

What is a Computer?

A computer is an electronic device that takes input such as numbers, text, sound, image, animations, video, etc., processes it, and converts it into meaningful information that could be understood, presenting the changed input (processed input) as output. All numbers, text, sound, images, animations, and video used as input are called **data**, and all numbers, text, sound, images, animations, and video returned as output are called **information**.

- The data consists of numbers, text, sound, images, animations, and video.
- The process converts numbers, text, sound, images, animations, and video (data) into usable data, which is called information.
- The information consists of numbers, text, sound, images, animations, and video that has been converted by the process.
- The data is inserted using an **input device**.
- The central processing unit (CPU) converts data to information.
- The information is put on an **output device**.

A storage device is an apparatus for storing data and information. A basic computer consists of 4 components: an input device, a CPU, output devices, and memory.

- One of the most common input devices is the keyboard, used to enter text.

Computer Parts

Computers have two main parts that work together to form the computer system. These parts are:

I. Hardware part: A part of computer components that can be seen and touched. The hardware parts of a computer include processor (CPU), input devices, output devices and communicating devices.

- i) Central Processing Unit (CPU): This is the brain of the computer that makes the whole processes or tasks done by a computer. It is also called processor.
- ii) Input devices: those are part of the computer system that helps the computer to accept data to be

processed. Examples: keyboard, mouse, scanner, microphone.

- iii) Output devices: are devices (computer system parts) used to view the output after processed by the processor. Examples: monitor, printer, speaker.
- iv) Communicating devices: these are devices that facilitates the data transfer in between the different parts of the computer.

II. Software parts: General term used to describe a collection of computer programs and documentation that perform some tasks on the computer. It is the invisible part of a computer that makes the computer working by cooperating with the hardware parts. It also have two main parts.

1. System Software: Computer programs that directly communicate with computer hardware. It includes a combination of the following:
 ≡ Operating system: examples Windows XP, Windows 7, Windows 8, Ubuntu...
 ≡ Drivers: examples Network driver, function key driver, Sound driver...
 ≡ Programming Software: examples compilers (Code block, Dev, Borland, turbo) .
2. Application software: Allows end users to accomplish one or more specific (not directly computer development related) tasks. Typical applications include: Ms Office, database, Browser, paint, image editor, and any other software. Application software communicates with the computer via the system software. see the following diagram.

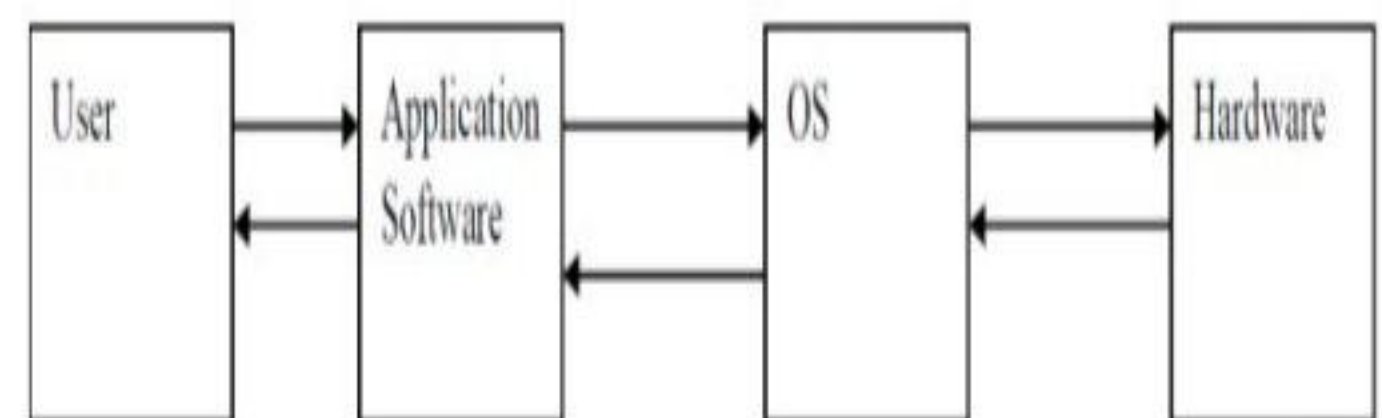


Fig 1. block diagram showing the interaction of AS with SS

Numeral systems

A writing method for expressing numbers is called a "numeral system". In the most common numeral system, we write numbers with combinations of 10 symbols {0,1,2,3,4,5,6,7,8,9}. These symbols are called digits, and numbers that are expressed using 10 digits are called "decimal" or "base-10" numbers. The other most common numeral systems are **binary**, **hexadecimal** and **octal**. The binary numeral system, or base-2 number system, represents numeric values using two symbols: 0 and 1. More

specifically, the usual base-2 system is a positional notation with a radix of 2. Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used internally by almost all modern computers.

Decimal Numeral System

In the first method discussed we write numbers with combinations of 10 symbols {0,1,2,3,4,5,6,7,8,9} called digits. Numbers that are expressed with 10 digits are called "base-10" numbers or "Decimal Numeral System". **For example:**

2 (one digit)

45 (two digit)

643 (three digit)

8785 (four digit)etc.

In Decimal Numeral Systems, the value of a digit is multiplied according to its placement in a numerical sequence: (base-number ^ 0,1,2,3,...), from right to left.

...	100,000	10,000	1,000	100	10	1
...	10^5	10^4	10^3	10^2	10^1	10^0
...	6	5	4	3	2	1
	Sixth digit	Fifth digit	Fourth digit	Third digit	Second digit	First digit

Value of digits in the "Decimal numeral system"

First digit = (base-number ^ 0): $10^0 = 1$

Second digit =(base-number ^ 1): $10^1 = 10$

Third digit =(base-number ^ 2): $10^2 = 100$

Fourth digit =(base-number ^ 3): $10^3 = 1000$

etc.

For example:

$20 = (2*10) + (0*1) = 20 + 0 = 20$

$456 = (4*100) + (5*10) + (6*1) = 400 + 50 + 6$

$84568 = (8*10000) + (4*1000) + (5*100) + (6*10) + (8*1) = 80000 + 4000 + 500 + 60 + 8$

Binary Numeral System

Numbers expressed with 2 symbols (0, 1) are called binary, or "base-2" numbers.

B denotes binary prefix.

For example:

1 (one-digit-read: 1)

10 (two-digit-read: 1, 0)

100 (three-digit-read: 1,0,0)

1101 (four-digit-read: 1, 1, 0, 1)

Examples:

$10101_2 = 10101_B = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 4 + 1 = 21$

$10111_2 = 10111_B = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 4 + 2 + 1 = 23$

$100011_2 = 100011_B = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 2 + 1 = 35$

In the Binary Numeral System, digits have a value specified, this value being equal with (base-number ^ 0,1,2,3,...): (right to left)

Converting Binary to Decimal

To convert binary to decimal, each digit is multiplied by the value of its position, and the results are added.

For example:

$10 = (1*2^1) + (0*2^0) = 1*2 + 0*1 = 2 + 0 = 2 \rightarrow 10$
(binary) =2 (decimal)

$101 = (1*2^2) + (0*2^1) + (1*2^0) = 1*4 + 0*2 + 1*1 = 4 + 0 + 1 = 5 \rightarrow 101 \text{ (binary)} = 5 \text{ (decimal)}$

$11001 = (1*2^4) + (1*2^3) + (0*2^2) + (0*2^1) + (1*2^0) = 1*16 + 1*8 + 0*4 + 0*2 + 1*1 = 16 + 8 + 0 + 0 + 1 = 25 \rightarrow 11001 \text{ (binary)} = 25 \text{ (decimal)}$

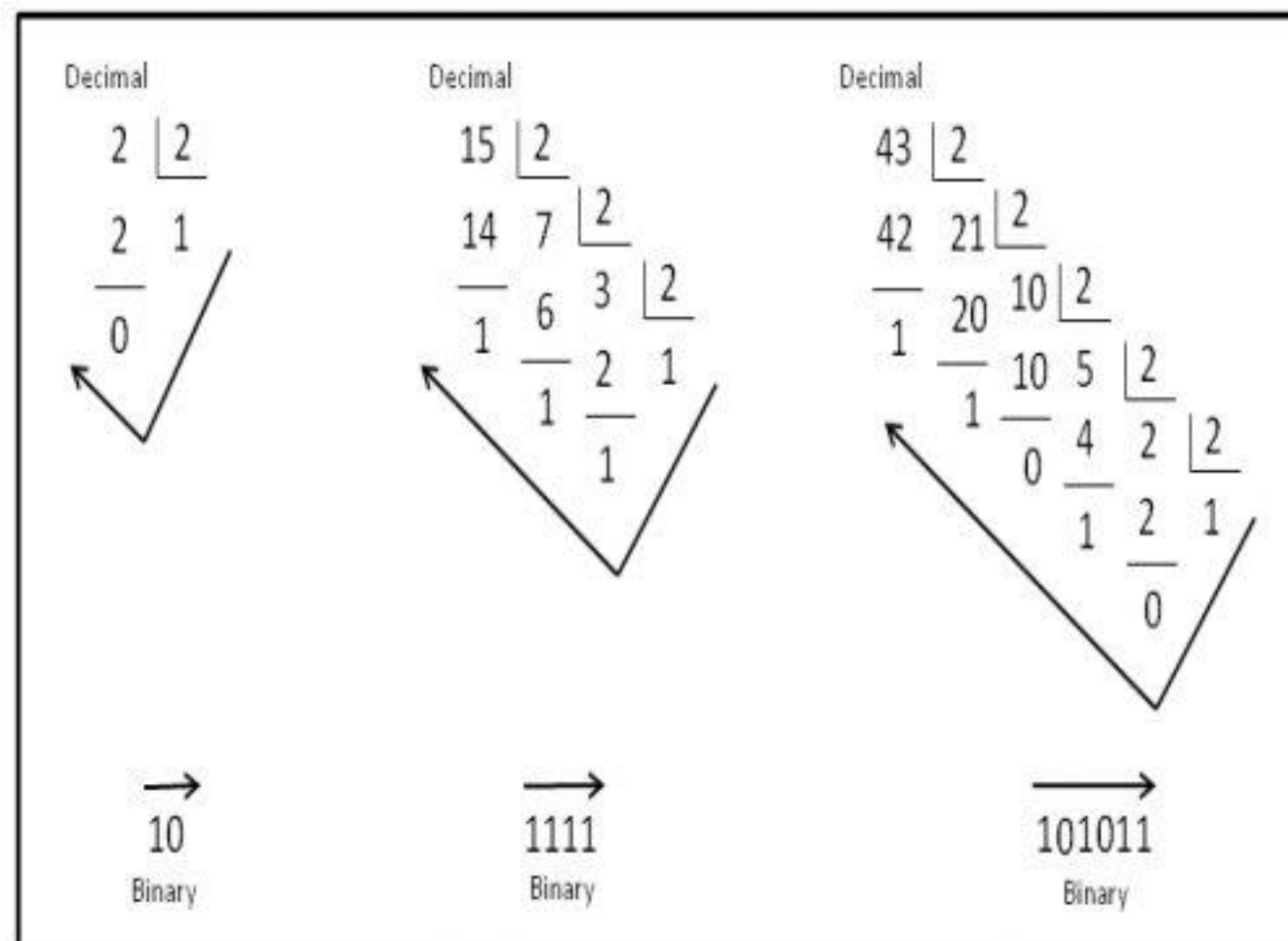
$111011 = (1*2^5) + (1*2^4) + (1*2^3) + (0*2^2) + (1*2^1) + (1*2^0) = 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1$

$= 32 + 16 + 8 + 0 + 2 + 1 = 59 \rightarrow 111011$ (binary) = 59 (decimal)

Converting Decimal to Binary

To convert decimal to binary

Divide the decimal number by 2



- If there IS a remainder the rightmost column will be a 1
- If there is NO remainder, the rightmost column will be a 0.

Then repeat the process, moving one column to the left each time until you have divided down to 1.

Example 1

$15/2 = 7$ remainder 1 (Binary number = ???1)

$7/2 = 3$ remainder 1 (Binary number = ??11)

$3/2 = 1$ remainder 1 (Binary number = ?111)

The final result will always be 1 in the leftmost column (Binary number = 1111)

Example 2 $74/2 = 37$ remainder 0 (Binary number = ??????0)

$37/2 = 18$ remainder 1 (Binary number = ?????10)

$18/2 = 9$ remainder 0 (Binary number = ???010)

$9/2 = 4$ remainder 1 (Binary number = ??1010)

$4/2 = 2$ remainder 0 (Binary number = ?01010)

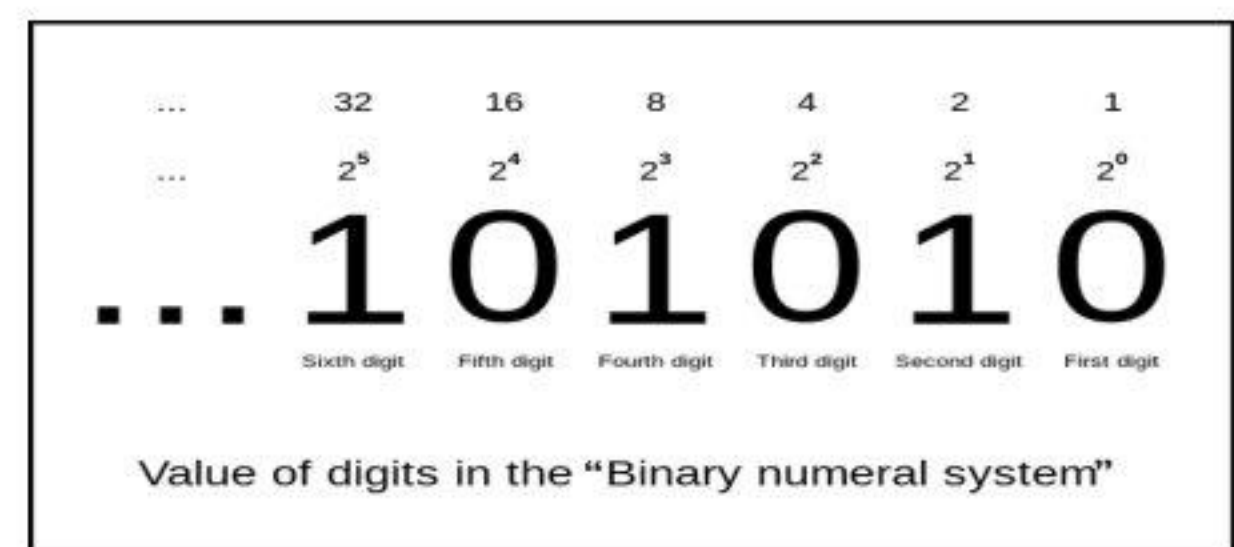
$2/2 = 1$ remainder 0 (Binary number = ?001010)

The final result will always be 1 in the leftmost column (Binary number = 1001010)

NB - Although I've put ? in at each stage, you won't know how many columns are needed until you complete the process.

For a shortcut to see how many columns are needed, find the largest factor of 2 that is smaller than the decimal number you started with, e.g.

Example 1: The largest factor less than 74 is 64, which is 2 to the power 6. As the furthest right column is 2 to the power 0, this means we need 7 columns.



First digit (base-number⁰): $2^0 = 1$

Second digit (base-number¹): $2^1 = 2$

Third digit (base-number²): $2^2 = 4$

Fourth digit (base-number³): $2^3 = 8$

etc.

Octal Numeral System - Base-8

Octal numbers uses digits from 0.-7.

Examples:

$$27_8 = 2 \times 8^1 + 7 \times 8^0 = 16 + 7 = 23$$

$$30_8 = 3 \times 8^1 + 0 \times 8^0 = 24$$

$$4307_8 = 4 \times 8^3 + 3 \times 8^2 + 0 \times 8^1 + 7 \times 8^0 = 2247$$

Octal to decimal conversion

Example: Convert 764_8 to decimal:

$$764_8 = 7 \times 8^2 + 6 \times 8^1 + 4 \times 8^0 = 448 + 48 + 4 = 500_{10}$$

For double-digit octal numbers this method amounts to multiplying the lead digit by 8 and adding the second digit to get the total.

Example: $65_8 = 6 \times 8 + 5 = 53_{10}$

Octal to binary conversion

To convert octal to binary, replace each octal digit by its binary representation.

Example: Convert 51_8 to binary:

$$\begin{aligned} 5_8 &= 101_2 \\ 1_8 &= 001_2 \end{aligned}$$

Therefore, $51_8 = 101\ 001_2$.

Binary to octal conversion

The process is the reverse of the previous algorithm. The binary digits are grouped by threes, starting from the least significant bit and proceeding to the left and to the right. Add leading zeroes (or trailing zeroes to the right of decimal point) to fill out the last group of three if necessary. Then replace each trio with the equivalent octal digit.

For instance, convert binary 1010111100 to octal:

001	010	111	100
1	2	7	4

Therefore, $1010111100_2 = 1274_8$.

Convert binary 11100.01001 to octal:

011	100	.	010	010
3	4	.	2	2

Therefore, $11100.01001_2 = 34.22_8$.

Shortcut method - Binary to Octal

Steps

- **Step 1** – Divide the binary digits into groups of three (starting from the right).
- **Step 2** – Convert each group of three binary digits to one octal digit.

Example

Binary Number – 10101_2

Calculating Octal Equivalent –

Step	Binary Number	Octal Number
------	---------------	--------------

Step 1	10101_2	$010\ 101$
Step 2	10101_2	$2_8\ 5_8$
Step 3	10101_2	25_8

Binary Number – $10101_2 =$ Octal Number – 25_8

Shortcut method - Octal to Binary

Steps

- **Step 1** – Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2** – Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example

Octal Number – 25_8

Calculating Binary Equivalent –

Step	Octal Number	Binary Number
Step 1	25_8	$2_{10}\ 5_{10}$
Step 2	25_8	$010_2\ 101_2$
Step 3	25_8	010101_2

Octal Number – $25_8 =$ Binary Number – 10101_2

Hexadecimal Numeral System

Numbers written with 16 symbols

$\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ are called "base-16" numbers. For example:

A (one digit)

B5 (two digit)

6C3 (three digit)

AF85 (four digit)

etc.

so:

A(hexadecimal)=10(decimal).

B(hexadecimal)=11(decimal).

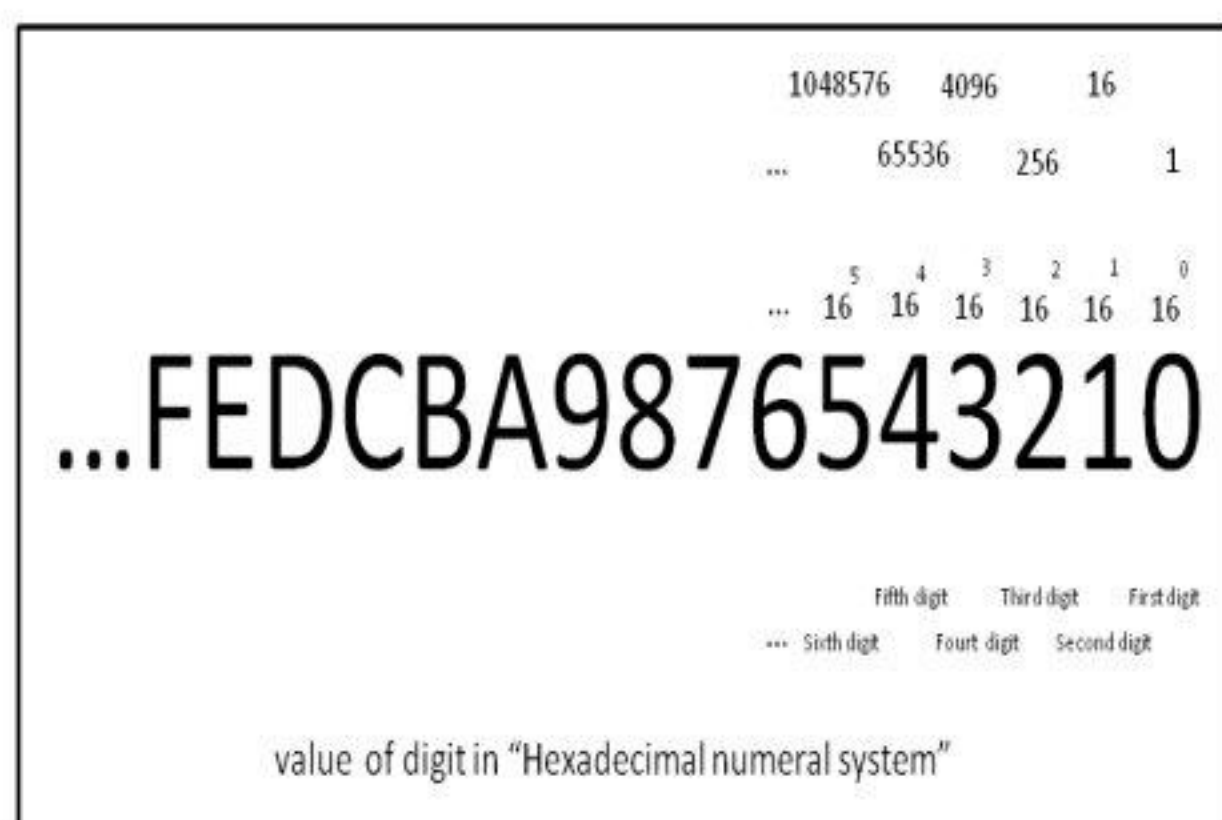
C(hexadecimal)=12(decimal).

D(hexadecimal)=13(decimal).

E(hexadecimal)=14(decimal).

F(hexadecimal)=15(decimal)

In the "Hexadecimal Numeral System", digits have a value specified, this value of digits are equal with (base-number^{0,1,2,3,...}):(right to left)



First digit (base-number ⁰): $16^0 = 1$

Second digit (base-number ¹): $16^1 = 16$

Third digit (base-number ²): $16^2 = 256$

fourth digit (base-number ³): $16^3 = 4096$ etc.

Converting Hexadecimal to Decimal

To convert hexadecimal to decimal, each digit is multiplied by the value of its position, and the results are added.

For example:

$A = (10 \times 16^0) = 10 \times 1 = 10 \rightarrow A(\text{hexadecimal}) = 10(\text{decimal})$

$B5 = (11 \times 16^1) + (5 \times 16^0) = 11 \times 16 + 5 \times 1 = 181 \rightarrow B5(\text{hexadecimal}) = 181(\text{decimal})$

$6C3 = (6 \times 16^2) + (12 \times 16^1) + (3 \times 16^0) = 6 \times 256 + 12 \times 16 + 3 \times 1 = 1536 + 192 + 3 = 1731 \rightarrow 6C3(\text{hexadecimal}) = 1731(\text{decimal})$

$AF85 = (10 \times 16^3) + (15 \times 16^2) + (8 \times 16^1) + (5 \times 16^0) = 10 \times 4096 + 15 \times 256 + 8 \times 16 + 5 \times 1 = 40960 + 3840 + 128 + 5 = 44933 \rightarrow AF85(\text{hexadecimal}) = 44933(\text{decimal})$

Converting Decimal to Hexadecimal

To convert decimal to hexadecimal

- Divide the decimal number by 16 - the remainder given is the last hexadecimal value.
- The quotient is then divided by 16 to get another remainder. Like the binary calculation the values are read right to left (first remainder value is the last in the hexadecimal number, then next to last, third to last, etc.)
- The process is terminated once a remainder of less than 16 is reached. This would give a quotient of 0 which is indivisible.
- Keep in mind that 10-15 are represented as single character "numbers" in the hexadecimal system. A=10, B=11, C=12, D=13, E=14, F=15 - remainders must reflect their appropriate hexadecimal value.

Examples

- Decimal 15

o 15/16 remainder is 15 (>16 so process terminates) so the "number" value is F

- Decimal 16

o 16/16 remainder is 0 [hex ?0]
o The quotient of 1 is then divided - 1/16 which leaves a remainder of 1 (giving a quotient of 0 so process terminates) [hex 10]

- Decimal 45

o 45/16 – remainder 13 [hex ?D]
o Quotient 2 | 2/16 – remainder 2 [hex 2D]

- Decimal 47825

o 47825/16 - remainder 1 [hex ???1]
o Quotient 2989 | 2989/16 – remainder 13 [hex ??D1]
o Quotient 186 | 186/16 – remainder 10 [hex ?AD1]
o Quotient 11 | 11/16 – remainder 11 [hex BAD1]

	Decimal - 2	Decimal - 16	Decimal - 973	Decimal - 3057609
A = 10				
B = 11	2 16	16 16	973 16	3057609 16
C = 12	0 0	16 1 16	960 60 16	3057600 191100 16
D = 13	2	0 0 0	13 (D) 48 3 16	9 191088 11943 16
E = 14		1	12 (C) 0 0	12 (C) 11936 746 16
F = 15			3	7 736 46 16
				10 (A) 32 2 16
				14 (E) 0 0
				2
	Hexadecimal	Hexadecimal	Hexadecimal	Hexadecimal
	2	10	3CD	2EA7C9

- **Step 1** – Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2** – Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Shortcut method - Binary to Hexadecimal

Steps

- **Step 1** – Divide the binary digits into groups of four (starting from the right).
- **Step 2** – Convert each group of four binary digits to one hexadecimal symbol.

Example

Binary Number – 10101_2

Calculating hexadecimal Equivalent –

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101
Step 2	10101_2	$1_{10} 5_{10}$
Step 3	10101_2	15_{16}

Binary Number – 10101_2 = Hexadecimal Number – 15_{16}

Shortcut method - Hexadecimal to Binary

Steps

Example

Hexadecimal Number – 15_{16}

Calculating Binary Equivalent –

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	$1_{10} 5_{10}$
Step 2	15_{16}	$0001_2 0101_2$
Step 3	15_{16}	00010101_2

Hexadecimal Number – 15_{16} = Binary Number – 10101_2

Decimal Base-10	Binary Base-2	Octal Base-8	Hexadecimal Base-16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1A
27	11011	33	1B
28	11100	34	1C
29	11101	35	1D
30	11110	36	1E
31	11111	37	1F
32	100000	40	20

Numeral systems conversion table

Algorithm and flowchart

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication. The term programming language usually refers to high-level languages, such as BASIC, C, C++, JAVA, FORTRAN, , and Pascal. Each language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Three level of programming language:

1. Machine language- binary representation of the instructions that computer hardware can perform. it is a language that is directly understood by the computer. i.e. computers only understand binary numbers (0 and 1).

2. Assembly language- is a low level programming language in which a mnemonic is used to represent each of the machine language instruction for a specific computer. an assembler is used to change the assembly to machine language so that the computer can understand it easily. 8085,8086 and 8051 programs are examples of assembly languages.

n.b. this programming language is easy to understand but needs advanced assemblers.

3. High level language- is more understandable and closer to standard notations than assembly language. a compiler is used to change the high level language to machine language. e.g. C,C++, Java and C# are high level programming languages. the following two tables helps you to understand the difference between the computer languages mentioned above.

Algorithm, pseudo code and flowchart

A. Algorithm: Is a procedure for solving a problem in terms of the actions to be executed and the order in which those actions are to be executed. An algorithm is merely the sequence of steps taken to solve a problem.

B. Pseudo code: is an artificial and informal language that helps programmers develop algorithms. In this course we will enforce three standards for good pseudo code :-

1. Number each instruction. This is to enforce the notion of an ordered sequence of ... operations. Furthermore we introduce a dot notation (e.g. 3.1 come after 3 but before 4) to number subordinate operations for conditional and iterative operations

2. Each instruction should be unambiguous (that is the computing agent, in this case the reader, is capable of carrying out the instruction) and effectively computable (doable).

3. Completeness.

C. Flow Chart: Is a representation, primarily through the use of symbols, of the sequence of activities in a system (process, operation, function...). it should be interpretable by a professional. The following symbols with their names are used in the building of systems using flowchart.

A typical programming task can be divided into two phases:

- Problem solving phase

- produce an ordered sequence of steps that describe solution of problem
- this sequence of steps is called an algorithm
- Implementation phase
- implement the program in some programming language

Steps in Problem Solving

- First produce a general algorithm (one can use pseudocode)
- Refine the algorithm successively to get step by step detailed algorithm that is very close to a computer language.
- Pseudocode is an artificial and informal language that helps programmers develop algorithms.
- Pseudocode is very similar to everyday English.
- Finally draw the Flow chart and solve the problem (write program)

Pseudocode & Algorithm

- Example 1: Write a pseudocode and an algorithm to convert the length in meter to centimeter.

Solution for Example 1

Pseudocode:

- Input the length in meter
- Calculate the length in cm by multiplying length in meter with 100
- Print length in cm.

Example 2

Algorithm

- Step 1: Input L_m • Step 2: $L_{cm} \leftarrow L_m \times 100$
- Step 3: Print L_cm

The Flowchart

- A schematic representation of a sequence of operations, as in a manufacturing process or computer program. It is a graphic representation of how a process works, showing, at a minimum, the sequence of steps. A flowchart consists of a sequence of instructions linked together by arrows to show the order in which the instructions must be carried out.

Each instruction is put into a box. The boxes are different shapes depending upon what the instruction is.

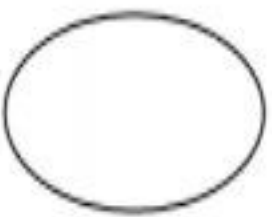
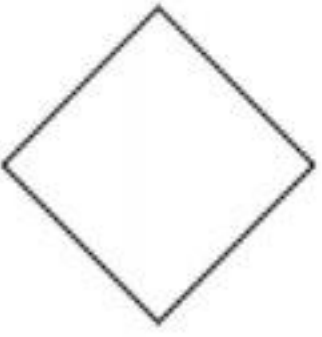
- Different symbols are used to draw each type of flowchart.

A Flowchart

- shows logic of an algorithm
- emphasizes individual steps and their interconnections
- e.g. control flow from one action to the next

Flowchart Symbols

Basic symbols to draw flowchart

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

Example 2

- Write the pseudocode, algorithm and draw a flowchart to convert the length in Km to meter.

Pseudocode:

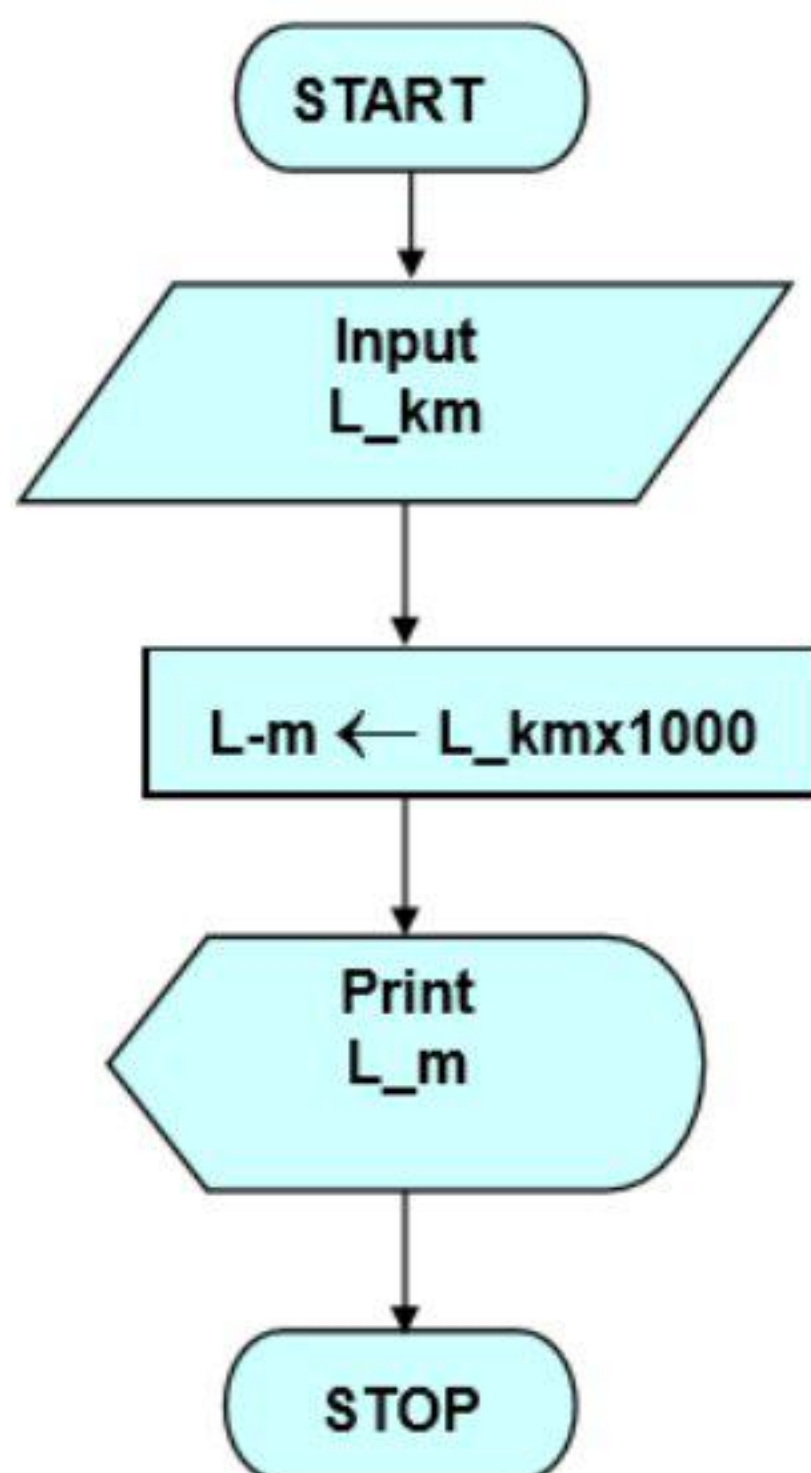
- Input the length in Km (L_Km)
- Calculate the length in meter (L_m) by multiplying L with 1000
- Print length in m (L_m)

Example 2

Algorithm

- Step 1: Input L_km
- Step 2: $L_m \leftarrow L_{km} \times 1000$
- Step 3: Print L_m

Flowchart



Example 3

- Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.

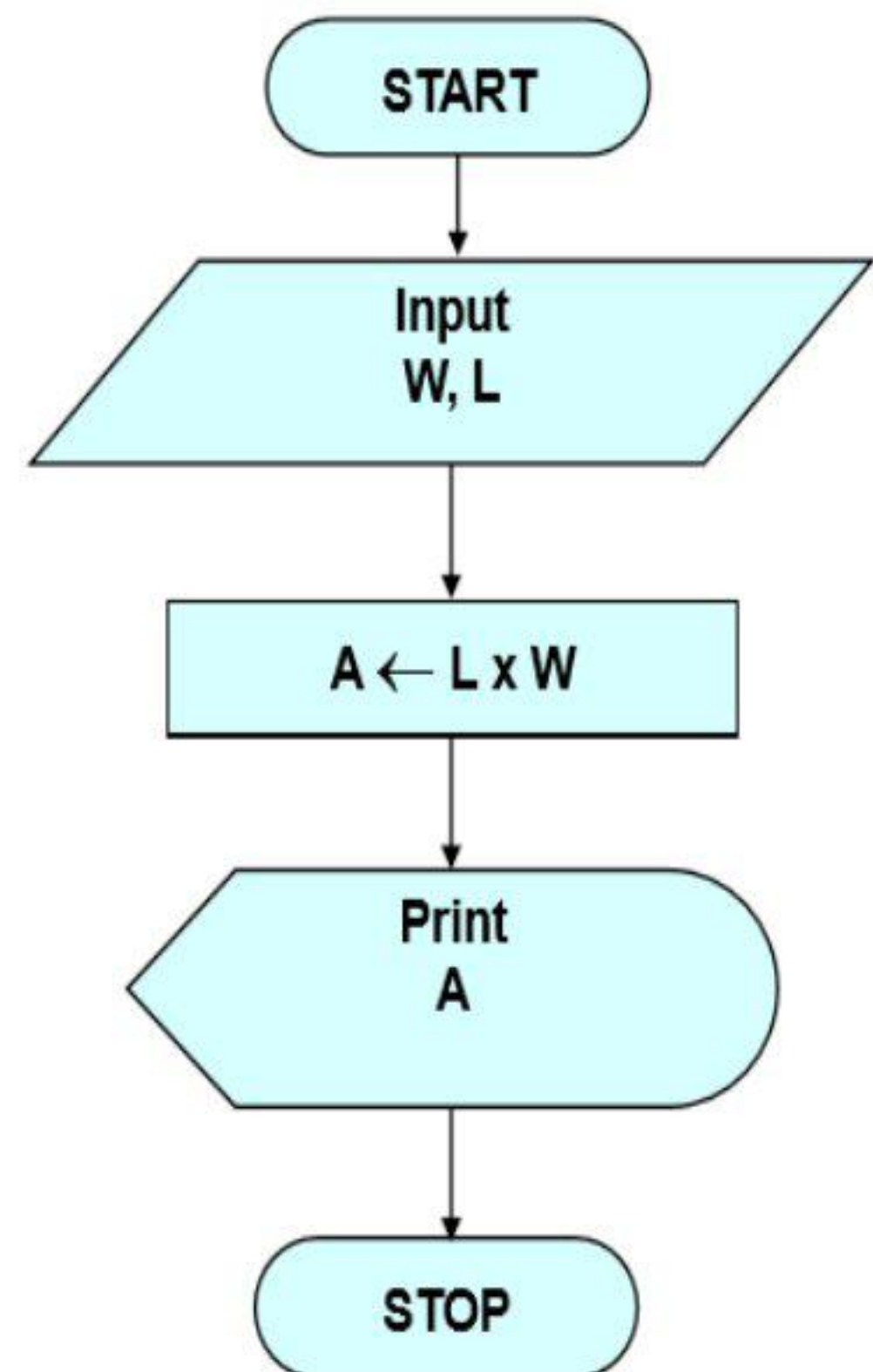
Pseudocode

- Input the width (W) and Length (L) of a rectangle
- Calculate the area (A) by multiplying L with W
- Print A

Example 3

Algorithm

- Step 1: Input W,L
- Step 2: $A \leftarrow L \times W$
- Step 3: Print A



13

Example 4

- Write an algorithm and draw a flowchart that will calculate the roots of a quadratic equation $ax^2+bx+c=0$

- Hint: $d = \sqrt{b^2-4ac}$, and the roots are:

$$x_1 = \frac{-b + d}{2a} \quad \text{and} \quad x_2 = \frac{-b - d}{2a}$$

Example 4

Pseudocode:

- Input the coefficients (a, b, c) of the quadratic equation
- Calculate d
- Calculate x1
- Calculate x2
- Print x1 and x2

Algorithm:

- Step 1: Input a, b, c

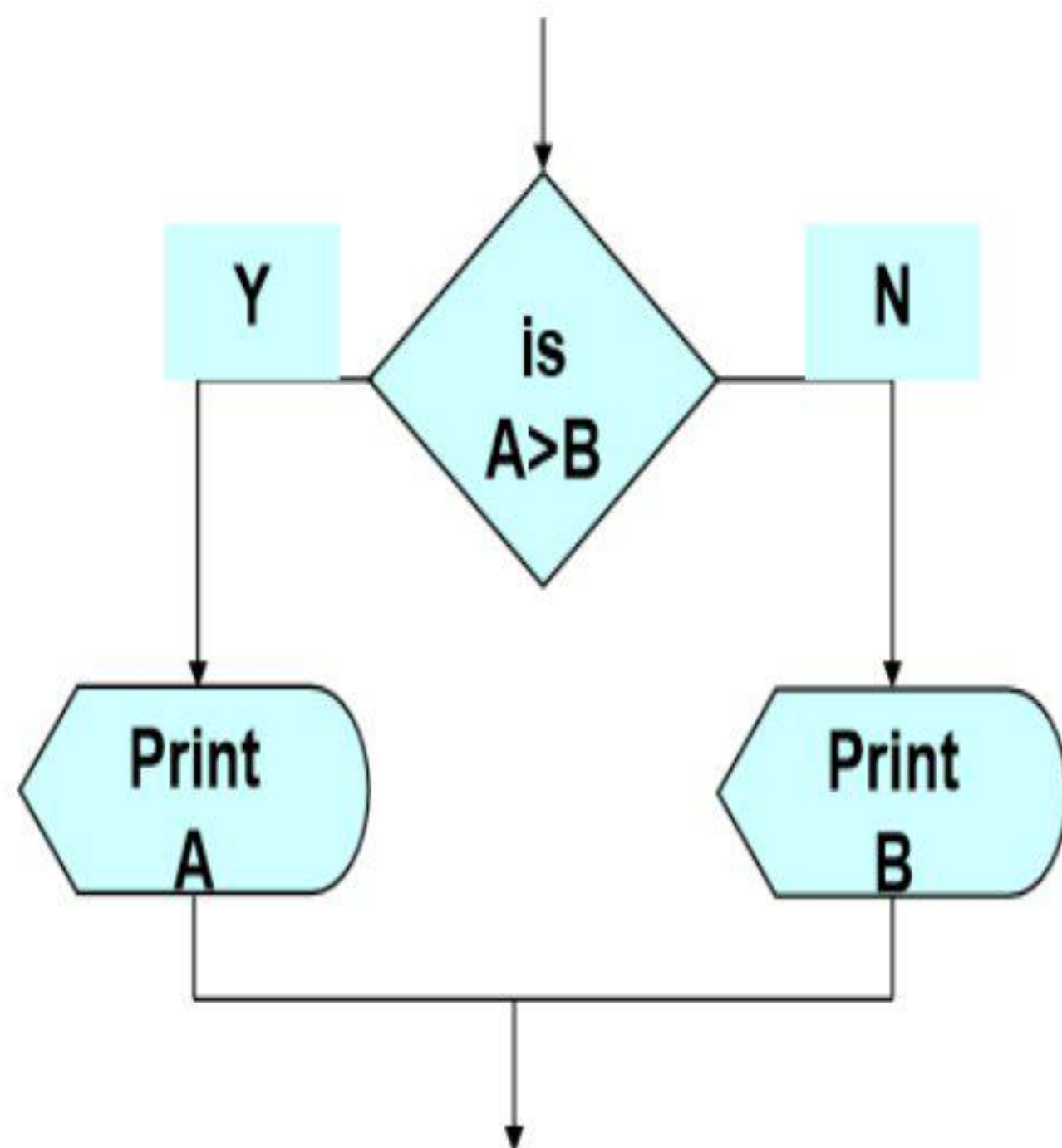
- Step 2: $d \leftarrow \sqrt{b^2 - 4ac}$
- Step 3: $x_1 \leftarrow \frac{-b + d}{2a}$
- Step 4: $x_2 \leftarrow \frac{-b - d}{2a}$
- Step 5: Print x_1, x_2

Decision Structures

- The expression $A > B$ is a logical expression
- it describes a condition we want to test
- if $A > B$ is true (if A is greater than B) we take the action on left
 - print the value of A
- if $A > B$ is false (if A is not greater than B) we take the action on right
 - print the value of B

Decision Structures

- Here is the decision structure



Example IF-THEN-ELSE STRUCTURE

- The structure is as follows If condition then true alternative else false alternative endif

IF-THEN-ELSE STRUCTURE

If $A > B$ then print A else print B endif

Relational Operators

Relational Operators	
Operator	Description
$>$	Greater than
$<$	Less than
$=$	Equal to
$\geq$	Greater than or equal to
$\leq$	Less than or equal to
$\neq$	Not equal to

Example 5 • Write an algorithm that reads two values, determines the largest value and prints the largest value with an identifying message.

ALGORITHM

Step 1: Input VALUE1, VALUE2

Step 2: if (VALUE1 > VALUE2) then

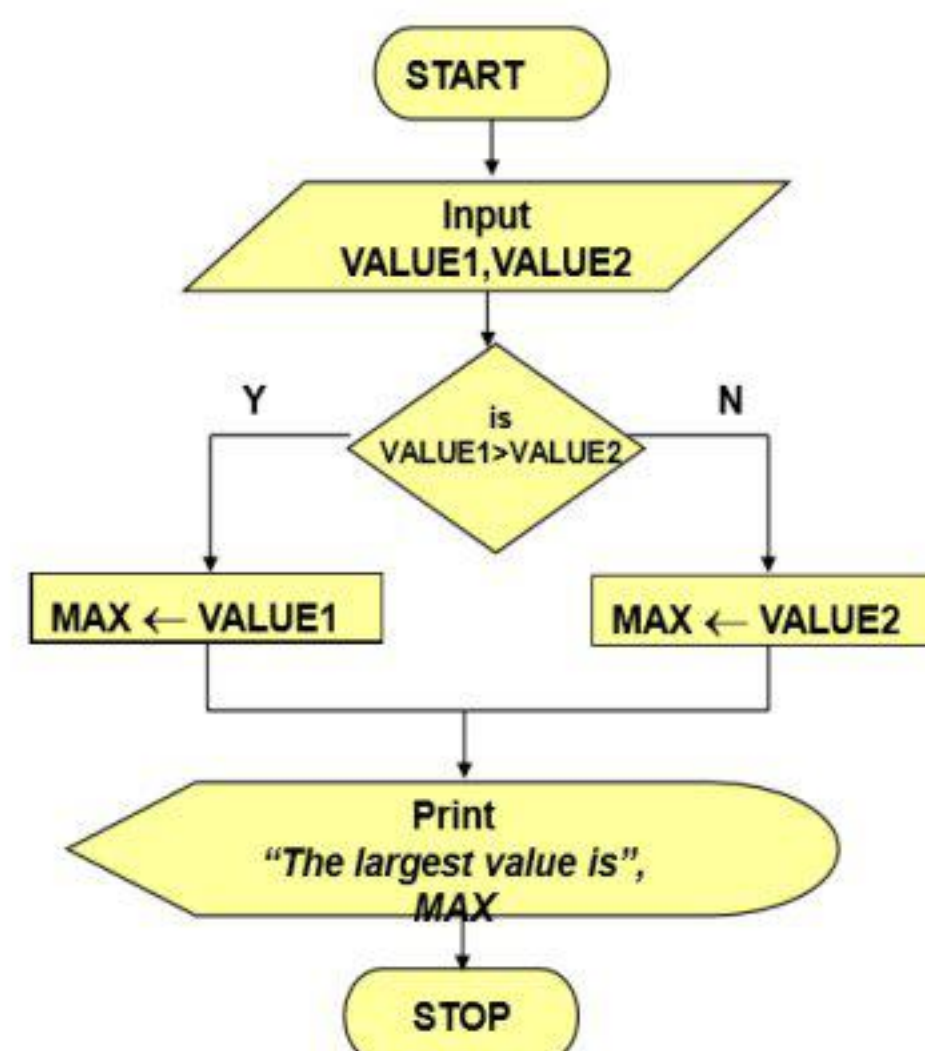
MAX ← VALUE1

else

MAX ← VALUE2

End if

Step 3: Print "The largest value is", MAX



NESTED IFS

- Nested ifs refers to many alternatives for making decisions

- One of the alternatives within an

IF–THEN– ELSE statement

–may involve further IF–THEN–ELSE statement

Exercise 1

- Write the pseudocode, algorithm and draw the flowchart that reads three numbers and prints the value of the largest number.

Answer

Algorithm:

Step 1: Input N1, N2, N3 Step

2: if (N1>N2) then

if (N1>N3) then

MAX N1 [N1>N2, N1>N3]

else

MAX N3 [N3>N1>N2]

endif

else if

(N2>N3) then MAX N2 [N2>N1, N2>N3]

Else

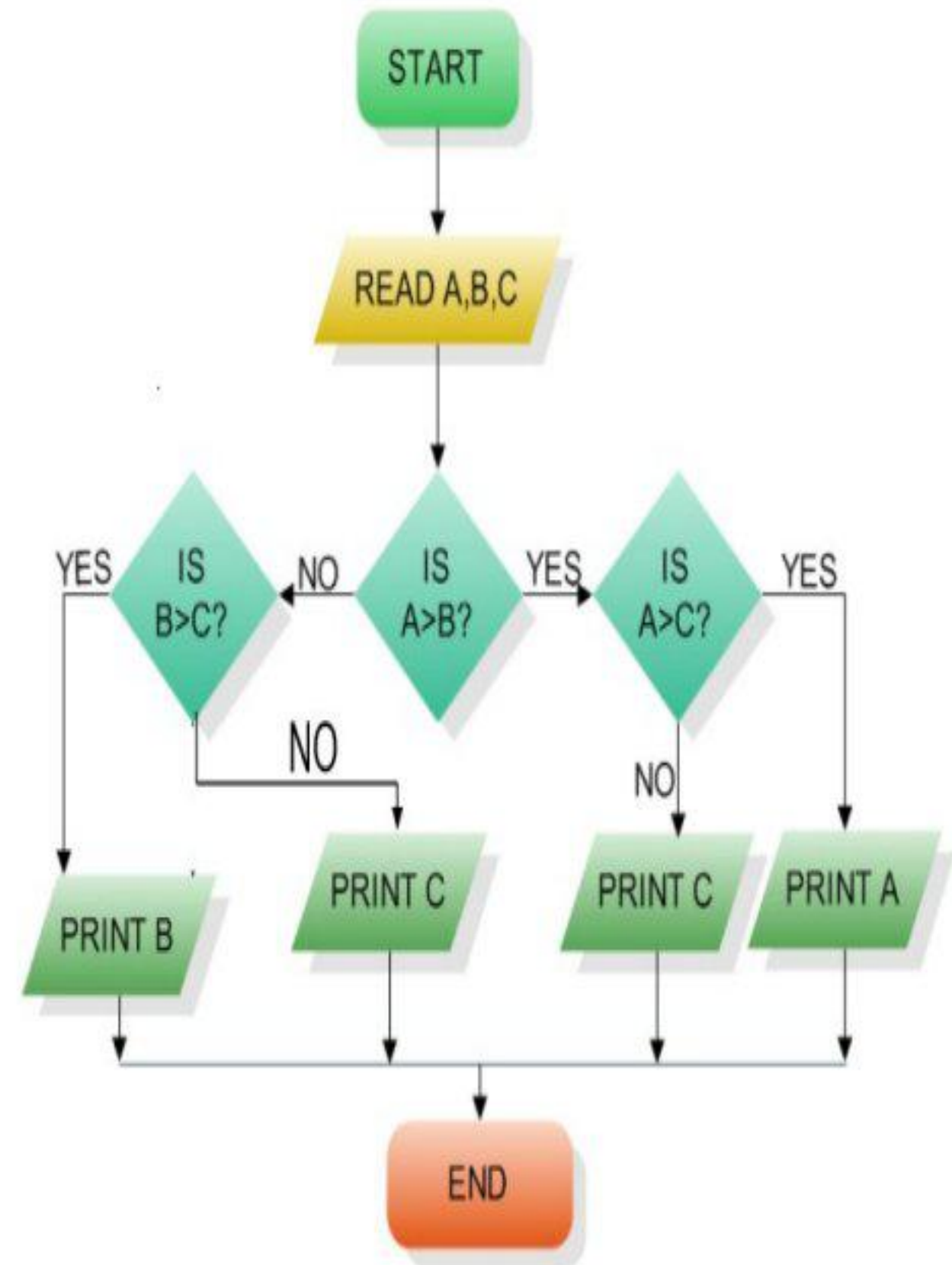
MAX← N3 [N3>N2>N1]

endif

endif

Step 3: Print “The largest number is”, MAX

Flowchart for the exercise



Chapter 2

C++ programming basics

Introduction

A computer program is a set of instructions that a programmer writes to tell a computer how to carry out a certain task. The instructions, however, must be in a language that the computer understands. Computers understand only binary languages (0's & 1's). This is a low level language and very hard to program in. So, high level languages such as C++ were discovered to make the programming task easier.

Basic C++ Program Structure

A single C++ program may include the following main parts.

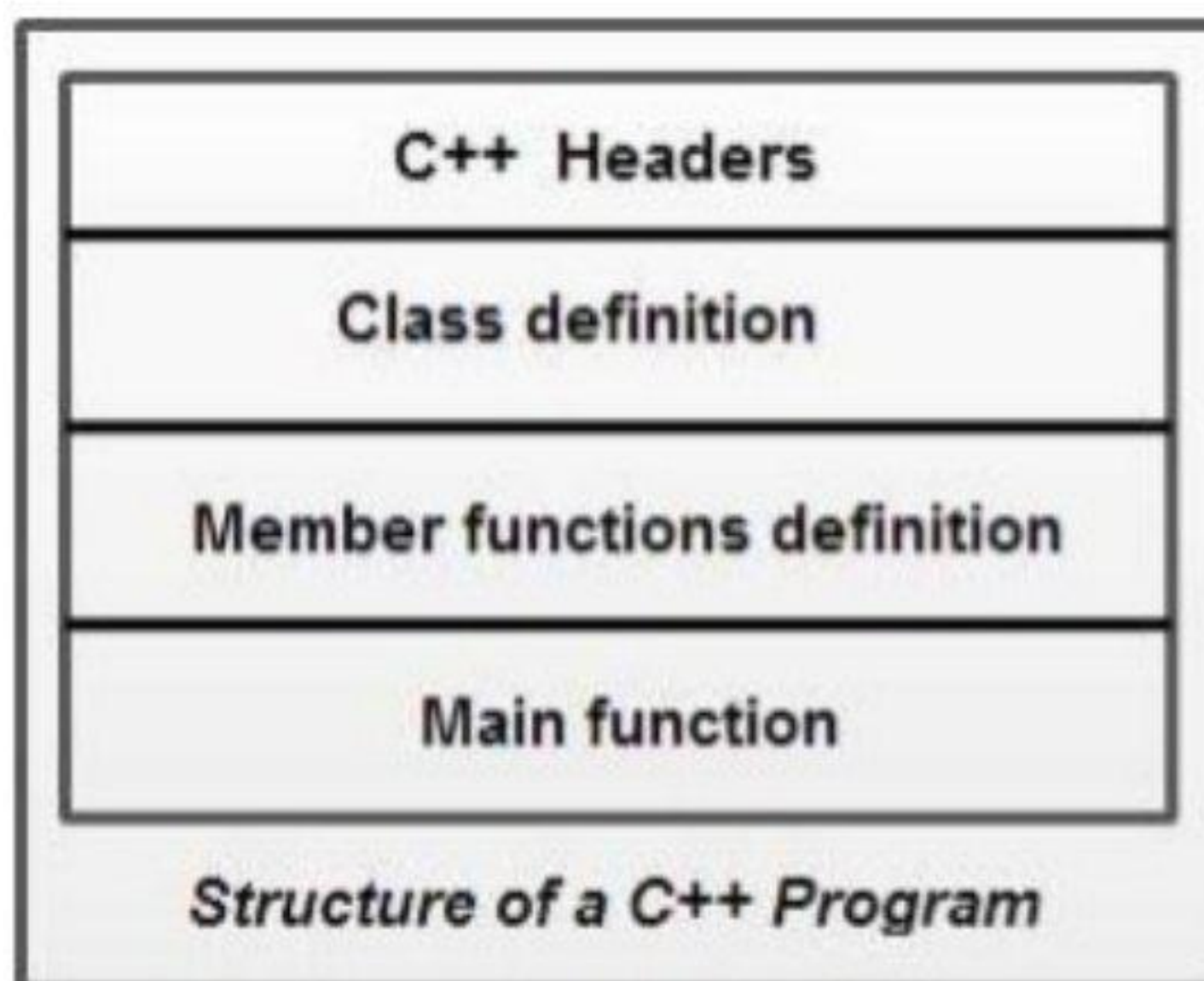


fig 2.1) the structure of C++ program

However, it doesn't mean that every C++ program should have this format. There might be a simple C++ program that doesn't include classes and member functions. Here is the simple structure of C++ program that every program should include.

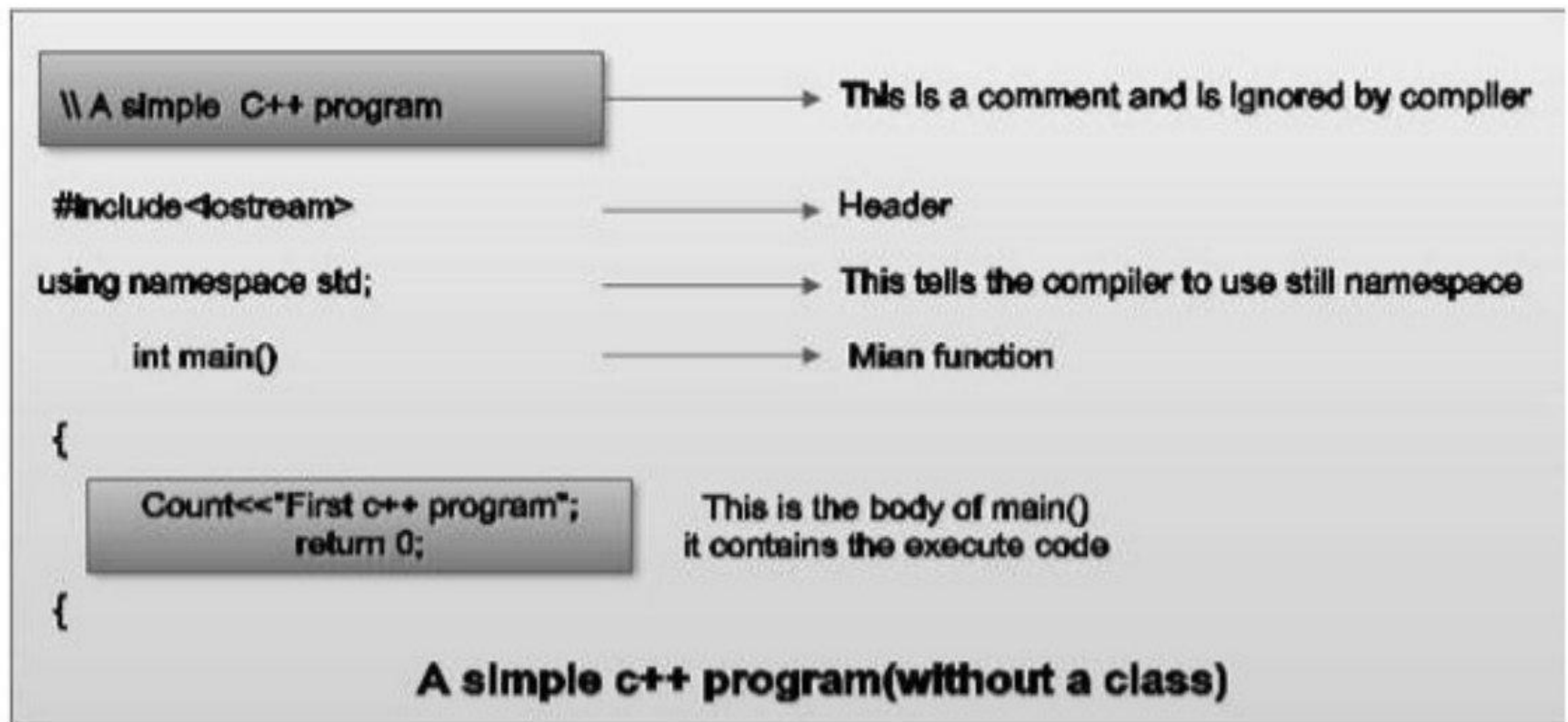


fig 2.2) simple C++ program structure

When we write this code in a Dev compiler it looks like shown below.

```

1  #include<iostream>
2  using namespace std;
3  int main ()
4  {
5
6
7
8      //Program statements;
9      //many program statements here
10
11     return 0;
12 }
13

```

Explaining the main parts of C++ program

- **Comments:**

Comments are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and hence, do not increase the size of a file. C++ supports two comment styles: single line comment and multiline comment.

- Single line comments are used to define line-by-line descriptions. Double slash (//) is used to represent single line comments. To understand the concept of single line

comment, consider this statement. `// this paragraph describes the advantage of single line comment (me).`

- Multiline comments are used to define multiple lines descriptions and are represented as `/* .... */`. For example, consider this statement.

`/* this paragraph uses to show An example to demonstrate`

`multiline comment. it is used if the lines of comments are more than one like this*/`

Generally, multiline comments are not used in C++ as they require more space on the line. However, they are useful within the program statements where single line comments cannot be used. For example, consider this statement.

`for(int i = 0; i<10; //loop runs 10 times i++) // this is error! it ignores all after comment`

Compiler ignores everything written after the single line comment and hence, an error occurs. Therefore, in this case multiline comments are used. For example, consider this statement.

`for(int i = 0; i<10; /*loop runs 10 times */ i++) // no error here. don't worry u it will be clear
//after you know the format of for loop.`

Headers:

Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc.,

that are already defined in the standard C++ library. In order to use such predefined elements in a program, an appropriate header must be included in the program.

The standard headers contain the information like prototype, definition and return type of library functions, data type of constants, etc.

As a result, programmers do not need to explicitly declare (or define) the predefined programming elements.

Standard headers are specified in a program through the preprocessor directive `#include`. In Figure 2.2, the `iostream` header is used.

When the compiler processes the instruction `#include<iostream>`, it includes the contents of `iostream` in the program.

This enables the programmer to use standard input, output and error facilities that are provided only through the standard streams defined in `<iostream>`.

These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow.

The standard streams defined in <iostream> are listed here.

- **cin** (pronounced "see in") : It is the standard input stream that is associated with the standard input device (keyboard) and is used to take the input from users.
- **cout** (pronounced "see out") : It is the standard output stream that is associated with the standard output device (monitor) and is used to display the output to users.

Namespace: Since its creation, C++ has gone through many changes by the C++ Standards Committee. One of the new features added to this language is namespace.

A namespace permits grouping of various entities like classes, objects, functions and various C++ tokens, etc., under a single name.

Different users can create separate namespaces and thus can use similar names of the entities. This avoids compile-time error that may exist due to identical-name conflicts.

The C++ Standards Committee has rearranged the entities of the standard library under a namespace called std. In Figure 2.2 above, the statement using namespace std informs the compiler to include all the entities present in the namespace std.

As soon as the new-style header is included, its contents are included in the std namespace. Thus, all the modern C++ compilers support these statements.

#include<iostream>

using namespace std;

However, some old compilers may not support these statements. In that case, the statements are replaced by this single statement.

#include<iostream.h>

➤ Main Function:

The **main ()** is a startup function that starts the execution of a c++ program. All C++ statements that need to be executed are written within **main ()**. The compiler executes all the instructions written within the opening and closing curly braces ' {} ' that enclose the body of **main ()**. Once all the instructions in **main ()** are executed, the control passes out of **main ()**, terminating the entire program and returning a value to the operating system.

By default, **main ()** in C++ returns an int value to the operating system. Therefore, **main ()** should end with the **return 0** statement. *A return value zero indicates success and a non-zero value indicates failure or error.*

Identifiers

Identifiers can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier (variable). The identifier is a sequence of characters taken from C++ character set. The rule for the formation of an identifier are:

- An identifier can consist of alphabets, digits and/or underscores.
- It must not start with a digit and any character without an alphabet.
- Identifiers must not be keywords (reserved words).
- Identifiers are case sensitive. i.e. Q and q are different.

Keywords (Reserved words): are words or statements that are assigned for some function in the system, so that we can't use them for identifiers. The following are the most known reserved words in C++.

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum
extern	inline	int	float	for
friend	goto	if	long	new
operator	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	Try
typedef	union	unsigned	virtual	void
volatile	while			

Character constants

A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character. The following table gives a listing of common escape sequences.

Escape Sequence	Non graphic Character
\a	Bell (beep)
\n	Newline
\r	Carriage Return
\t	Horizontal tab
\0	Null Character

Punctuators

The following characters are used as punctuators in C++.

Brackets []	Opening multidimensional and closing array bracket subscript. indicate single and
Parentheses ()	Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc.
Braces { }	Opening and closing braces indicate the start and end of a compound statement.
Comma ,	It is used as a separator in a function argument list.
Semicolon ;	It is used as a statement terminator.
Colon :	It indicates a labeled statement or conditional operator symbol.
Asterisk *	It is used in pointer declaration or as multiplication operator.
Equal sign =	It is used as an assignment operator.
Pound sign #	It is used as pre-processor directive.

Data Types:

C++ a large number of data types. these data types describes what kind of data is to be accepted, processed and stored in memory. The most common build in types are int, float and char. however there are other different types that C++ can use them. the most common are here in the table below.

Type	Byte Range		Description
int	9	-32768 to +32767	Small whole number
long int	4	-2147483648 to +2147483647	Large whole number
float	4	3.4x10 ⁻³⁸ to 3.4x10 ⁺³⁸	Small real number
double	8	1.7x10 ⁻³⁰⁸ to 1.7x10 ⁺³⁰⁸	Large real number
long double	10	3.4x10 ⁻⁴⁹³² to 3.4x10 ⁺⁴⁹³²	Very Large real number
char	1	0 to 255	A Single Character

Variables (identifiers)

It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

To understand more clearly we should study the following statements:

```
Total = 20.00;
```

In this statement a value 20.00 has been stored in a memory location Total.

Declaration of a variable

Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.

```
float Total;
```

You can declare more than one variable of same type in a single single statement

```
int x,y;
```

Initialization of variable

When we declare a variable it's default value is undetermined. We can declare a variable with some initial value.

```
int a = 20;
```

Input/ Output Operation in C++:

Computers can accept, process, store and print outputs. This is possible in C++ programming by using the input and output keywords and other following operators. The following stream objects are used to accept inputs and print our outputs to output devices.

cin console input and **cout** console output

cout is used in conjunction with << operator, known as insertion or put to operator.

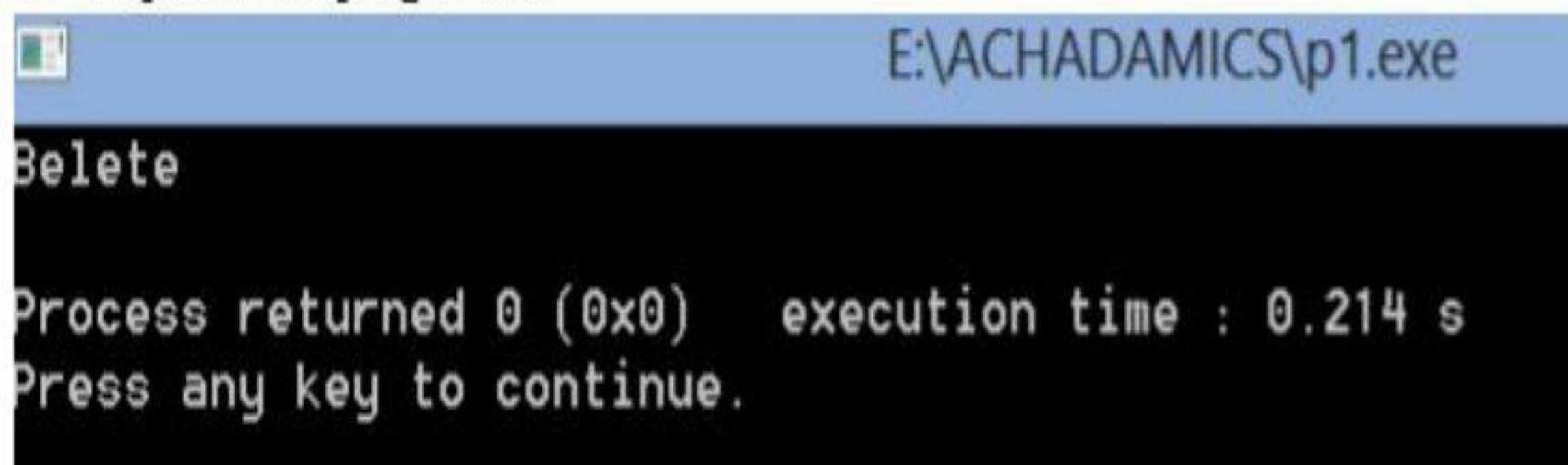
cin is used in conjunction with >> operator, known as extraction or get from operator.

Now let's start to write some simple programs using the compiler to see some outputs and accepting inputs from keyboard.

Example 1: write a C++ program that prints your name to the screen. let me write this program with name "Belete " and try by replacing your name.

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      cout<<"Belete"<<endl; // the endl in this line tells this line will be printed in new line
6      return 0; // finishes the program
7  }
```

The output of this program is:



```
E:\ACHADAMICS\p1.exe
Belete
Process returned 0 (0x0)   execution time : 0.214 s
Press any key to continue.
```

Now let's see how we can accept any number from keyboard. We use the keyword **cin** to accept data from keyboard. but before accepting data a memory should be reserved for that data and hence should be declared first. when declaring a variable it must first identify its data type from the above data types and must have a valid identifier to hold the memory for the data temporarily.

The syntax is: *data type* identifier; // we can also initialize the identifier

Example of declaration **int x**; // we can also say **int x=0**; which initializes the value of x to zero

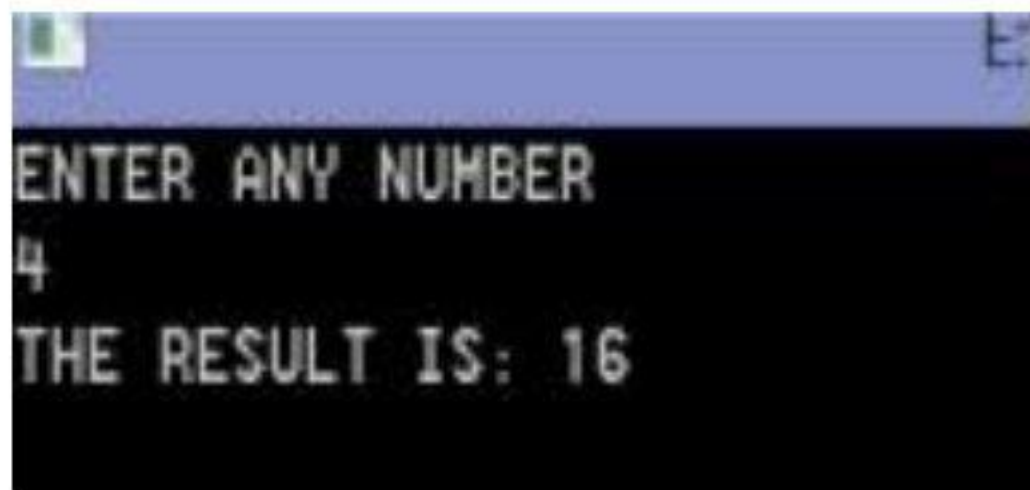
Exercise2: what is the output of the program below, what is the problem with it and modify it.

```
1      #include<iostream>
2      using namespace std;
3      int main()
4      {
5          cout<<"trfie";
6          cout<<"shashie";
7          cout<<"eke";
8          return 0;
9      }
```

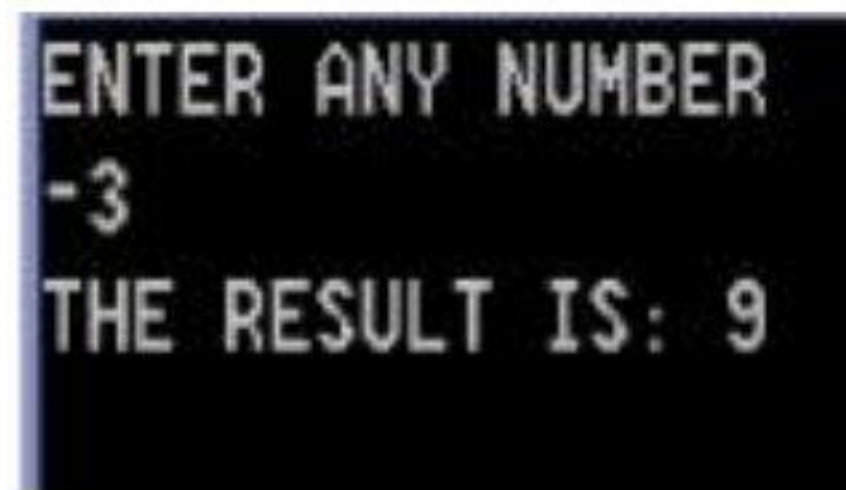
Example 3: Write a C++ program that accepts an integer from keyboard and prints its square to the screen.

```
1      #include<iostream>
2      using namespace std;
3      int main()
4      {
5          int x;
6          cout<<"ENTER ANY NUMBER "<<endl;
7          cin>>x;
8          cout<<"THE RESULT IS: ";
9          cout<<x*x<<endl;
10         return 0;
11     }
```

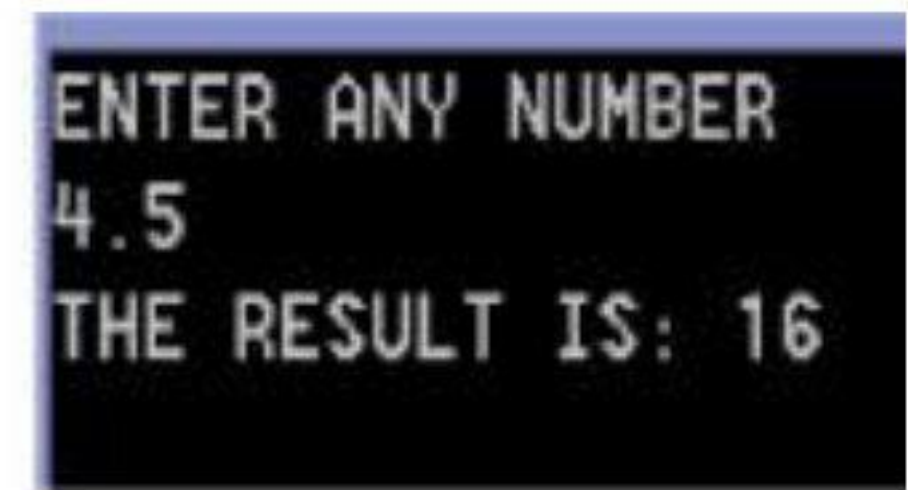
Let's see outputs for some inputs from keyboards. Let's insert 4, -3 and 4.5 from keyboard and see the output.



```
ENTER ANY NUMBER
4
THE RESULT IS: 16
```



```
ENTER ANY NUMBER
-3
THE RESULT IS: 9
```



```
ENTER ANY NUMBER
4.5
THE RESULT IS: 16
```

when we insert 4.5 we expect an output of $4.5 \times 4.5 = 20.25$. but the output displays 16 = 4×4 . This is because x is declared as integer data type so that it doesn't have memory to store floating point decimal.

Exersise: Write a C++ program that can accept two numbers and provide their sum to the screen.

Calculating in C++

As we calculate in mathematics, physics and other courses, we can also calculate in C++. we can add, multiply, divide and subtract numbers using available operators. operators can be Arithmetic operators, logical operators, relational operators, unary operator, compound operators, increment/decrement operators

Arithmetical operators

Arithmetical operators +, *, /, and % are used to perform an arithmetic (numeric) operation.

You can use the operators +, -, *, and / with both integral and floating-point data types.

Modulus or remainder % operator is used only with the integral data type.

Operators that have two operands are called binary⁷ operators.

Relational operators

The relational operators are used to test the relation between two values. All relational operators are binary⁷ operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

Relational Operators	Meaning
<	Less than
<=	Less than or equal to
=	Equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal to

Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are

Operators	Meaning
	OR
&&	AND
!	NOT

Unary operators:

C++ provides two unary operators for which only one variable is required.
For Example

```
a = -50; a = + 50;
```

Here plus sign (+) and minus sign (-) are unary because they are not used between two variables

Assignment operator

The assignment operator is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. For example: **m = 5;**

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

```
x = y = z = 32;
```

This code stores the value 32 in each of the three variables x, y, and z.

In addition to standard assignment operator shown above, C++ also supports compound assignment operators.

Compound Assignment Operators

Operator	Example	Equivalent to
+=	A += 2	A = A + 2
-=	A -= 2	A = A - 2
%=	A %= 2	A = A % 2
/=	A /= 2	A = A / 2
*=	A *= 2	A = A * 2

Increment and Decrement Operators

C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is:

Pre-increment: ++variable Post-increment: variable++

The syntax of the decrement operator is:

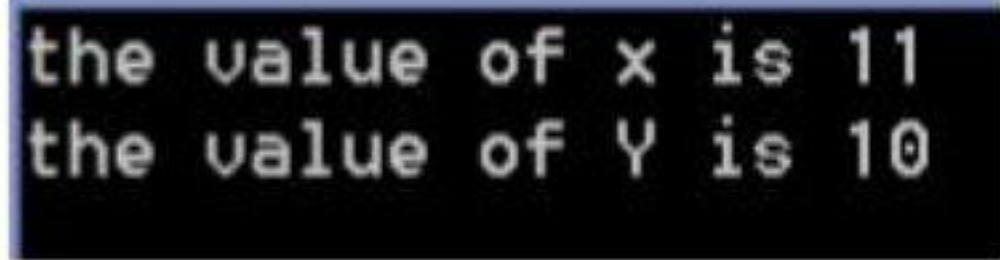
Pre-decrement: --variable Post-decrement: variable--

In Prefix form first variable is first incremented/decremented, then evaluated In Postfix form first variable is first evaluated, then incremented/decremented

let's see an example below to see the difference between pre increment and post increment.

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5  int x ,y;
6
7  int i=10,j=10;
8  x= ++i;
9
10 y= j--;//store the value of j to y then add one to j
11
12 cout<<"the value of x is "<<x<<endl; cout<<"the value of Y is
1  "<<y<<endl;
3  return 0;
```

Here is the output of the program:



```
the value of x is 11
the value of Y is 10
```

The size of operator

We know that there are different types of data types available in C++ program and each data type uses different size of memory. Hence we can now know how much memory does the variable (data type) holds in memory using the **sizeof** operator. The result is displayed as the number of bytes.

let's see at the following program and see the output.

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int x,y,z,i,h;
6      x=sizeof(char);
7      y=sizeof(float);
8      z=sizeof(double);
9      i=sizeof(int);
10     h=sizeof(bool);
11     cout<<"the size of character in memory in bytes is :    "<<x<<endl;
12     cout<<"the size of float in memory in bytes is :      "<<y<<endl;
13     cout<<"the size of double in memory in bytes is :     "<<z<<endl;
14     cout<<"the size of int in memory in bytes is :        "<<i<<endl;
15     cout<<"the size of bool in memory in bytes is :       "<<h<<endl;
16
17     return 0;
18 }
```

The output of the program that describes the size of the data types in memory in bytes is below:

```
the size of character in memory in bytes is :    1
the size of float in memory in bytes is :       4
the size of double in memory in bytes is :      8
the size of int in memory in bytes is :         4
the size of bool in memory in bytes is :        1
```

Any variable of data type also returns the size of the data type if it is inside the sizeof() operator. example: **int x; // sizeof(x) will print 4. and if x is char data type (char x) then sizeof(x) returns 1.**

Exercise: write a C++ program that accepts two characters, two integers and two floating decimals from keyboard and displays size of each variable and the sum of the numbers.

Order of precedence

Multiple operations can be done in a single program statement and order is necessary which is to be done first and it is done by giving precedence of operators or order of availability. It is just as BODMAS is in ordinary mathematics. The following rank shows the order of C++ arithmetic operators in C++.

Order	Operators
First	()
Second	*, /, %
Third	+, -

In the above ranking it means that things in brackets should be done first, Multiplication, division and modulus have the same precedence and finally addition and subtraction have the same precedence.

Example: think what the output of the expression : $2+3*(8-9)/4+98-(45*3)+6$; practice it and be reasonable.

The following table shows the precedence of all the operators from highest to lowest as we go down the column.

++, --(post increment/decrement)
++- (Pre increment) -- (Pre decrement), sizeof (), l(not), -(unary). +(unary)
V, %
+, -
<, <=, >, >=
==, !=

Comma operator

Now let's try to write some programs to calculate in C++.

Example: Write an error free C++ program that accepts three numbers from keyboard and provides the sum, average and product of the three numbers to the screen.

solution:

```

1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      float n1,n2,n3;// this is declering the three variables
6      float s,p, a; /*these are declering variables
7          to hold the sum product and average*/
8      cout<<"ENTER THREE NUMBERS FROM KEYBOARD TO SEE SOME C++ CALCULATION \n";
9      cin>>n1>>n2>>n3;
10     s=n1+n2+n3; //this is the sum
11     p=n1*n2*n3; // this is the product
12     a=s/3; // this is equal to (n1+n2+n3)/3
13     cout<<"the SUM IS = " <<s<<endl;
14     cout<<"THE PRODUCT IS = " <<p<<endl;
15     cout<<"THE AVERAGE IS " <<a<<endl;
16     return 0;
17 }

```

see some outputs; example lets take three numbers 4, 4.5 and 6

```

ENTER THREE NUMBERS FROM KEYBOARD TO SEE SOME C++ CALCULATION
4
4.5
6
the SUM IS =    14.5
THE PRODUCT IS =    108
THE AVERAGE IS    4.83333

```

Exersise: write a C++ program that accepts four characters and displays them in reverse order of acceptance.

Chapter Three

Flow of Control

Statements

Statements are the instructions given to the computer to perform any kind of action. Action may be in the form of data movement, decision making etc. Statements form the smallest executable unit within a C++ program. Statements are always terminated by semicolon.

Compound Statement

A compound statement is a grouping of statements in which each individual statement ends with a semi-colon. The group of statements is called block. Compound statements are enclosed between the pair of braces ({}). The opening brace ({) signifies the beginning and closing brace (}) signifies the end of the block.

Null Statement

Writing only a semicolon indicates a null statement. Thus ';' is a null or empty statement. This is quite useful when the syntax of the language needs to specify a statement but the logic of the program does not need any statement. This statement is generally used in for and while looping statements.

Conditional Statements

Sometimes the program needs to be executed depending upon a particular condition. C++ provides the following statements for implementing the selection control structure.

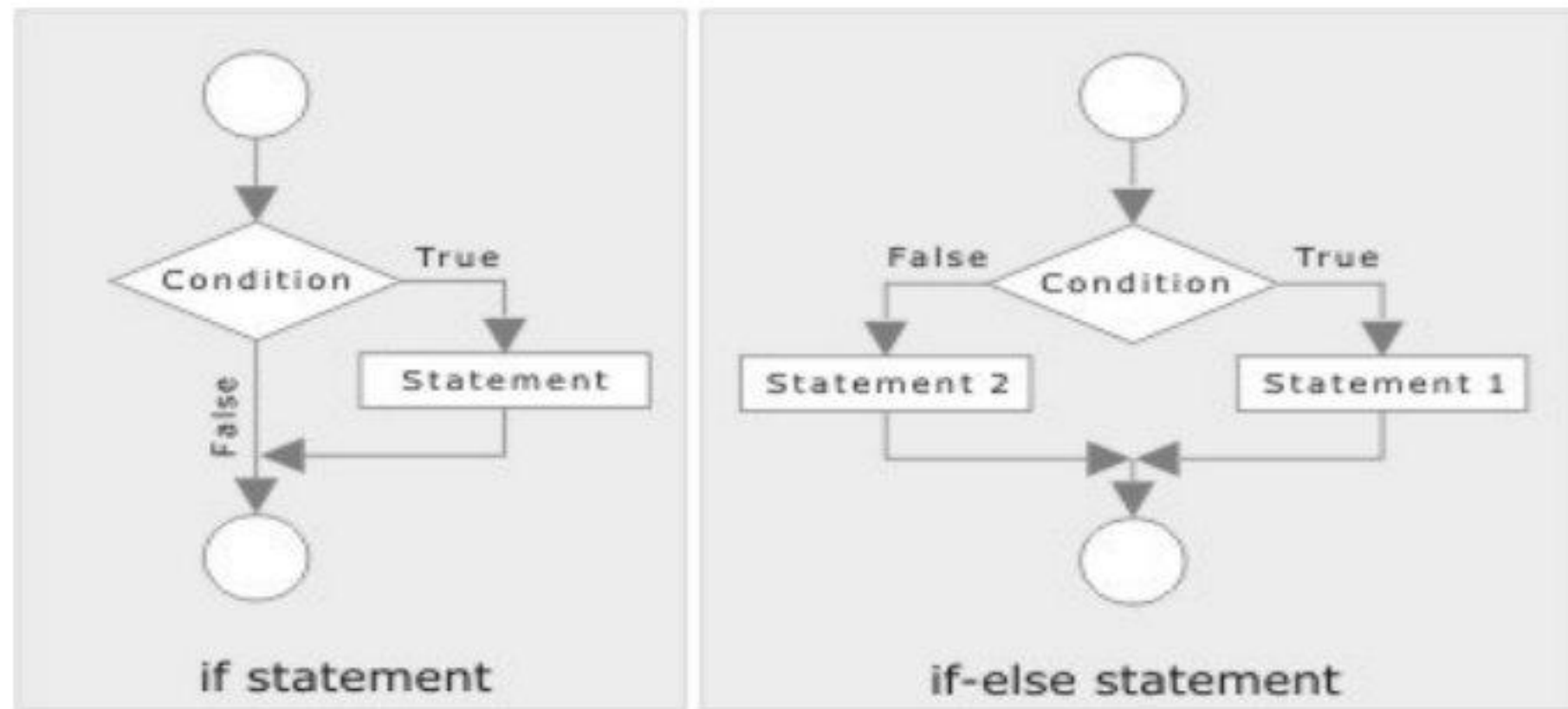
- if statement
- if else statement
- nested if statement
- switch statement

if statement

syntax of the if statement

```
if (condition)
{
    statement(s);
}
```

From the flowchart it is clear that if the if condition is true, statement is executed; otherwise it is skipped. The statement may either be a single or compound statement.



Example

```
#include<iostream>

using namespace std;

int main()
{
    int n;

    cout<<"enter any integer you want"<<endl; // we say integer because the declaration is type integer.
    cin>>n;

    if(n>1000)

    cout<<"the number is large"<<endl;

    return 0;
}
```

Exercise :

Write a program that says THE NUMBER IS TOO SMALL if it is less than 0.

if.....THEN.... else statement

syntax of the if - else statement

if (*condition*)

statement1; // this statement will be displayed if and only if the statement (*condition*) is **true**.

else

statement2; // this statement displays if option one fails. i.e. if the above condition under if is

false.

From the above flowchart it is clear that the given condition is evaluated first. *If the condition is true, statement1 is executed. If the condition is false, statement2 is executed.* It should be kept in mind that statement1 and statement2 can be single or compound statement.

if example

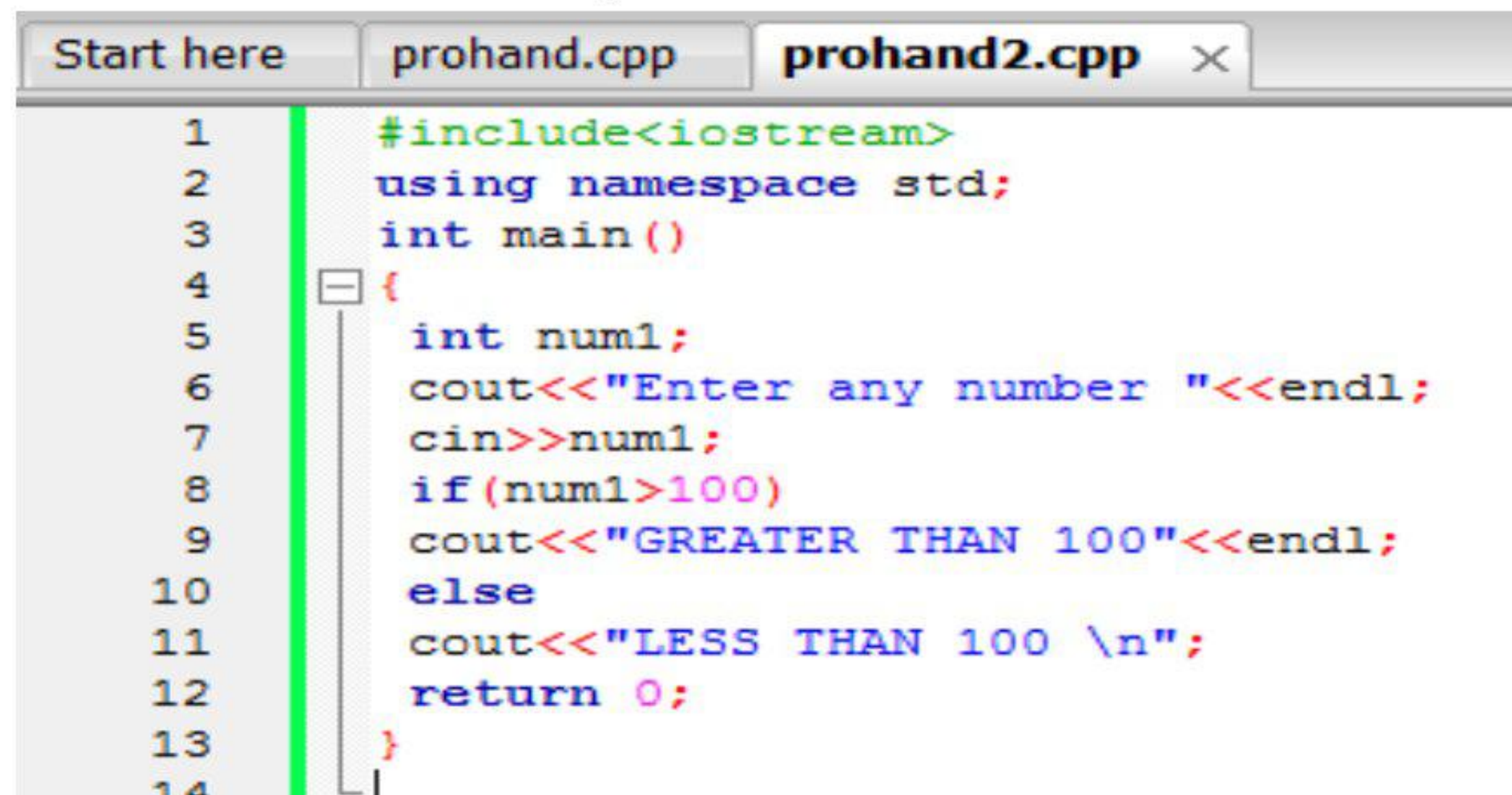
```
int x;  
if (x > 1000)  
    cout << "x is too large";
```

ifthen....else example

```
int x; // declaration of type integer  
if (x > 1000)  
    cout << "x is too large";  
else  
    cout << "x is too small";
```

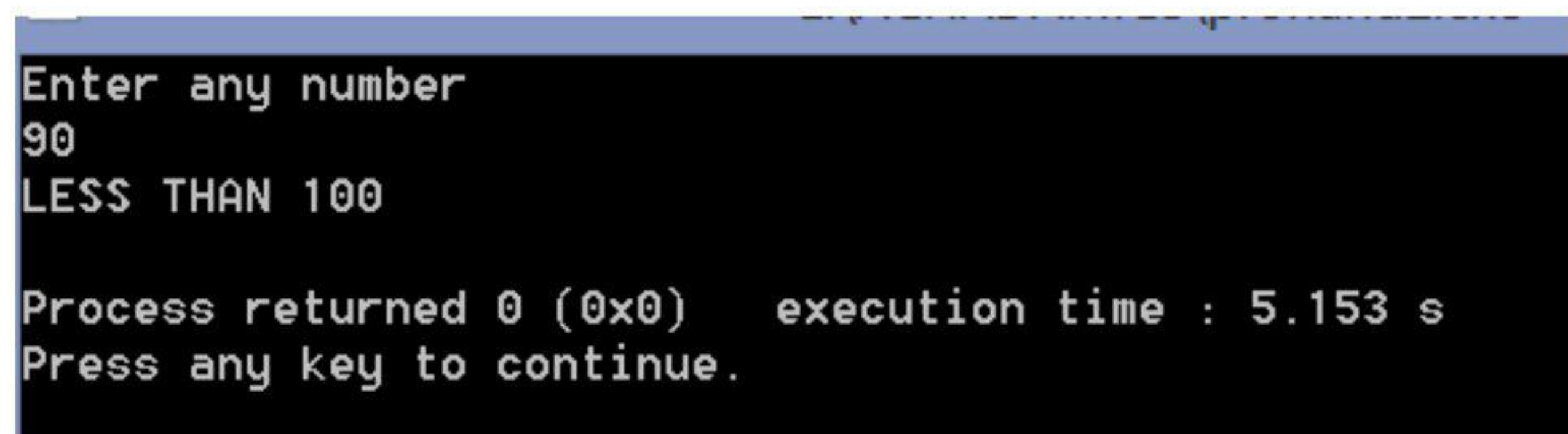
Example 1

write a c++ program that accepts a number from keyboard and displays a text message GREATER THAN 100 if it is greater than 100 and LESS THAN 100 if it is less than 100.



```
1  #include<iostream>  
2  using namespace std;  
3  int main()  
4  {  
5      int num1;  
6      cout<<"Enter any number "<<endl;  
7      cin>>num1;  
8      if(num1>100)  
9          cout<<"GREATER THAN 100"<<endl;  
10     else  
11         cout<<"LESS THAN 100 \n";  
12     return 0;  
13 }  
14
```

Output of the program:



```
Enter any number  
90  
LESS THAN 100  
  
Process returned 0 (0x0)   execution time : 5.153 s  
Press any key to continue.
```

```
E:\ACHADAMICS\prohand2.exe

Enter any number
750
GREATER THAN 100

Process returned 0 (0x0)    execution time : 4.610 s
Press any key to continue.
```

Exersise: Modify example 1 so that it can display a message when the number is 100.

Example 2

Write an error free C++ program that check's whether a given number is even or odd.

```
Start here  prohand.cpp x

1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int n;
6      cout<<"enter the number to be checked for evenness or oddness "<<endl;
7      cin>>n;
8      if(n%2==1){
9          cout<<"the number is odd"<<endl;
10     }
11     else{
12         cout<<"the number is even"<<endl;
13     }
14     return 0;
15 }
16
```

Output of the program:

```
E:\ACHADAMICS\prohand.exe

enter the number to be checked for evenness or oddness
7
the number is odd

Process returned 0 (0x0)    execution time : 10.108 s
Press any key to continue.
```

```
E:\ACHADAMICS\prohand.exe
enter the number to be checked for evenness or oddness
8
the number is even

Process returned 0 (0x0)    execution time : 3.732 s
Press any key to continue.
```

Exercise

Write a program that prints the sum of even number from 0 to 20.

Nested if and multiple if statements

The **if** block may be nested in another if or else block. This is called nesting of if or else block.

Syntax of the nested if statement

```
if(condition 1)
{
    if(condition 2)
    {
        statement(s);
    }
}
```

```
.....
if(condition 1)
    statement 1;
else if (condition 2)
    statement2;
else
    statement3; // the exercise after example 1 is to be done using this syntax.
```

example:

A program that calculates solution of a quadratic equation is given below.

```
#include<iostream>
```

```
#include<cmath>
```

```
using namespace std;
```

```

int main()
{
float a,b,c,d,root1,root2;
cout<<"Enter value of a, b and c : ";
cin>>a>>b>>c;
d=b*b-4*a*c;
if(d==0)
{
root1=(-b)/(2*a);
root2=root1;
cout<<"Roots are real & equal";
}
else if(d>0)
{
root1=(-b+sqrt(d))/(2*a);
root2=(-b-sqrt(d))/(2*a);
cout<<"Roots are real & distinct";
}
else
{
root1=(-b)/(2*a);
root2=sqrt(-d)/(2*a);
cout<<"Roots are imaginary";
}
cout<<"\nRoot 1= "<<root1<<"\nRoot 2= "<<root2;
return 0;
}

```

Switch statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

The if and if-else statements permit two way branching whereas switch statement permits multiple branching. The

syntax of switch statement is:

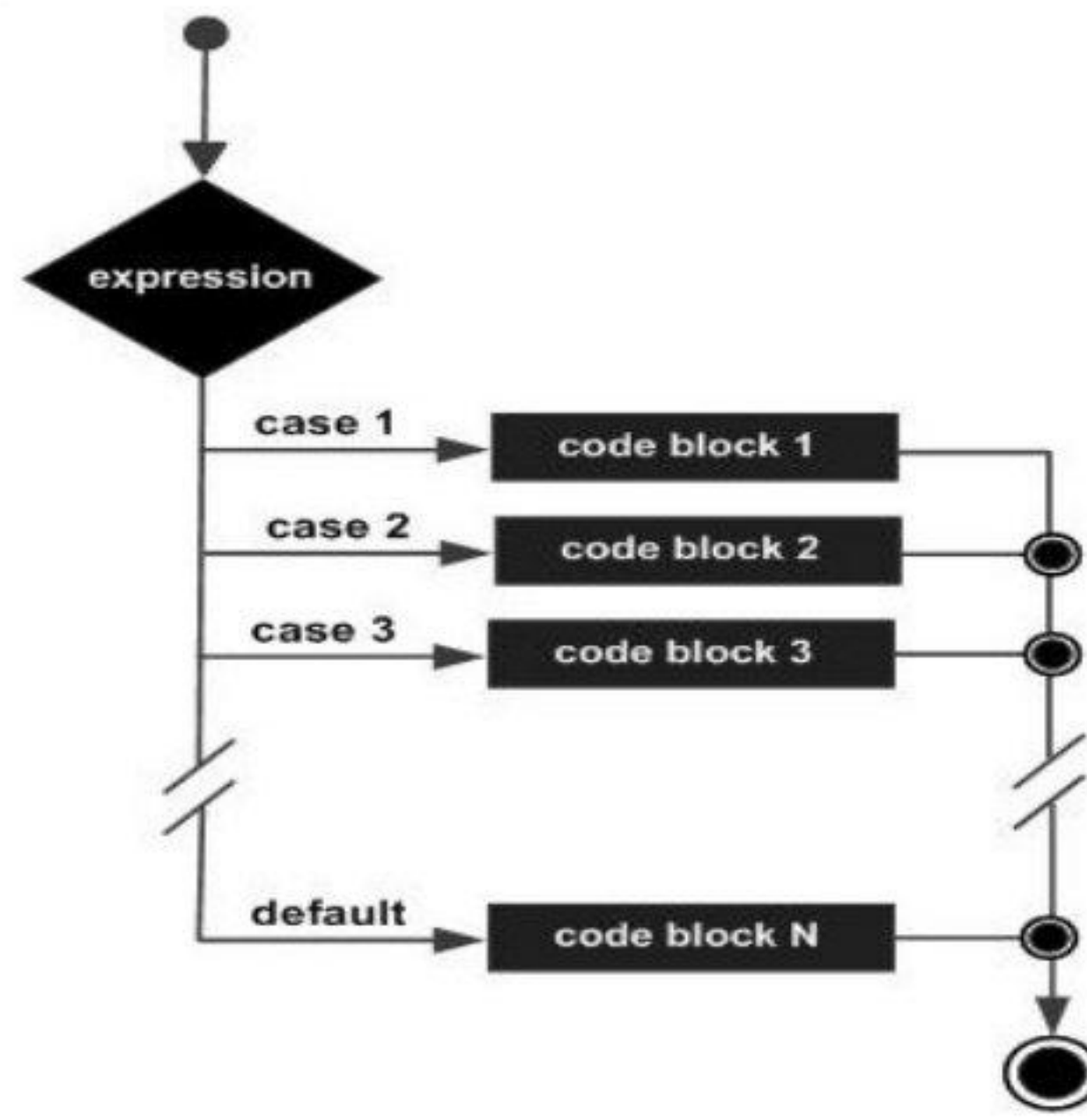
```
switch (var / expression)
{
    case constant1 : statement 1;
    break;
    case constant2 : statement2;
    break;
    .
    .
    default: statement3;
    break;
}
```

The execution of switch statement begins with the evaluation of expression. If the value of expression matches with the constant then the statements following this statement execute sequentially till it executes break. The break statement transfers control to the end of the switch statement. If the value of expression does not match with any constant, the statement with default is executed.

Some important points about switch statement

- The expression of switch statement must be of type integer or character type.
- The default case need not to be used at last case. It can be placed at any place.
- The case values need not to be in specific order.

Flow Diagram:



Example

A program that accepts grade of a student prints its grade and status

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    char grade; // local variable declaration:
```

```
    cout<<"the grade u have"<<endl;
```

```
    cin>>grade;
```

```
    switch(grade)
```

```
    {
```

```
        case 'A' :
```

```
            cout << "Excellent!" << endl;
```

```
            break;
```

```
        case 'B' :
```

```
        case 'C' :
```

```
            cout << "Well done" << endl;
```

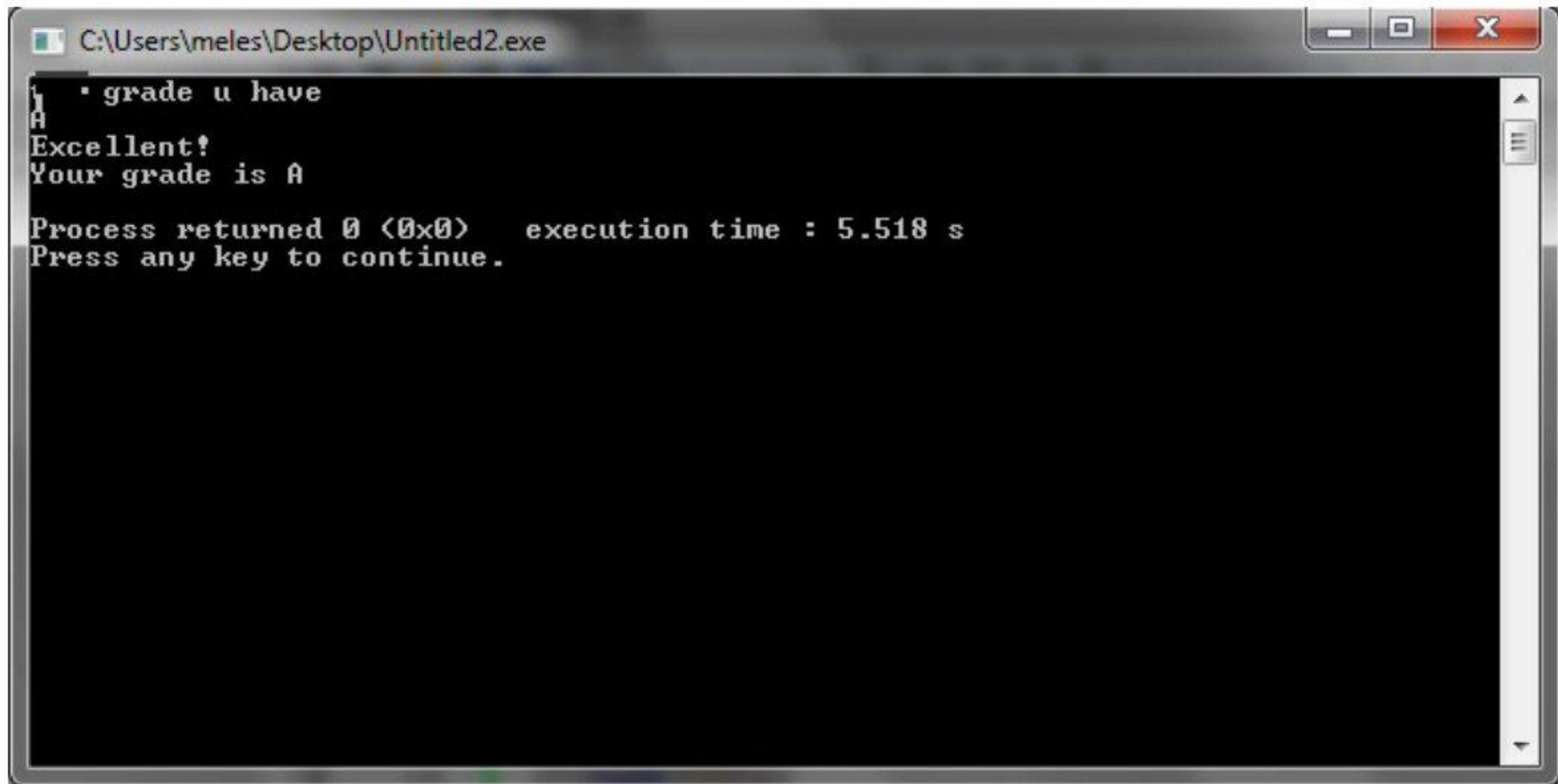
```

        break;
    case 'D' :
        cout << "You passed" << endl;
        break;
    case 'F' :
        cout << "Better try again" << endl;
        break;
    default :
        cout << "Invalid grade" << endl;
    }
    cout << "Your grade is " << grade << endl;
    return 0;
}

```

Out put

If the input grade is A



```

C:\Users\meles\Desktop\Untitled2.exe
> grade u have
A
Excellent!
Your grade is A
Process returned 0 (0x0)   execution time : 5.518 s
Press any key to continue.

```

2. Write a program using switch that displays the days of a week (days from Monday to Sunday).

Flow of Control

Looping statement

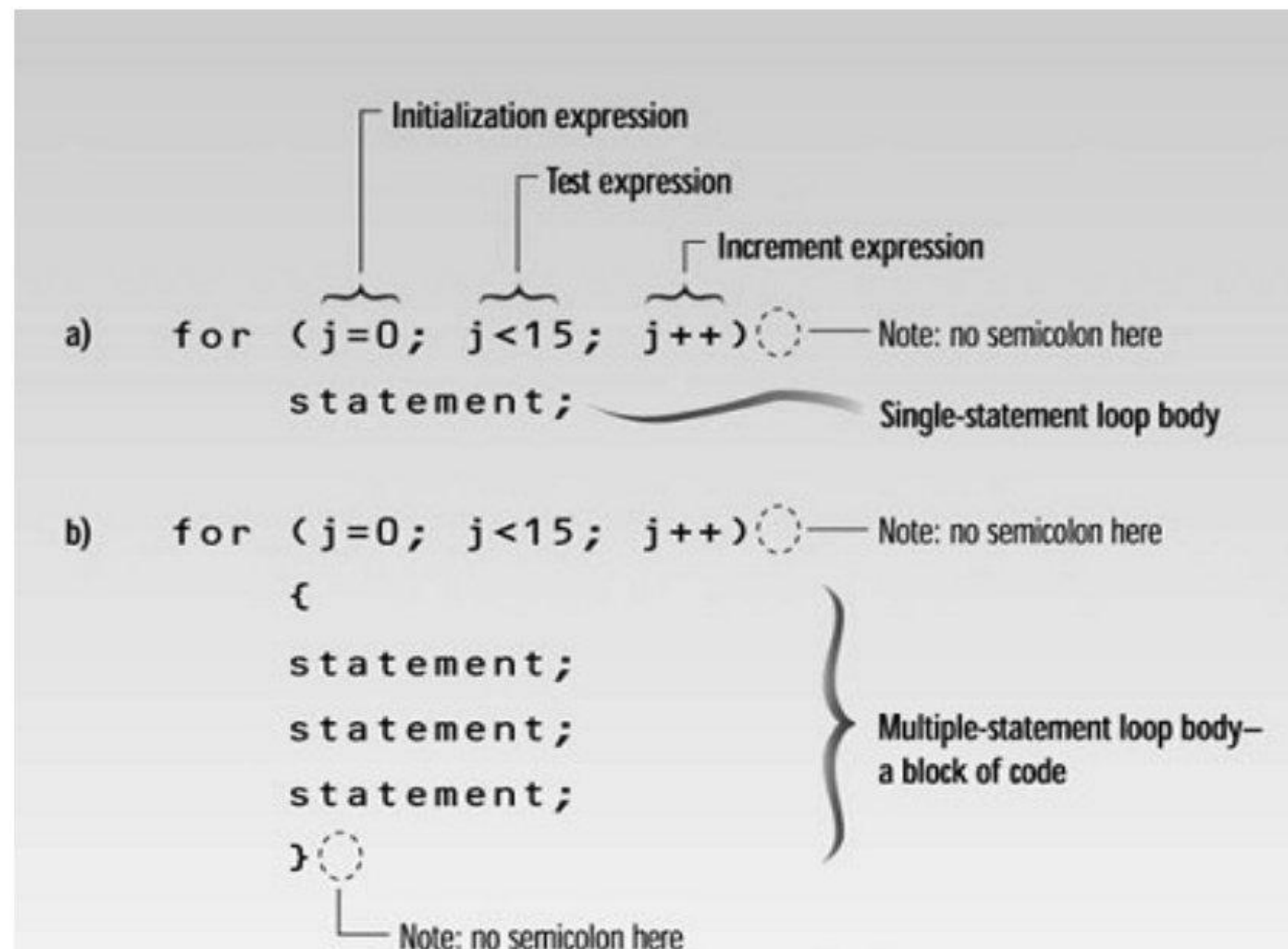
It is also called a Repetitive control structure. Sometimes we require a set of statements to be executed a number of times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called looping. C++ provides the following construct.

- **for loop**
- **while loop**
- **do-while loop**

The for loop

The “for loop” executes a section of code a fixed number of times. It is usually (although not always) used when you know, before entering the loop, how many times you want to execute the code.

Syntax of for loop:



Initialization expression

The initialization expression is executed only once, when the loop first starts. It gives the loop variable an initial value.

Increment/decrement expression

The increment/decrement expression changes the value of the loop variable. It is always executed at the end of the loop, after the loop body has been executed. Here in the above, the increment operator ++ adds 1 to j each time through the loop.

Test expression

The test expression usually involves a relational operator. It is evaluated each time through the loop, just before the body of the loop is executed. It determines whether the loop will be executed again. If the test expression is true, the loop is executed one more time. If it's false, the loop ends, and control passes to the statements following the loop.

Flow chart representation of for loop

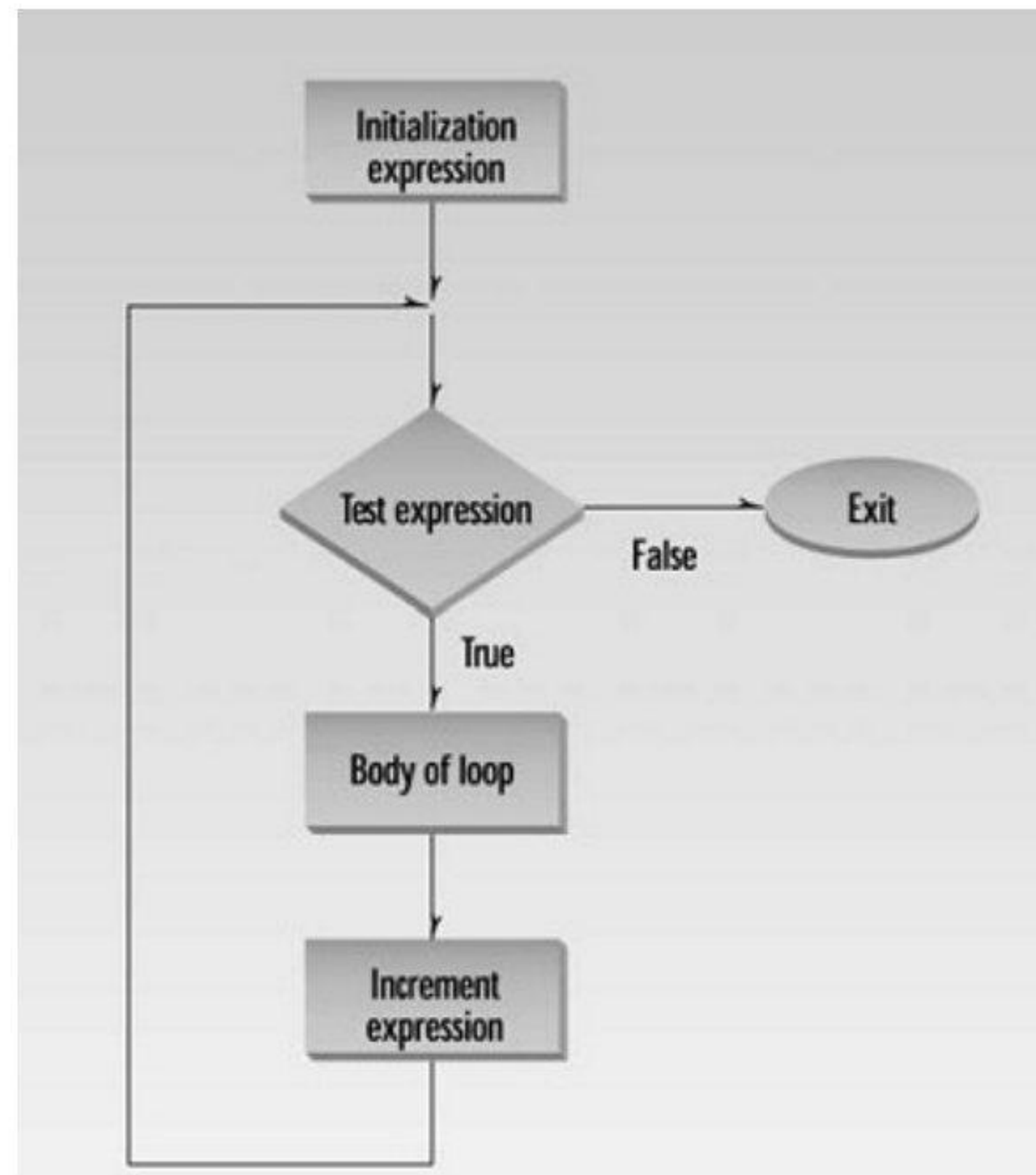


Fig operation of for loop

Examples

- 1) Write a program that calculates the sum of the square of natural numbers from 1 up to 10($1^2+2^2+3^2+\dots\dots\dots+10^2$)
- 2) Modify the above program so that the program displays the content of the loop body in each iteration and finally display the sum in new line. So the result of program seems the following
 1, 5, 14, 30, 55, 91, 140, 204, 285, 385

 385
- 3) Again modify the above program so that the program does not calculate the sum but it only displays the square of natural numbers from 1 up to 10

Answers

#1) program for Q1

```
#include<iostream>

using namespace std;

int main()

{

    int num;

    int sum=0;

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

    {

        sum+=num*num;

    }

    cout<<sum;

    return 0;

}
```

#2)**program for Q2**

```
#include<iostream>

using namespace std;

int main()

{

    int num;

    int sum=0;

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

    {

        sum+=num*num;

        cout<<sum<<" ";

    }

    cout<<endl;

    cout<<sum;

    return 0;

}
```

Exercise

1) Write a program that will display the following format on the screen

a) x
xx

xxx

xxxx

xxxxx

c) x
xx

xxx

xxxx

xxxxx

b) 1
12

123

1234

12345

d) 55555
4444

333

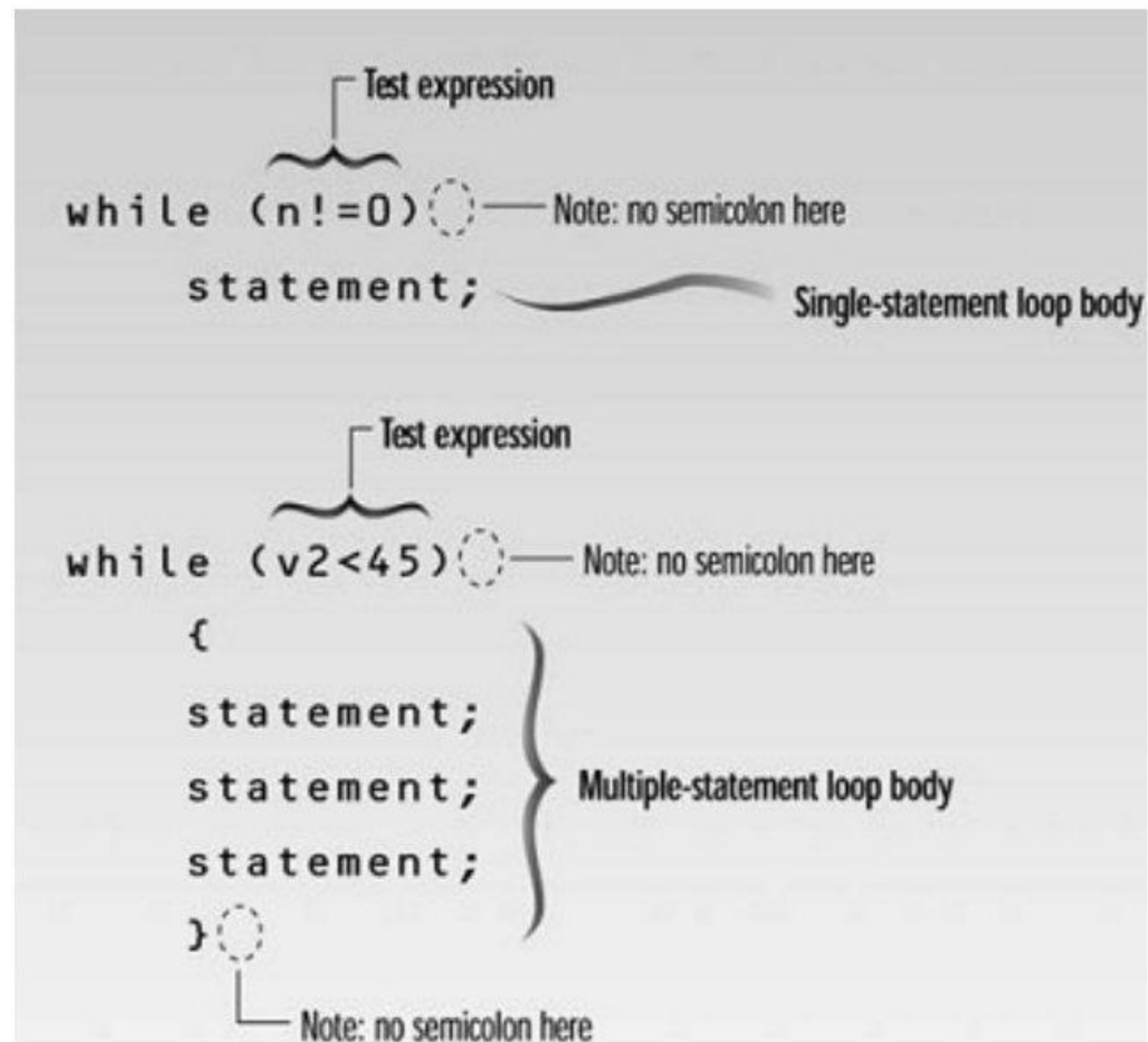
22

1

The while loop

The “for loop” does something a fixed number of times. What happens if you don’t know how many times you want to do something before you start the loop? In this case a different kind of loop may be used: the while loop.

Syntax:



As long as the test expression is true the loop continues to be executed

Flow chart representation

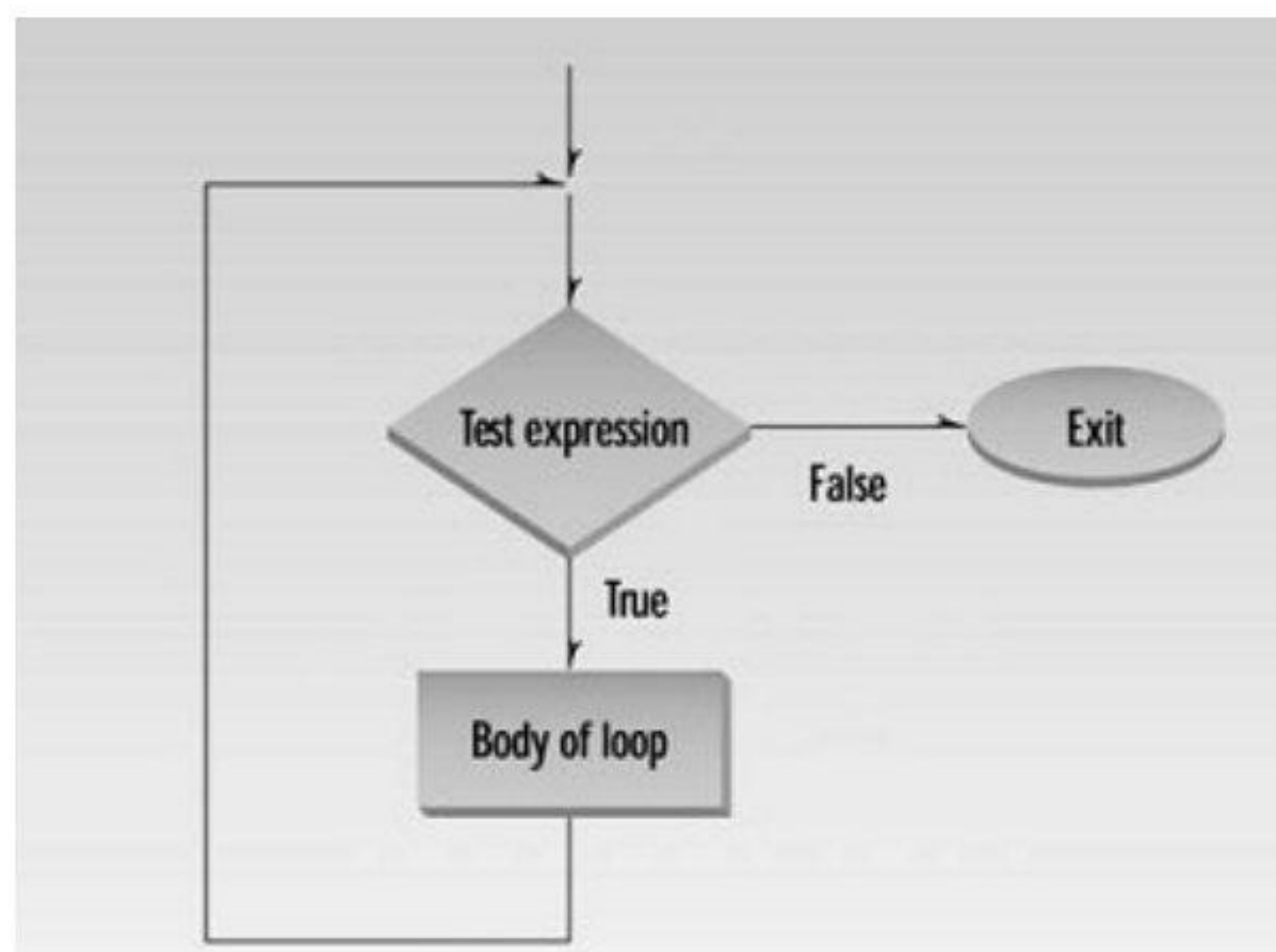


Fig operation of while loop

Example

- 1) write a program that calculates the factorial of a number

```
#include<iostream>

using namespace std;

int main()

{

    int num;

    float fact=1;

    Cout<<"enter the number\n";

    Cin>>num;

    while(num>1)

    {

        fact*=num;

        --num;

    }

    cout<<"the factorial is :"<<fact;

    Return 0;

}
```

The do loop

In a while loop, the test expression is evaluated at the beginning of the loop. If the test expression is false when the loop is entered, the loop body won't be executed at all. But sometimes you want to guarantee that the loop body is executed at least once, no matter what the initial state of the test expression. When this is the case you should use the do loop, which places the test expression at the end of the loop.

Syntax:

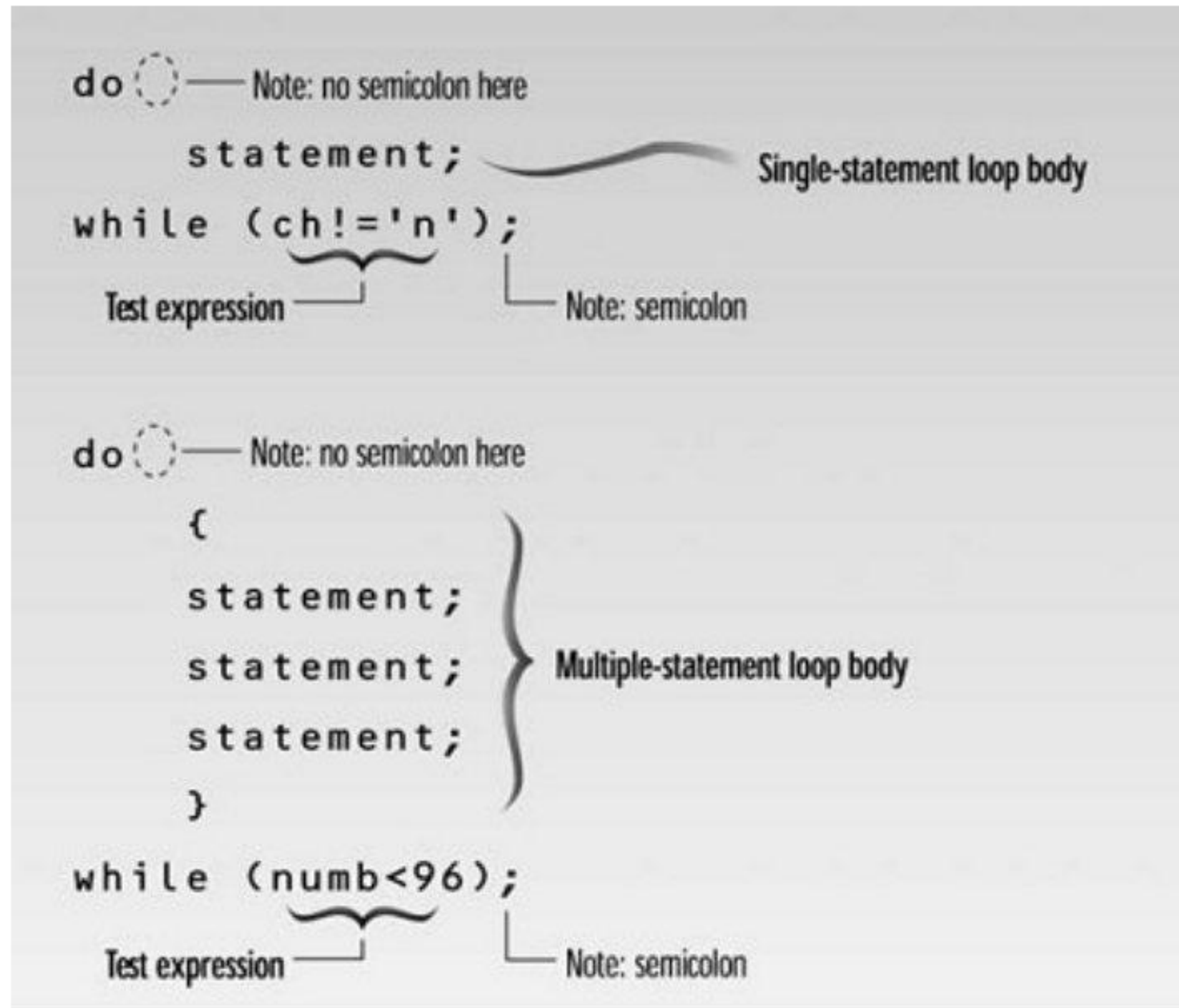
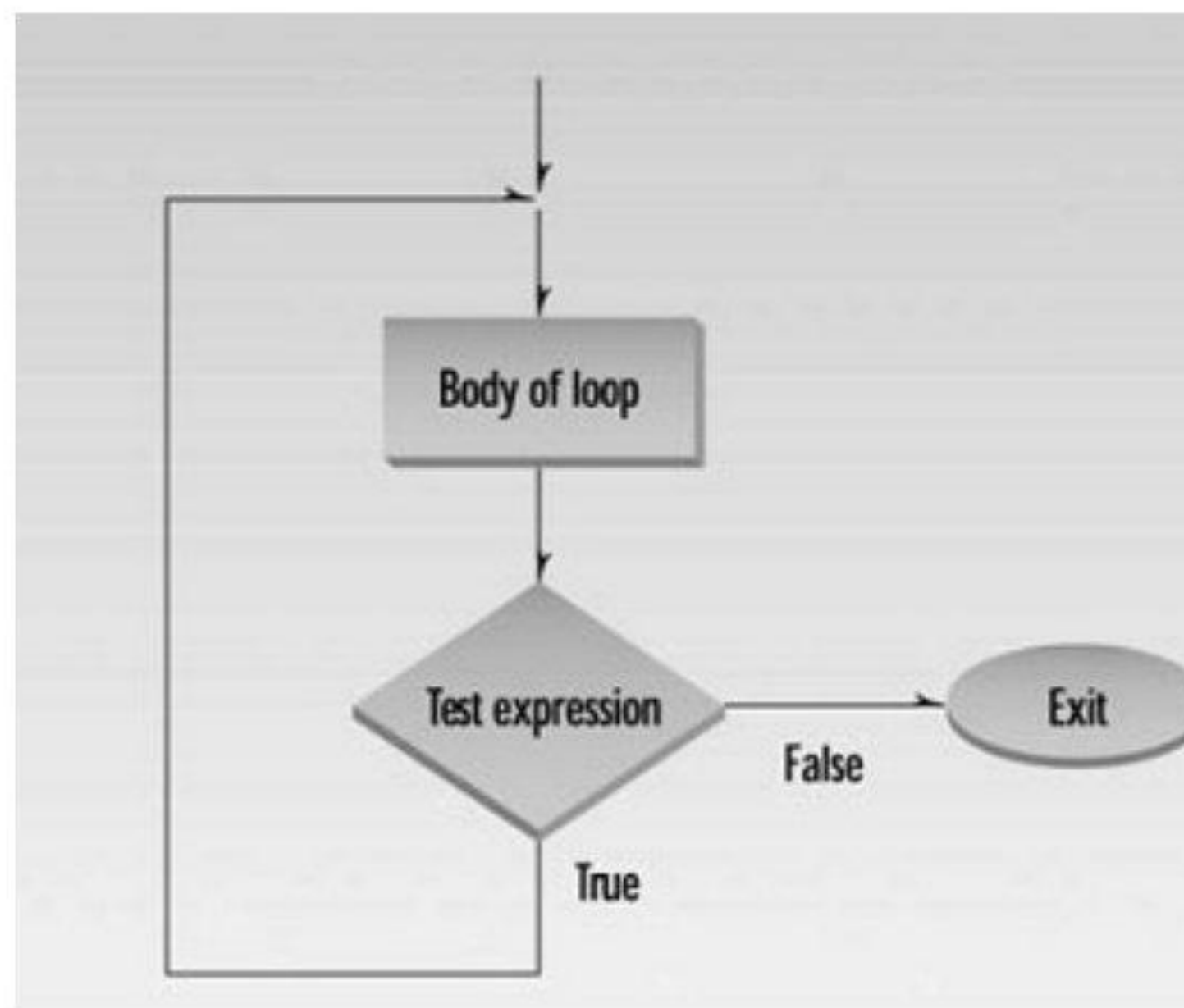


Fig operation of do loop

Flow chart representation



Example

- 1) Write a program invites the user to enter two numbers: a dividend (the top number in a division) and a divisor (the bottom number). It then calculates the quotient (the answer) & the remainder and prints out the result. The program should also ask if the user wants to repeat the above operation and depending on user input the program terminate or loop again

Answer:

```
#include<iostream.h>

using namespace std;

int main()
{
int divisor,dividend;

char ch;

do
{
cout<<"enter the dividend\n";

cin>>dividend;

cout<<"enter the divisor\n";

cin>>divisor;

cout<<"the quotient is"<<dividend/divisor<<endl;

cout<<"the remainder is"<<dividend%divisor<<endl;

cout<<"do you want to continue(y/n)\n";

cin>>ch; }

while(ch=='y');

return 0;

}
```

Jump Statements

The jump statements unconditionally transfer program control within a function.

- **Goto statement**
- **Break statement**
- **Continue statement**

The goto statement

goto allows to make jump to another point in the program.

goto pqr;

pqr: pqr is known as label. It is a user defined identifier. After the execution of goto statement, the control transfers to the line after label pqr.

The break statement

The break statement, when executed in a switch structure, provides an immediate exit from the switch structure. Similarly, you can use the break statement in any of the loop. When the break statement executes in a loop, it immediately exits from the loop.

<pre>// break loop example #include <iostream> using namespace std; int main () { int n; for (n=10; n>0; n--) { cout << n << ", "; if (n==3) { cout << "countdown aborted!"; break; } } return 0; }</pre>	<pre>10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!</pre>
--	--

The continue statement

The continue statement is used in loops and causes a program to skip the rest of the body of the loop.

while (condition)

{

Statement 1;

 If (condition)

 continue;

 statement;

}

Example:

What is the output of the following program?

```
#include<iostream>

using namespace std;

int main()

{

for(int j=0;j<5;j++)

{

    if(j%3==0)

        continue;

cout<<"x"<<endl;

}

return 0;

}
```

The continue statement skips rest of the loop body and starts a new iteration.

The exit () function

The execution of a program can be stopped at any point with exit () and a status code can be informed to the calling program. The general format is

exit (code) ;

where code is an integer value. The code has a value 0 for correct execution. The value of the code varies depending upon the operating system.

Chapter 4

Arrays

An array is a series of elements of **the same data type placed** in continuous memory locations that can be individually referenced by adding an index to a unique identifier. an array lets you declare and work with a collection of values of the same type. Let's say you want to declare four integers. With the knowledge from the last few tutorials you would do something like this:

```
int a , b , c , d;
```

What if you wanted to declare a thousand variables?

That will take you a long time to type. This is where arrays come in handy. An easier way is to declare an array of four integers, like this:

```
int a[4]; // this holds four different variables of type int.
```

The four separate integers inside this array are accessed by an index. Each element can be accessed, by using square brackets, with the element number inside. All arrays start at element zero and will go to n-1 (where n is the array size). (In this case from 0 to 3.).

Note: The index number, which represents the number of elements the array is going to hold, must be a constant value. Because arrays are build out of non-dynamic memory blocks. In a later tutorial we will explain arrays with a variable length, which uses dynamic memory.

So if we want to fill or initialize each element you get something like this:

```
int a[4];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
```

If you want to use an element, for example for printing, you can do this:

```
cout << a[1];
```

See another example; we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called group. this can be represented: **int group[5]**. The values of the five variables in memory could be represented like this.

Index 0	1	2	3	4
1 st element	2 nd element	3 rd element	4 th element	5 th elements

Where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

General Array Declaration:

- ✓ First give the type of the array such as int, double, float, bool, char, void... etc.
- ✓ Give any valid name for the array. It can be any valid identifier.
- ✓ Finally the size of the array inside the square brackets.
 - **type** name [size of elements];
 - Examples, int age [20], bool truearr[12], double mark[100], float average[]

Where type is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain. Therefore, in order to declare an array called **group** as the one shown in the above table, it can be declared as: **int group [5];**

Initialization of arrays

Initialization is the process of giving initial value of the array elements. For example for the above example of array group [5] I can initialize with any of five groups in your class. For example let me take groups 2,5,6,8 and 11 to be the top five groups out of the 11 groups. Hence, it has the form:

int group [5] = {2, 5,6,8,11} // this shows that:

- ❖ group[0]=2
- ❖ group[1]=5
- ❖ group[2]=6
- ❖ group[3]=8
- ❖ group[4]=11

eg.2 . char alphabet[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};

What is the value of alphabet [6], alphabet[25], alphabet [0], alphabet[26]

We can also initialize as : int group[5]={0} // this initializes all the members to zero.

Eg3. float temperature[] = { 0.0, 112.37, 98.6 };

We can also initialize some of the members and the others will be initialized by default to zero. See for the above example: int group[5]={9,4,3} // this initializes the first three elements with the values in the

braces and the values of `group[3]`, `group[4]` are initialized to zero by default.

Accessing Individual Array Components

- You can access any one particular element if you need or want. For example, let's have an array of size 4, type **float** and **name** angle.

— **float** angle[4];

- ✓ angle[0] = 1.2; // this is the first element and we are initializing it to 1.2.
- ✓ angle[1] = 3.4;
- ✓ angle[2] = 0.0;
- ✓ angle[3] = 45.6;
- ✓ angle[4] does not exist // this is error due to the reason that index is from **0 to n-1** where n is size.

Things that u can do with array elements

- ✓ angle[2] = 9.5; // assigning any value u want independently (accessing them)
- ✓ cin >> angle[2]; // you can accept value from keyboard to any member.
- ✓ Cout << angle[3]; // you can print a single element or many to the screen
- ✓ Y = sqrt(angle[1]); // you can do mathematical operations to a single element
- ✓ Z = 8.4 * angle[0] + 65.00; // you can use them in any calculations independently

Things you shouldn't do with array elements

- ✗ Out of bounds array index → an index value that is greater than the **array size-1** or less than **zero**.

The advantage of using arrays is clear. They help us to have clear, short and neat programs in addition to other advantages. For example, If you wanted to read in 1000 ints and print them in reverse order, it would take a program that's over 3000 lines long. Doesn't worry there is a better way to solve this problem using arrays. An array of size 1000 would do the trick. See how we can

```
#include<iostream>
```

```
using namespace std;
```

```
int main(){
```

```
int values[1000];
```

```
for(int i=0;i<1000;i++)
```

```
cin>>values[i]<<endl;
```

```
for(int i=1000;i>0;i--)
```

```
cout<<values[i]<<endl;
```

```
return 0;
```

}

One of the nice things about arrays is that you can use a loop to manipulate each element. When an array is declared, the values of each element are not set to zero automatically.

In some cases **you** want to “re-initialize” the array (which means, setting every element to zero). This can be done like in the example above, but it is easier to use a loop. Here is an example:

```
#include<iostream>
using namespace std;

int main()
{
    int a[4];
    int i;

    for ( i = 0; i < 4; i++ )
        a[i] = 0;
    for ( i = 0; i < 4; i++ )
        cout << a[i] << '\n'; //prints out all zero values.
    return 0;
}
```

What is the output of this program????

N.B. When thinking about arrays, whenever you type just the name of the array without any brackets you mean the entire array. When you type the array name with brackets and the number inside, you mean just that particular location. i.e. see the above program and when I say **a** I mean the entire array 4 elements. When I say **a[1]** it is to mean the second element of the array.

Exercises

- 1) Write a C++ program that accepts the ID of your group members and display to the screen all the ID numbers. (*please use arrays*)
- 2) Modify number one to display the ID number in ascending order to the screen.

- 3) Write a C++ program that accepts age of 6 students from keyboard and display only the age of odd indexed students in the array.
- 4) Modify question three to print out the name of the even indexed students.
- 5) Write a C++ program to identify sex of your group members. You should declare the size of array equal with number of members, initialize the sex of every member with M for male and F for female and print the number of female and male students of your group to the screen. Hint: use **char** type to declare your array.
- 6) Write a C++ program that initializes all 65 size of integer variables to zero and print all the values in each separated by tap.

Multi-dimensional arrays

The programming language C++ supports the use of multi-dimensional arrays.

Like one-dimensional arrays, multi-dimensional arrays are homogeneous. This means that all values stored in the array variable are of the same data type. All int or all float or all char etc... Unlike one-dimensional arrays, multi-dimensional arrays have more than one position or index value that must be specified in order to access a single value stored in the array.

Two-Dimensional Arrays

A two-dimensional array requires two position or index values be specified in order to access a single value stored in the array variable. In general these are referred to as the “row” and “column” values.

Basic Declaration Format:

data_type array_variable_name [Rows] [Columns] {optional initializing list}; **Data Type:**

This is the data type of all of the values stored in the array. This can be any legal data type supported by C++.

Array Variable Name:

Identifier name used to reference the multiple values stored in the array.

Rows:

This is the “maximum” number of rows that can be used to store values in the array. This value gives the array row storage positions 0 to the maximum - 1.

Columns:

This is the “maximum” number of columns that can be used to store values in the array. This value gives the array column storage positions 0 to the maximum - 1.

Optional Initializing List:

This allows values to be stored in the array when the array variable is declared.

Examples:

```
int Numbers [ 3 ] [ 4 ];
```

Numbers can have a maximum of 12 values stored in it. All 12 values are data type int. All 12 values are uninitialized.

Numbers

Rows\Columns	Column 0	Column 1	Column 2	Column 3
Row 0	Numbers[0][0]	Numbers[0][1]	Numbers [0] [2]	Numbers[0][3]
Row 1	Numbers[1][0]	Numbers[1][1]	Numbers[1][2]	Numbers[1][3]
Row 2	Numbers[2][0]	Numbers[2][1]	Numbers [2] [2]	Numbers[2][3]

Numbers

Rows\Columns	Column 0	Column 1	Column 2	Column 3
Row 0	?	?	?	?
Row 1	?	?	?	?
Row 2	?	?	?	?

```
float Values [ 2 ] [ 3 ] = { { 5.5, -3.2, 0.0 }, { 10.0, 17.5, -25.13 } };
```

Values can have a maximum of 6 values stored in it. All 6 values are data type float. All 6 values are initialized.

Values

Rows\Columns	Column 0	Column 1	Column 2
Row 0	Values[0][0]	Values[0][1]	Values[0][2]
Row 1	Values[1][0]	Values[1][1]	Values[1][2]

Values

Rows\Columns	Column 0	Column 1	Column 2
Row 0	5.5	-3.2	0.0
Row 1	10.0	17.5	-25.13

```
char Letters [ 2 ] [ 3 ] = { { 'A' }, { 'D', 'E' } };
```

Values can have a maximum of 6 values stored in it. All 6 values are data type char. Only 3 of the 6 values are initialized from the initializing list. The other 3 values are initialized with the NULL character - character 0.

Letters

Rows\Columns	Column 0	Column 1	Column 2
Row 0	Letters[0][0]	Letters[0][1]	Letters[0][2]
Row 1	Letters[1][0]	Letters[1][1]	Letters[1][2]

Letters

Rows\Columns	Column 0	Column 1	Column 2
Row 0	A	'\0'	'\0'
Row 1	D	E	'\0'

If Letters had been declared as a numeric data type then the 3 unspecified positions in the initializing list would all have been initialized with 0s.

Simple Examples of using Two-Dimensional Array Variables

When using array variables use them just like any other variable of the same data type. The only difference is that there is now a list of values instead of a single value. Specify the name of the array variable and the row position value and the column position value that contains the value that is desired.

```
int Number_1, Number_2; int
Numbers [ 3 ] [ 2 ];
```

```
Number_1 = 5;
```

```
Numbers [ 0 ] [ 0 ] = 5;
```

```
cout << Number_1 << endl;
```

```
cout << Numbers [ 2 ] [ 1 ] << endl;
```

```
cin >> Number_1;
```

```
cin >> Numbers [ 2 ] [ 0 ];
```

```
Sum = Sum_2_Numbers ( Number_1, Number_2 );
```

```
Sum = Sum_2_Numbers ( Numbers [ 0 ] [ 0 ], Numbers [ 0 ] [ 1 ] );
```

```
int Sum_2_Numbers ( const int, const int );
```

```
// function declaration
```

Do not try to use position values that do not exist. When running a program that does this a “Segmentation Fault” error may occur. This means trying to access memory that does not exist. This is another example of a runtime error.

Array Storage - Row Major versus Column Major

Even though Two-Dimensional array variables are referenced in programs using a row position value and a column position value, they are all store in physical memory in sequential order as a one dimensional array.

There are two methods that are supported by programming languages for storing array variables in physical memory: Row Major and Column Major.

Row Major - arrays are stored in physical memory one entire row at a time. Row 0 followed by Row 1 followed by Row 2 followed by etc...

int List [2] [3]			
Rows\Columns	Column 0	Column 1	Column 2
Row 0	List[0][0]	List[0][1]	List[0][2]
Row 1	List[1][0]	List[1][1]	List[1][2]

List

List[0][0]	List[0][1]	List[0][2]	List[1][0]	List[1][1]	List[1][2]
------------	------------	------------	------------	------------	------------

The programming language C++ uses Row Major.

Column Major - arrays are stored in physical memory one entire column at a time. Column 0 followed by Column 1 followed Column 2 followed by etc.

int List [2] [3]			
Rows\Columns	Column 0	Column 1	Column 2
Row 0	List[0][0]	List[0][1]	List[0][2]
Row 1	List[1][0]	List[1][1]	List[1][2]

List

List[0][0]	List[1][0]	List[0][1]	List[1][1]	List[0][2]	List[1][2]
------------	------------	------------	------------	------------	------------

The programming language FORTRAN uses Column Major.

The for loop is commonly used to access the values that are stored in an array variable. With two-dimensional arrays two for loops are required in order to access all of the values. One of the for loops is used to control processing the row position values. One of the for loops is used to control processing the column position values.

It is important to remember that C++ uses Row Major to store values in array variables. In order to efficiently access all the values stored in the array process the two-dimensional array one entire row at a time.

```
int List [ 2 ] [ 3 ] = { { 0, 1, 2 }, { 3, 4, 5 } };
```

List

Rows\Columns	Column 0	Column 1	Column 2
Row 0	0	1	2
Row 1	3	4	5

List - as stored in memory - Row Major

0	1	2	3	4	5
---	---	---	---	---	---

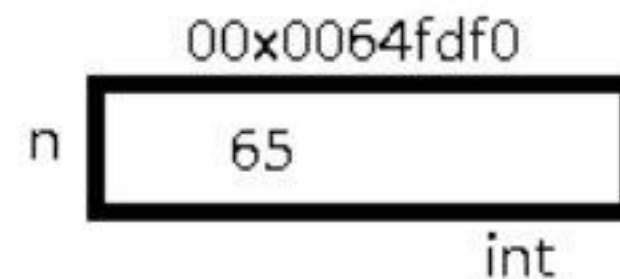
Chapter 5

References and Pointers in C++

We already know that a variable declaration associates three fundamental attributes to the variable: its *name*, its *type*, and its memory *address*. For example, the declaration `int n;` associates the name `n`, the type `int`, and the address of some location in memory where the value of `n` is stored. Suppose that address is `0x0064fdf0`. Then we can visualize `n` like this:



Let `n` holds the number of section B students. I.e. `int n = 66;` this will be like follows in memory.



In C++, you can obtain the address of a variable by using the *reference operator* `&`, also called the *address operator*. The expression `&n` evaluates to the address of the variable `n`. Let's see the program output for the above example.

```
int main()
{ int n=66;
  cout << "n = " << n << endl; // prints the value of n
  cout << "&n = " << &n << endl; // prints the address of n
}
```

Output:

```
n = 66
&n = 0x0064fdf0
```

References

A *reference* is an alias or synonym for another variable. It is declared by the syntax

`type& ref-name = var-name;`

where *type* is the variable's type, *ref-name* is the name of the reference, and *var-name* is the name of the variable. For example, in the declaration

```
int& rn=n; // rn is a synonym for n
```

`rn` is declared to be a reference to the variable `n`, which must already have been declared.

Let's write.

```
#include<iostream>
using namespace std;
int main()
{ int n=66;
  int& rn=n; // r is a synonym for n
  cout << "n = " << n << ", rn = " << rn << endl;
  --n;
  cout << "n = " << n << ", rn = " << rn << endl;
  rn *= 2;
  cout << "n = " << n << ", rn = " << rn << endl;
  return 0;
}
```

Output:

```
n = 66, rn = 66
n = 65, rn = 65
n = 130, rn = 130

Process returned 0 (0x0)   execution time : 0.727 s
Press any key to continue.
```

The two identifiers `n` and `rn` are different names for the same variable; they always have the same value.

Like constants, references must be initialized when they are declared. But unlike a constant, a reference must be initialized to a variable, not a literal:

```
int& rn=66; // ERROR: 66 is not a variable!
```

(Some compilers may allow this, issuing a warning that a temporary variable had to be created to allocate memory to which the reference `rn` can refer.)

```
n=66;
int& rn=n; // right
```

```
#include<iostream>
using namespace std;
int main()
{ int n=44;
  int& rn=n; // r is a synonym for n
  cout << " &n = " << &n << ", &rn = " << &rn << endl;
  int& rn2=n; // r is another synonym for n
  int& rn3=rn; // r is another synonym for n
  cout << "&rn2 = " << &rn2 << ", &rn 3 = " << &rn3 << endl;
  return 0;
}
```

Output:

```
&n = 0x22fef0, &rn = 0x22fef0
&rn2 = 0x22fef0, &rn 3 = 0x22fef0

Process returned 0 (0x0)   execution time : 0.627 s
Press any key to continue.
```

The hexadecimal value in the output is the address of the value in the memory. They are all the same because they all refer to the value of `n`. This address is the memory address where 66 is stored in memory. You will never get the same output for memory address

for the same program executed in different computers. Why do you think?

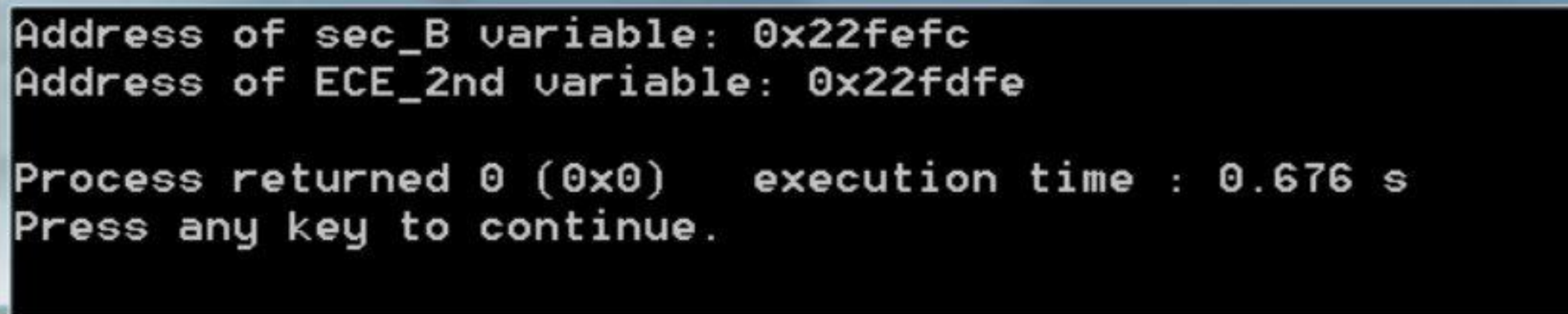
Pointers

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined: please have a look at the following program of your section.

```
#include <iostream>
using namespace std;
int main ()
{
    int sec_B;
    char ECE_2nd[254];
    cout << "Address of sec_B variable: ";
    cout << &sec_B << endl;
    cout << "Address of ECE_2nd variable: ";
    cout << &ECE_2nd << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:



```
Address of sec_B variable: 0x22fefc
Address of ECE_2nd variable: 0x22fdfe

Process returned 0 (0x0)   execution time : 0.676 s
Press any key to continue.
```

Note the difference between the addresses. It is two bytes. I.e 0x22fefc to 0x22fdfe. It is the size of the data type int. Those address can be easily accessed using pointers.

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long **hexadecimal** number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++:

There are few important operations, which we will do with the pointers very frequently. **(a)** we define a pointer variables **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator `*` that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <iostream>
using namespace std;
int main ()
{
    int var = 20;    // actual variable declaration.
    int *ip;         // pointer variable
    ip = &var;       // store address of var in pointer variable
    cout << "Value of var variable: ";
    cout << var << endl;
    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;
    // access the value at the address available in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;
    return 0;
}
```

Output:

```
Value of var variable: 20
Address stored in ip variable: 0x22fef8
Value of *ip variable: 20

Process returned 0 (0x0)   execution time : 1.112 s
Press any key to continue.
```

The second line on the output is not the same if you run in other computer. it is because it is the particular location of the variable in the memory.

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr;

    // let us have array address in pointer.
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Address of var[" << i << "] = ";
        cout << ptr << endl;

        cout << "Value of var[" << i << "] = ";
        cout << *ptr << endl;

        // point to the next location
        ptr++;
    }
    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
C:\Users\User\Documents\pointar.exe
Address of var[0] = 0x22feec
Value of var[0] = 10
Address of var[1] = 0x22fef0
Value of var[1] = 100
Address of var[2] = 0x22fef4
Value of var[2] = 200

Process returned 0 (0x0)   execution time : 1.207 s
Press any key to continue.
```

However, pointers and arrays are not completely interchangeable. For example, consider the following program:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    for (int i = 0; i < MAX; i++)
    {
        *var = i;    // This is a correct syntax
        var++;        // This is incorrect.
    }
    return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value.

Pointer Arrays

Before we understand the concept of array of pointers, let us consider the following example, which makes use of an array of 3 integers:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = ";
        cout << var[i] << endl;
    }
    return 0;
}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Process returned 0 (0x0)   execution time : 0.767 s
Press any key to continue.
```

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX]; // MAX represents the maximum size of the array
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr[MAX];
    for (int i = 0; i < MAX; i++)
    {
        ptr[i] = &var[i]; // assign the address of integer.
    }
    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = ";
        cout << *ptr[i] << endl;
    }
    return 0;
}
```

Output:

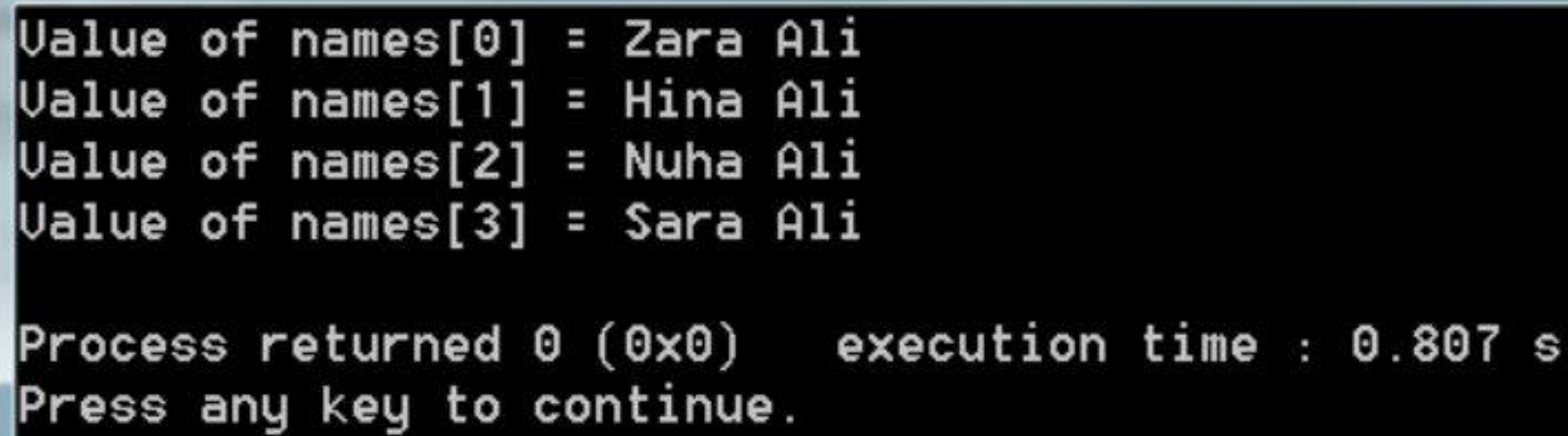
```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Process returned 0 (0x0)   execution time : 0.709 s
Press any key to continue.
```

You can also use an array of pointers to character to store a list of strings as follows: Don't worry about strings you will see in the next portion. But see how we can see pointers in strings.

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main ()
{
    char *names[MAX] = {
        "Zara Ali",
        "Hina Ali",
        "Nuha Ali",
        "Sara Ali",
    };
    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of names[" << i << "] = ";
        cout << names[i] << endl;
    }
    return 0;
}
```

Output:



```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali

Process returned 0 (0x0)   execution time : 0.807 s
Press any key to continue.
```

Exercises:

- 1) What is the difference between arrays and pointers?
- 2) What is the advantage of pointers?
- 3) Give five examples of valid pointer declarations?
- 4) Write a C++ program that accepts five values of an array, prints their value and memory address to the screen in ascending order.
- 5) What is the scope of the variable MAX in the above program and what is the advantage of the keyword **const**?