[0]);
24      printf("Ch[1]: %c\n",  data.chAry[1]);
25
26      return 0;
27   }  // main
```

```
Results:
Short: 16706
Ch[0]: A
Ch[1]: B
```

Bit Field

• The variables defined with a predefined width are called bit fields.

• A bit field can hold more than a single bit.

• for example if you need a variable to store a value from 0 to 7
only then you can define a bit field with a width of 3 bits .

Bit Field Declaration
the declaration of a bit-field has the form inside a structure:
struct
{
          type [member_name] : width ;
};

Bit Field

Below the description of variable elements of a bit field:

Elements	Description
type	An integer type that determines how the bit-field's value is interpreted. The type may be int, signed int, unsigned int.
member_name e	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Bitfiled.txt

Structures

❖ Type Definition (*typedef*)

      A type definition, typedef, gives a name to a data type by creating a new type that can then be used anywhere a type is permitted.

Structures

- **Enumerated Types**

The enumerated type is a user-defined type based on the standard integer type. In an enumerated type, each integer value is given an identifier called an enumeration constant.

Syntax:
enum identifier {value1, value2,.... Value n};

- enum is Enumerated Data Type .
  enum is user defined data type
- In the above example "identifier" is nothing but the user defined data type .
- Value1,Value2,Value3….. etc creates one set of enum values.
- Using "identifier" we are creating our variables

Memory Allocation Functions

C gives us two choices when we want to reserve memory locations for an object: static allocation and dynamic allocation.

static memory allocation:

It requires that the declaration and definition of memory be fully specified in the source program.
The number of bytes reserved cannot be changed during runtime.

Dynamic memory allocation:

It uses predefined function to allocate and release memory for data while the program is running.

It effectively postpones the data definition, but not the data declaration, to run time.

MEMORY USAGE

We can refer to memory allocated in the heap only through a pointer.

Dynamic memory allocation has no identifier associated with it; it has only an address that must be used to access it.

To access data in dynamic memory, therefore, we must use a pointer.

Memory management functions

There are four memory management functions used with dynamic memory.

They are:

(i)malloc()

(ii)calloc()

(iii)realloc()

 (iv)free()

Malloc(): Block memory allocation is Malloc function.

The Malloc function allocates a block of memory that contains the number of bytes specified in its parameter.

It returns a void pointer to the first byte of the allocated memory.

The allocated memory is not initialized.

The malloc function declaration is as shown below.
   Void* malloc(size_t size);
the type, size_t, is defined in several header files including stdio.h.
The processed data finds a place in a heap rather than stack.
The contents of the heap should be free immediately after operation.

For example:
```
void*  malloc(size_t  size);
main()
{
int  *p;
 p=(int*)malloc(sizeof(int));
*p=60;

 printf("*p=%d",*p);
 printf("*p=%u",p);
 printf("*p=%u",&p);
 free(p);
 }
```

calloc(): It is contiguous memory allocation function.
It is primarily used to allocate memory for arrays.

- It differs from malloc only in that it sets memory to null characters. The calloc function declaration is shown below.

  void *calloc(size_t element_count,size_t element_size);

- The result is the same for both malloc() and calloc() when overflow occurs and when a zero size is given.

- Realloc(): realloc is the reallocation of memory .
  The operation of realloc() is as shown below:
  void *realloc(void* ptr,size_t,newsize);
  realloc() is highly inefficient.
  It changes the size of the block by deleting or extending the memory at the end of the block.

- If the memory cannot be extended because of other allocations, realloc() allocates a completely new block, copies the existing memory allocation to the new allocation, and deletes the old allocation.
- For example:
  ptr=realloc(ptr,15*sizeof(int));

Free():

it is used for releasing memory.

It is an error:

(1)To free memory with a NULL pointer.

(2)A pointer to other than the first element of an allocated block.

(3)A pointer that is different type than the pointer that allocated the memory.

(4)Referring to a memory after releasing it, which is a logical error.

void free(void* ptr);

Releasing memory doesn't change the value in a pointer.

It still contains the address in a heap.

Immediately after freeing the memory, the pointer should be cleared by setting it to NULL.

The pointer used to free memory must be of the same type as a pointer used to allocate the memory.

UNIT-5
FILES

File:

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

File I/O Streams in C Programming Language :

- In C all **input and output** is done with streams
- Stream is nothing but the **sequence of bytes of data**
- A sequence of bytes flowing into program is called **input stream**
- A sequence of bytes flowing out of the program is called **output stream**
- Use of Stream make I/O machine independent.

Basic File Operations

- Creating a new file

- Opening an existing file

- Reading from and writing information to a file

- Closing a file

CREATING A FILE

- **Working with file**
- While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.
- FILE *ptr;

Opening a file

- Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

- ptr=fopen("fileopen","mode") For Example: fopen("E:\\cprogram\program.txt","w");

Closing a File

- The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

- fclose(ptr); //ptr is the file pointer associated with file to be closed.

Reading and writing of a binary file.

- Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

- Function fwrite() takes four arguments, address of data to be written in disk, size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

- fwrite(address_data,size_data,numbers_data,pointer_to_file);

File Types

- 1) Ordinary Files or Simple File
- 2) Directory files
- 3) Special Files
- 4) FIFO Files:

FILE OPENING MODES

Mode	Description(TEXT File)	Mode	Description(Binary File)
r or rt	To read data from text file	rb	To read data from the binary file
w or wt	To write data into the text file.	wb	To write data into the binary file.
a or at	To write data into the text file at the end.	ab	To write data into the binary file at the end.
r+ or rt+	To read data from the text file.	rb+	To read / write data from / to the binary file.
w+ or wt+	To write data into the text file.	wb+	To write / read data into / from the binary file.
a+ or at+	To write data into the text file at the end or to read.	ab+	To write data into the binary file at the end or to read.

FILE INPUT AND OUTPUT FUNCTIONS

Formatted Input/Output

- These functions are used to read numbers, characters or string from file or write them to file in format as our requirement.

I. fprintf()

- This function is formatted output function which is used to write some integer, float, char or string to a file. Its syntax is

    fprintf(file_ptr_variable, "control string", list_variables);

II. fscanf()

- This function is formatted input function which is used to read some integer,float, char or string from a file. Its syntax is

    fscanf(file_ptr_variable, "control_string", &list_variables);

FILE STATUS FUNCTIONS

- stat, fstat, lstat - get file status
- These functions return information about a file. No permissions are required on the file itself, but-in the case of **stat**() and **lstat**() - execute (search) permission is required on all of the directories in *path* that lead to the file.
- **stat**() stats the file pointed to by *path* and fills in *buf*.
- **lstat**() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
- **fstat**() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

FILE POSITIONING FUNCTIONS

- The C library function **int fseek(FILE *stream, long int offset, int whence)** sets the file position of the **stream** to the given **offset**.

- **Declaration**

- Following is the declaration for fseek() function.

- int fseek(FILE *stream, long int offset, int whence) **Parameters**

- **stream** − This is the pointer to a FILE object that identifies the stream.

- **offset** − This is the number of bytes to offset from whence.

- **whence** − This is the position from where offset is added. It is specified by one of the following constants −

- Constant Description SEEK_SET Beginning of file SEEK_CUR Current position of the file pointer SEEK_END End of file

Command Line Argument

- Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. command line arguments are passed to **main()** method.

- **Syntax :**

- int main(int argc, char *argv[]) Here **argc** counts the number of arguments on the command line and **argv[ ]** is a pointer array which holds pointers of type char which points to the arguments passed to the program.