University of Baghdad
College of Science
Computer Science Department

Programming Language Fundamentals

First Class – First Semester

Morning Undergraduate Study

Dr. Nasreen Jawad Kadhim

Lecture 1

1. INTRODUCTION TO COMPUTERS

Today, computers are an integral part of our lives and are found at offices, homes, schools, colleges, hotels, shops, etc. This advance in technology has made our lives easy and comfortable. For instance, we can execute a number of activities using computer based systems.

The computer comprises of technologically advanced hardware put together to work at great speed. To accomplish its various tasks, the computer is made of different parts, each serving a particular purpose in conjunction with other parts. Computers work through an interaction of hardware (refers to the parts of a computer that you can see and touch) and software (refers to the instructions, or programs, that tell the hardware what to do).

1.2 ADVANTAGES OF COMPUTERS

Compared to traditional systems, computers offer many noteworthy advantages. This is one reason that traditional systems are being replaced rapidly by computer-based systems. The main advantages offered by computers are as follows:

- High accuracy
- Superior speed of operation
- Large storage capacity
- User-friendly features
- Portability
- Economical in the long term

1.3 TYPES OF COMPUTERS

- **Supercomputers** Are the most powerful computers in terms of speed of execution and large storage capacity. NASA uses supercomputers to track and control space explorations.
- **Mainframe Computers** Are next to supercomputers in terms of capacity. The mainframe computers are multi terminal computers, which can be shared simultaneously by multiple users. For example, insurance companies use mainframe computers to process information about millions of its policyholders.
- **Minicomputers** These computers are also known as midrange computers. These are desk-sized machines and are used in medium scale applications. For example, production departments use minicomputers to monitor various manufacturing processes and assembly-line operations.
- **Microcomputers** As compared to supercomputers, mainframes and minicomputers, microcomputers are the least powerful, but these are very widely used and rapidly gaining in popularity. These computers are designed for use by a single person. Four types of microcomputers exists: Desktop, Notebook, Tablet PC, and Handheld.

2. BASIC COMPUTER FUNCTIONING

Computer Computer can be defined as an electronic device that accepts data from an input device, processes it, stores it in a disk and finally displays it on an output device such as a monitor.

To understand the basic principles of the functioning of the computer refer to the basic block diagram of a computer as shown in Figure 1. This flow of information holds true for all types of computers such as Personal Computers, Laptops, Palmtops etc. In other words, the fundamental principle of working is the same.

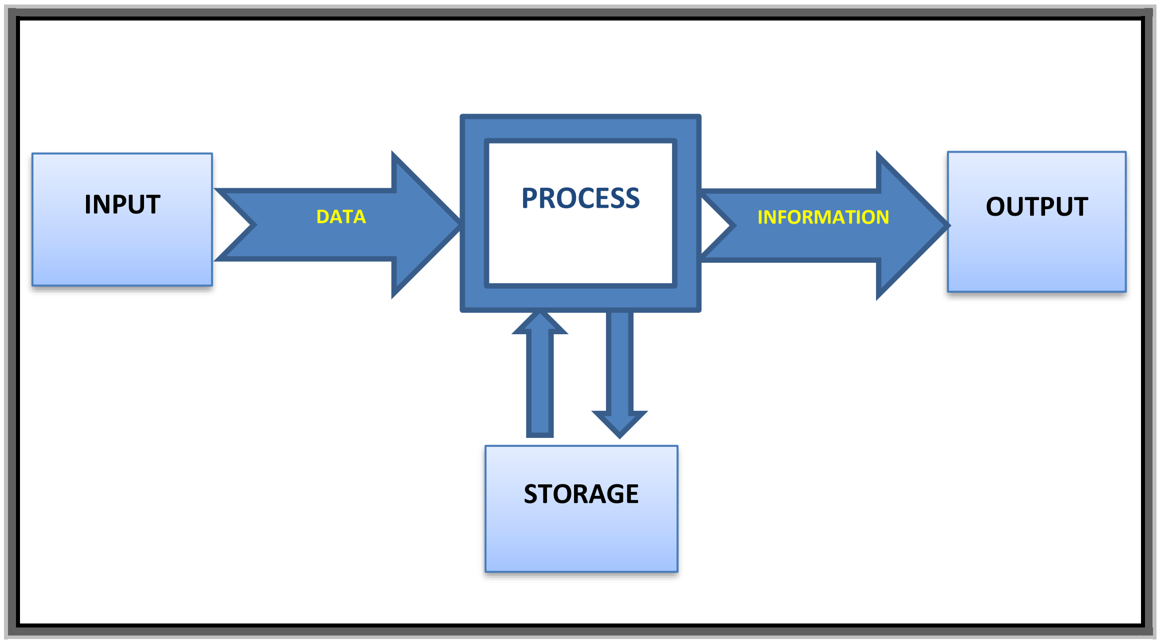


Figure 1: Information Processing Cycle**.**

As shown in Figure 1, there are four main building blocks in a computer's functioning— input, processor, output and memory. The data (***data*** is the name given to facts) is entered through input devices like the keyboard, disks or mouse. These input devices help convert data and programs into the language that the computer can process. The data received from the keyboard is processed by the CPU, i.e. the Central Processing Unit. The CPU controls and manipulates the data that produce information (***information*** is the meaningful data that is relevant, accurate, up to date and can be used to make decisions). The CPU is usually housed within the protective cartridge. The processed data is either stored in the memory or sent to the output device, as explained in the command given by the user. The memory unit holds data and program instructions for processing data. Output devices translate the processed information from the computer into a form that we can understand.

3. COMPUTER SYSTEM COMPONENTS

The omputer system consists of both hardware and software. ***Hardware components*** are the various physical components that comprise a computer system, as opposed to the nontangible software elements. The ***software components*** of a computer system are the data and the computer programs. Figure 2 illustrates the hardware and software components.

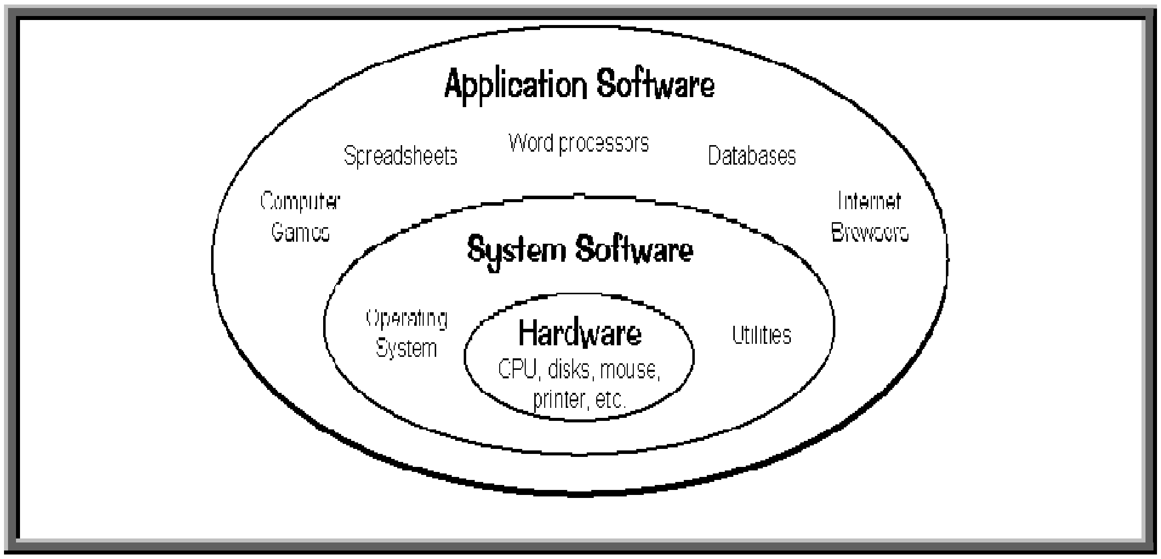


Figure 2: Basic components of computer system

3.1 Hardware Components

The main hardware components of a computer system are outlined at Figure 3:

1. Central Processing Unit

The Central Processing Unit also called Processor. This is the brain or heart of the computer equipment. The CPU carries out the calculations for the program and controls the other components of the system. It does the function by organizing circuits into two main units, called Arithmetic Logic Unit (ALU) and Control Unit (CU).

The ALU contains circuits that do arithmetic and perform logical operations. The control unit contains circuits that analyze and execute instructions.

i. Arithmetic Logic Unit

The ALU contains logic circuits that perform logic operations as well as arithmetic circuits that can subtract multiply and divide two numbers. More complex operations such as finding the square root of a number are done by sequence of their basic operations. The

ALU has storage locations called storage registers for storing numbers used in calculations and for storing the results of calculations. To perform a calculation or logical operation, number is transferred from primary storage to storage registers in the ALU. These numbers are sent to the appropriate arithmetic or logic circuit. The results are sent back to the storage registers. The results are transferred from the storage registers to primary storage.

ii. Control Unit

The Control Unit (CU) controls the whole computer system via performing the following functions:

- Directs and coordinates all operations called for by the program.
- Activates the appropriate circuits necessary for input and output devices.
- Causes the entire computer system to operate in an automatic manner.

The CU contains a temporary storage location called an instruction register for storing the instruction being executed. It also contains circuit called the instruction decoder –which analyzes the instruction register and causes it to be executed. The CU executes each instruction by following the same basic sequence of steps:

- The next instruction in the program is retrieved for primary storage and stored in the instruction register.
- The instruction is sent to the instruction decoder where it is analyzed.
- The decoder sends signals to the ALU, primary storage, I/O devices, and secondary storage, that cause the actions required by the instructions to be performed. These steps are repeated for each instruction in the program until all instructions have been executed.

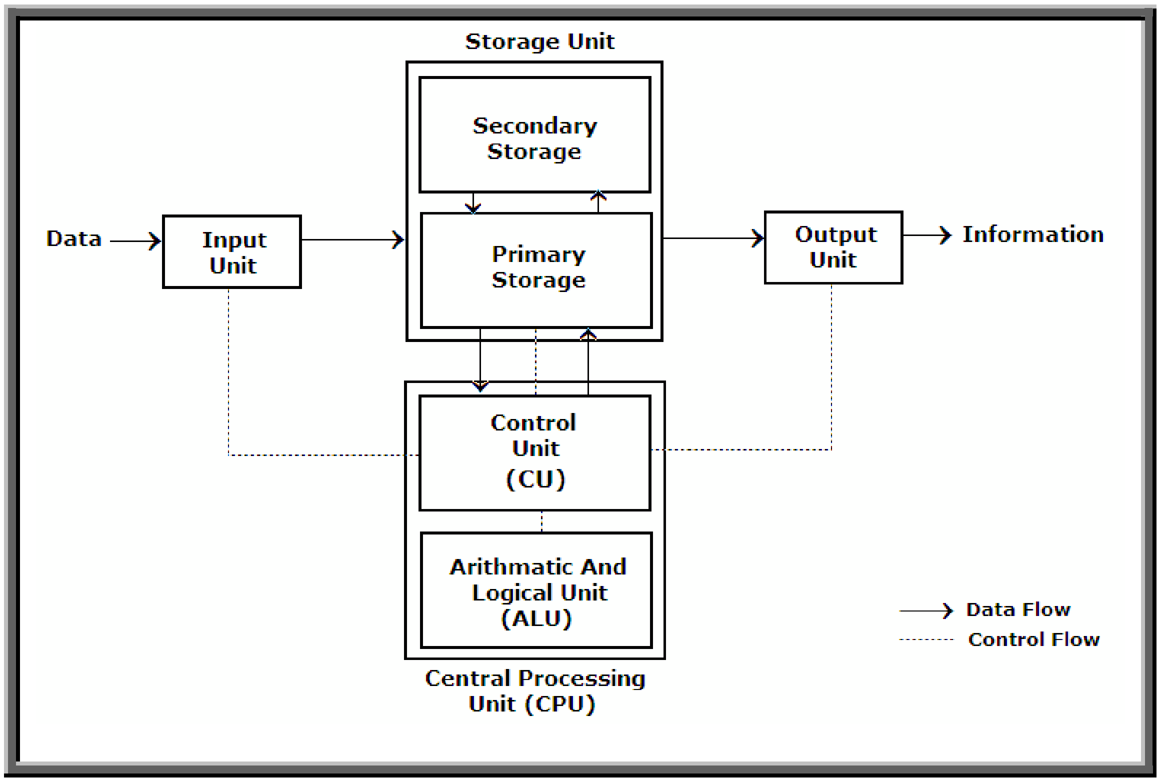


Figure 3: Block diagram of computer system.

2. Storage Unit

i. Primary Storage

Primary storage is also called internal storage or memory. It is used to store programs and data currently being processed by CPU. Primary storage circuits need electricity to stay on. If the power to the computer is turned off, all the circuits will turn off and all data in primary storage will be lost. When computer is turned back on, the data will not reappear. The data is lost forever. Because of this characteristics primary storage is called volatile storage. This type of primary storage is called RANDOM ACCESS MEMORY or RAM. RAM is the main type of primary storage used with computers and it is volatile. Many computers have another type of primary storage called ROM –Read Only Memory. ROM is non-volatile storage. ROM can store preset programs that are always put by computer manufactures.

<u>The Use of Primary Storage (RAM)</u>

Main Memory has several uses:

- Input area –where the data is stored when it is read into CPU, awaiting processing.
- Operating system –controls the operation of the computer.
- Working storage –where calculations are performed and data is stored temporarily.
- Output area where the information is stored prior to output. Both the input and output areas are buffers.
- Application program area –where the user program is held.

ii. Secondary Storage

Secondary is nonvolatile. Any data or programs stored in secondary storage stays there, even with the computer power turned off. Secondary storage is a permanent form of storage.

Differences between Primary storage and Secondary storage are explained at Table 1.

Table 1: Difference between Primary and Secondary Storage

Primary Storage	**Secondary Storage**
Primary storage/memory is also called Internal or main storage. High speed	Secondary storage or memory is also called External or auxiliary storage. Low speed
It is very expensive	It is not expensive as primary storage
It holds data or programs temporary	It holds data or programs permanently
It holds programs and data in current use in CPU	It holds program or data that will still be used in primary storage
It is faster than secondary	It is not fast as primary
It holds less data	It holds large volume of data or files

3. Input and Input Devices

Input is any data or instructions that are used by a computer. Input devices are hardware used to translate words, sounds, images, and actions that people understand into a form that the system unit can process. Some commonly used input devices are: Keyboard, Mouse, and Scanner.

4. Output and Output Devices

Output is processed data or information, and typically takes the form suitable for use by the computer's human operators such as text, graphics, photos, audio, and/or video. Output devices are any hardware used to provide or to create output. They translate information that has been processed by the system unit into a form that humans can understand. Some commonly used output devices are: Monitors (soft copy) and Printers (hard copy).

Lecture 2

3.2 Computer Software

Or just **software**, is the collection of computer programs and related data that provide instructions telling a computer what to do. The term was coined to contrast the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched".

There are two types of software -***system software*** and ***application/utility software***.

- **Application Software** is the end user software. It is a computer software designed to help the user to perform singular or multiple related specific tasks. The programs written under application software are designed for general purpose and special purpose applications. An example of application software is Microsoft Internet Explorer.

- **System Software** is computer software designed to provide services to other software. It enables an application software to interact with the computer hardware. System software is the ‘background’ software that helps the computer to manage its internal resources. The most important system software is the operating system. Microsoft Windows are examples of system software.

4. PROGRAMMING LANGUAGES

Different programming languages support different styles of programming (called programming paradigms). The choice of language used is subject to many considerations such as company policy, suitability to task, availability of third-party packages or individual preference. Ideally, the programming language best suited for the immediate task will be selected.

Programming language is a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. The term *programming language* usually refers to high-level languages, such as C, C++, C#, and Java. Each language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

High-level programming languages, while simple compared to human languages, are more complex than the languages the computer actually understands, called ***machine languages***. Each different type of CPU has its own unique machine language. Lying between machine languages and high-level languages are languages called assembly languages. ***Assembly languages*** are similar to machine languages, but they are much easier to program in because they allow a programmer to substitute names for numbers. Machine languages consist of numbers only.

Lying above high-level languages are languages called ***fourth-generation languages*** (usually abbreviated *4GL*). 4GLs are far removed from machine languages and represent the class of computer languages closest to human languages. Each of the programming language generations aims to provide a higher level of **abstraction** of the internal **computer hardware** details, making the language more programmer-friendly, powerful and versatile. Languages claimed to be 4GL may include support for **database management, report generation, mathematical optimization, GUI development,** or **web development.**

A **fifth generation programming language** (abbreviated as **5GL**) is a programming language based on solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most **constraint-based** and **logic** programming languages and some **declarative languages** are fifth-generation languages. While **fourth-generation programming languages** are designed to build specific programs, fifth-generation languages are designed to make the computer solve a given problem without the programmer. This way, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in **artificial intelligence** research. **Prolog, OPS5,** and **Mercury** are examples of fifth-generation languages.

Regardless of what language you use, you eventually need to convert your program into machine language so that the computer can understand it. There are two ways to do this:

1) ***Compile*** the program.

2) ***Interpret*** the program.

Compiler and ***interpreter*** are the internal programs that translate High level language to Machine language. ***Compiler*** takes **Entire** program as input, program need not be **compiled** every time, **errors** are displayed after **entire program** is checked, and it is **fast.** Whereas ***interpreter*** takes **Single** instruction as input, every time higher level program is converted into lower level program, **errors** are displayed for **every instruction** interpreted (if any), and it is **slow**.

3) For low level languages, the internal program that translates assembly language program (op code) to machine code is known as ***Assembler***.

Every language has its strengths and weaknesses. For example, Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features, but it is complex and difficult to learn. The choice of which language to use depends on the type of computer the program is to run on, what sort of program it is, and the expertise of the programmer.

5. PROGRAM DEVELOPMENT LIFE CYCLE

The process of developing a software, according to the desired needs of a user, by following a basic set of interrelated procedures is known as Program Development Life Cycle (PDLC). PDLC includes various set of procedures and activities that are isolated and sequenced for learning purposes but in real life they overlap and are highly interrelated.

5.1 TASKS OF PROGRAM DEVELOPMENT LIFE CYCLE

The basic set of procedures that are followed by various organizations in their program development methods are as follows:

1. **Program Analysis (Problem Definition),**
2. **Program Design,**
3. **Coding,**
4. **Debugging,**
5. **Testing,**
6. **Documentation,**
7. **Maintenance,**
8. **Extension and Redesign.**

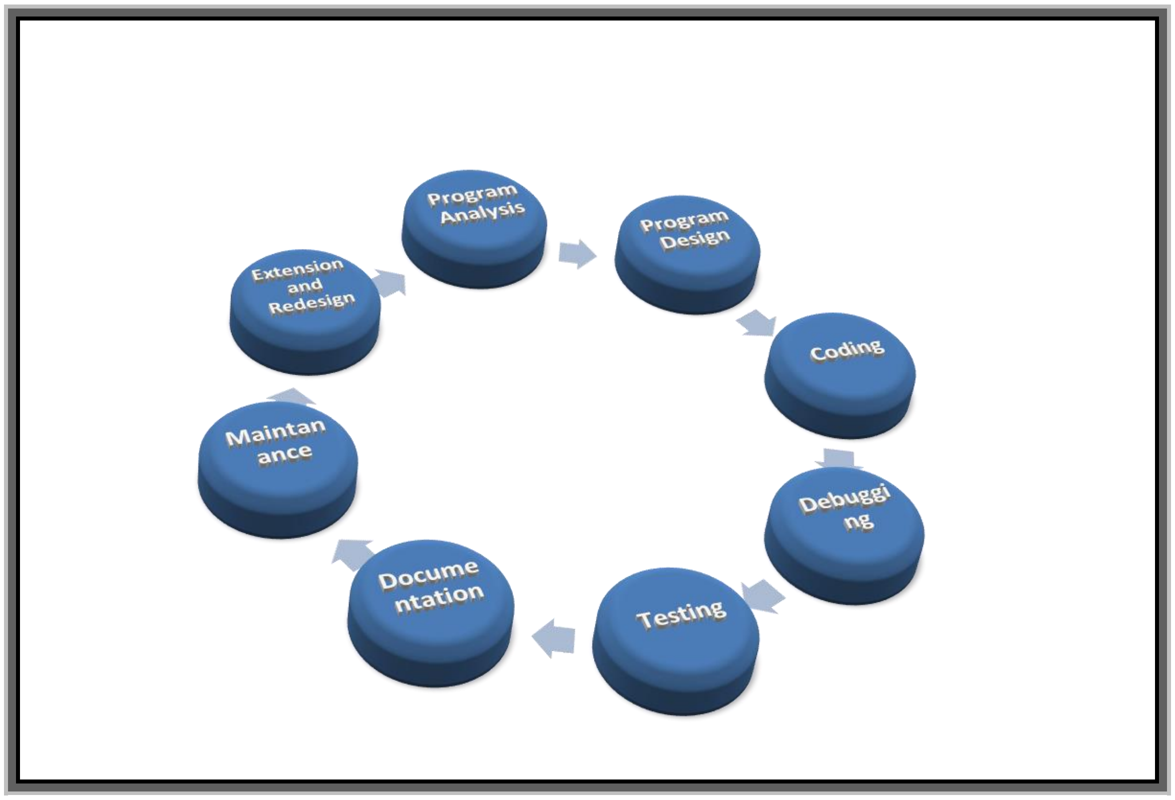


Figure 4 Program Development Life Cycle.

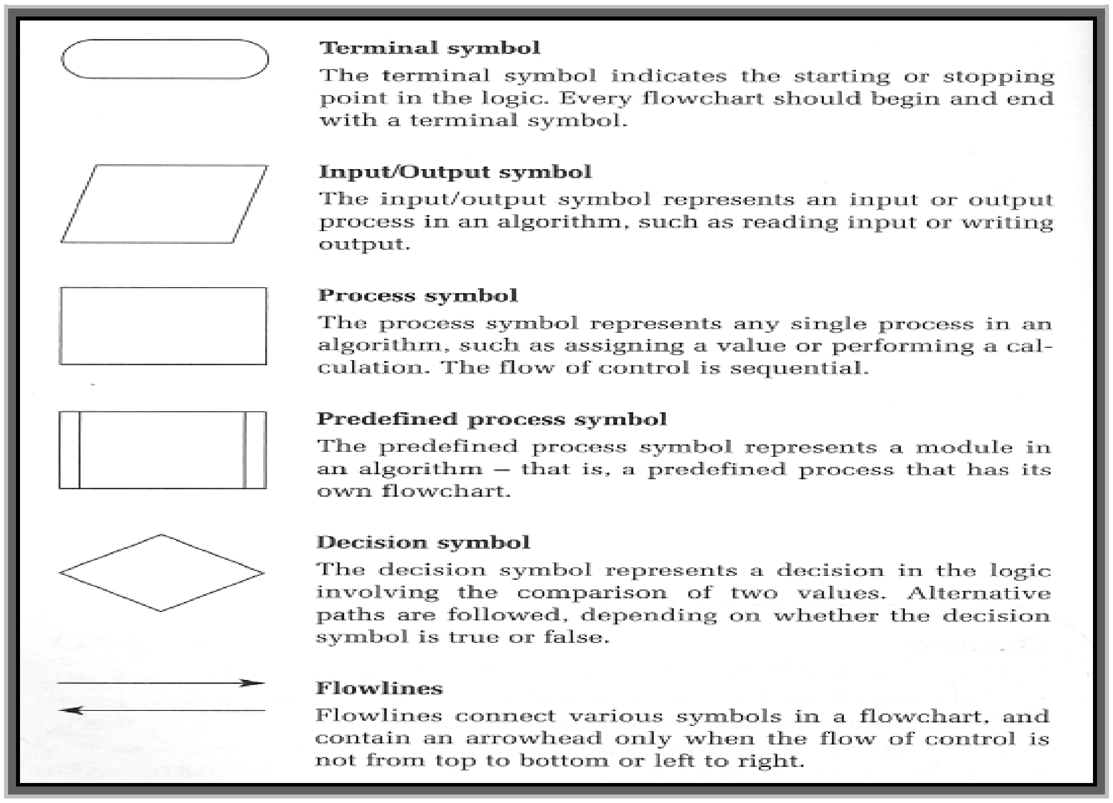
1. Program Analysis (Problem Definition):

This stage is the formal definition of the task. It includes the specification of inputs and outputs, processing requirements, system constraints, and error handling methods. This step is very critical for the completion of a satisfactory program. It is impossible to solve a problem by using a computer, without a clear understanding and identification of the problem. Inadequate identification of problem leads to poor performance of the system. The programmer should invest a significant portion of his time in problem identification. If he does not spend enough time at this stage, he may find that his well written program fails to solve the real problem. This step is the process of becoming familiar with the problem. It starts when the programmer is assigned a task. This step includes the reviewing of the design document that was prepared for the program, as well as any system wide information that would be helpful. The process ends when all the programmer's questions have been resolved and the requirements of the program are understood.

2. Program Design

Once the problem has been identified, the next stage is the program design. A computer is both fast and versatile, but it requires the careful specification of what actions it should take. For the user, there is seldom an opportunity to allow the computer to make an undirected decision. Therefore the programmer must decide, prior to writing his program, exactly which steps the computer should take to solve an identified problem. Such a functional description of the task is either called an algorithm or results in a diagram called flowchart.

Flowchart definition: Is a diagrammatic representation that illustrates a solution model to a given problem. It represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields. Figure 5 illustrates the basic symbols of the flowchart.



Terminal symbol
The terminal symbol indicates the starting or stopping point in the logic. Every flowchart should begin and end with a terminal symbol.

Input/Output symbol
The input/output symbol represents an input or output process in an algorithm, such as reading input or writing output.

Process symbol
The process symbol represents any single process in an algorithm, such as assigning a value or performing a calculation. The flow of control is sequential.

Predefined process symbol
The predefined process symbol represents a module in an algorithm — that is, a predefined process that has its own flowchart.

Decision symbol
The decision symbol represents a decision in the logic involving the comparison of two values. Alternative paths are followed, depending on whether the decision symbol is true or false.

Flowlines
Flowlines connect various symbols in a flowchart, and contain an arrowhead only when the flow of control is not from top to bottom or left to right.

Figure 5 The basic symbols of the flowchart.

3. Coding

The third step is the process of transforming the program logic design documents into a computer language format. This stage translates the program design into computer instructions. These instructions are the actual program or the software product. During this step the programmer eliminates all the syntax and formal errors from the program and all logic errors are detected and resolved during this process.

Lecture 3

4. Debugging

This stage is the discovery and correction of programming errors. Few programs run correctly the first time, so debugging is an important and time consuming stage of software development. Programming theorists often refer to program debugging and testing as

verification and validation, respectively. Verification ensures that the program does what the programmer intends to do. Validation ensures that the program gives the correct results for a set of test data.

There are basically three types of errors that you must contend with when writing computer programs:

i. Compile-time errors or Syntax errors (Error(s) in spelling and grammar):

- ✓ Occur if there is a syntax error in the code.
- ✓ The compiler will detect the error and the program won’t even compile. At this point, the programmer is unable to form an executable program that a user can run until the error is fixed.
- ✓ A compilation error message often helps programmers debugging the source code for possible syntax errors.
- ✓ Common examples are:

1) Misspelled variable and function names.

2) Missing semicolons (;).

3) Improperly matches parentheses.

ii. Run-time errors:

- ✓ Compilers aren’t perfect and so can’t catch all errors at compile time.
- ✓ Run-time error - refers to the period while a computer program is actually executed ("run") in a computer, from beginning to termination.
- ✓ Common examples are:

1) Trying to divide by a variable that contains a value of zero.

2) Trying to do impossible arithmetic operations such as calculation with non-numeric data.

3) Trying to open a file that doesn't exist.

iii. Logical errors:

- ✓ Errors that indicate the logic used when coding of the program failed to solve the problem (i.e. production of wrong solutions).
- ✓ You do not get error messages with logical errors.
- ✓ Common examples are:

1) Multiplying when you should be dividing.

2) Adding when you should be subtracting.

3) Opening and using data from the wrong file.

4) Displaying the wrong message.

5. Testing

This stage is the ***validation*** of the program. Testing ensures that the program performs correctly the required tasks. Program testing and program debugging are closely related. Testing is essentially a later stage of debugging in which the program is validated by trying it on a suitable set of cases. Program testing is, however, more than a simple matter of exercising the program a few times. Exhaustive testing of all possible cases is the best alternative, but this processes is usually impractical. Formal validation methods exist, but are only applicable to very simple programs. Thus, program testing requires a choice of test cases.

There are two goals in preparing a test plan. Firstly, a properly detailed test plan demonstrates that the program specifications are understood completely. Second, the test plan is used during the program testing to prove the correctness of the program. During this step a general approach to the testing of the program is prepared and documented, indicating the number of tests needed and the purpose of each test.

6. Documentation

This stage is the documentation of the program so that those who use and maintain it, can understand it, and program can be extended to further applications. Documentation is a stage of software development that is often overlooked. Yet proper documentation is not only useful in the testing and debugging stages, it is also essential in the maintenance and redesign stages. A properly documented program can be easily reused when needed; an undocumented program usually requires so much extra work that the programmer might as well start from scratch. Among the techniques commonly used in documentation are flowcharts, comments, memory maps, parameter and definition lists, and program library forms. Proper documentation combines all or most of the methods mentioned. Documentation is a time consuming task that the programmer performs simultaneously with the design, coding, debugging and testing stages of software development. Good documentation simplifies maintenance and redesign, and makes subsequent tasks simpler.

7. Maintenance

After the programs are developed and documented, it is placed into operation. During the operation, a program may fail to perform its objective and it might be necessary to add new functionality to a program or system. Changing program design, coding, and updating programs are part of maintenance.

8. Extension and Redesign

This stage is the extension of the program to solve problems beyond those described in initial problem definition. Obviously designers want to take advantage of programs developed for previous tasks. Redesign may involve adding new features or meeting new requirements. Such redesign should proceed through the previously mentioned stages of program development. The process may also involve making a program meet critical time or memory requirements. When relatively small increase in speed or reduction in memory usage is needed, a program can often be reorganized to meet the requirement.

Example 1:

Read two numbers A and B and compute The sum of them and print the result.

Input to the program:

Two numbers (A, and B).

Processing:

Compute Sum as: SUM = A+B

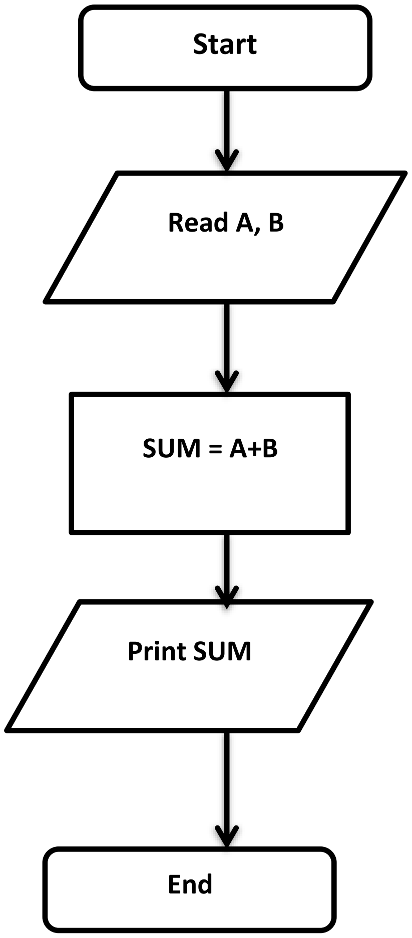
Program output:

The sum of the numbers (SUM).

Algorithm:

```
1- Start
2- Read A, and B
3- Compute the Sum (SUM):
     SUM = A+B
4- Print the Sum (SUM)
5- End
```

Flowchart:



Lecture 4

6. PROGRAM FLOW

Program is defined as a sequence of statements whose objective is to accomplish some task. Program statements can be processed in one of the following ways:

- **Sequentially (Simple Sequential Flowcharts):** Starts at the beginning and follows the statements in order until it comes to the end. No choices are made; there is no repetition.
- **Selectively (Branched Flowcharts):** By making a choice or decision when executing statement/statements, which is also called a branch.

- **Repetitively (Loop Flowcharts):** The program repeats particular statements a certain number of times based on some condition(s).

Every programming language provides constructs to support sequence, selection or iteration. So there are three programming language constructs. Figure 6 illustrates these three types.

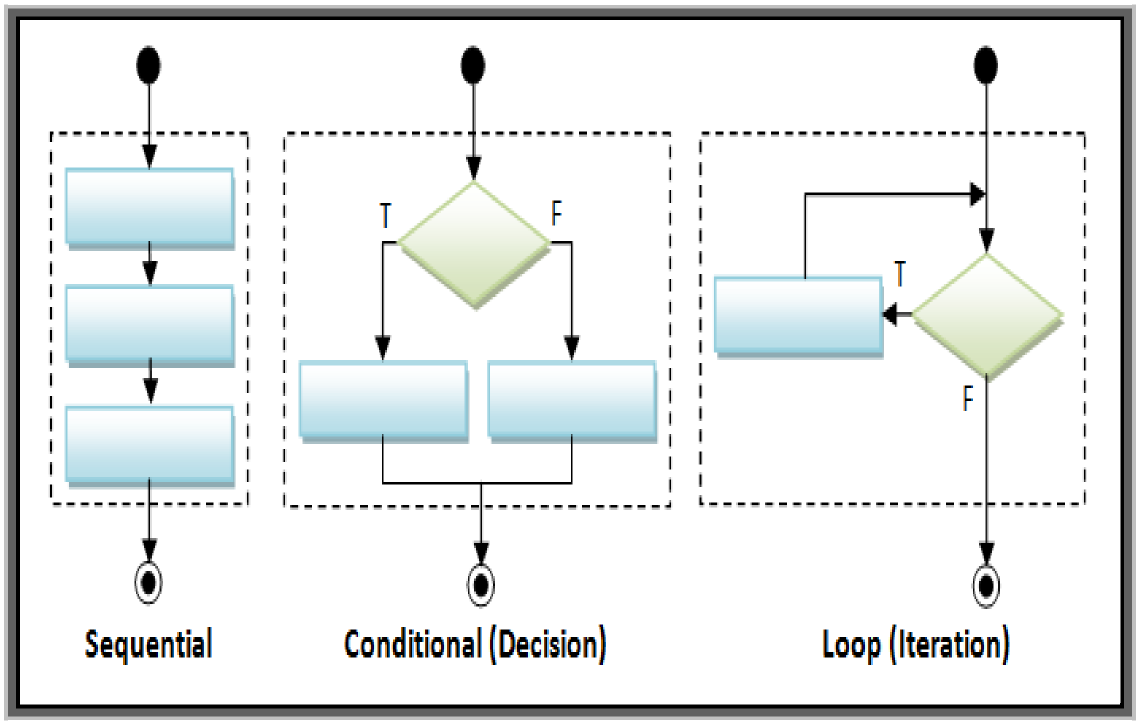
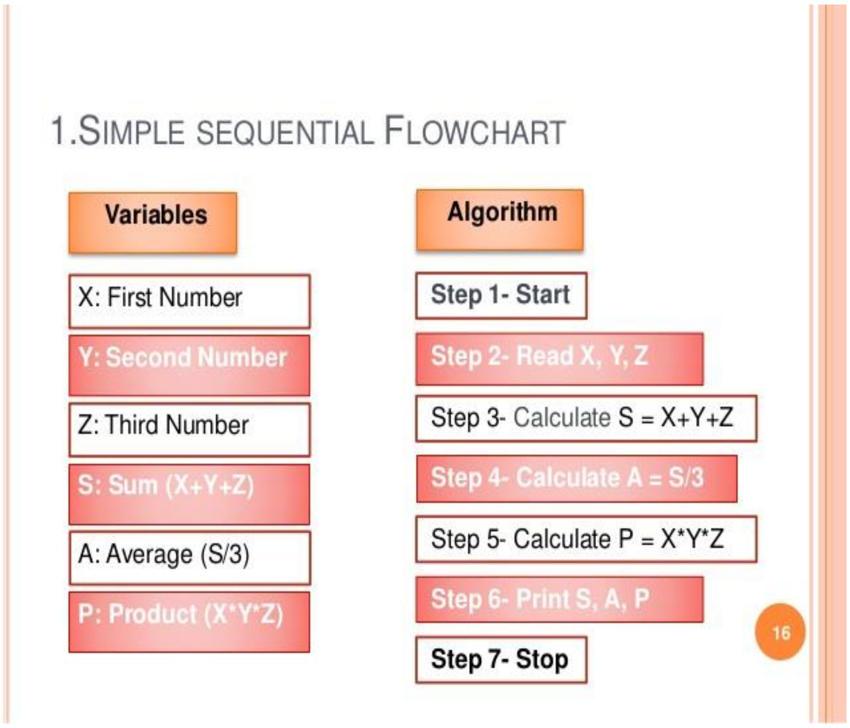


Figure 6 The types of program flow.

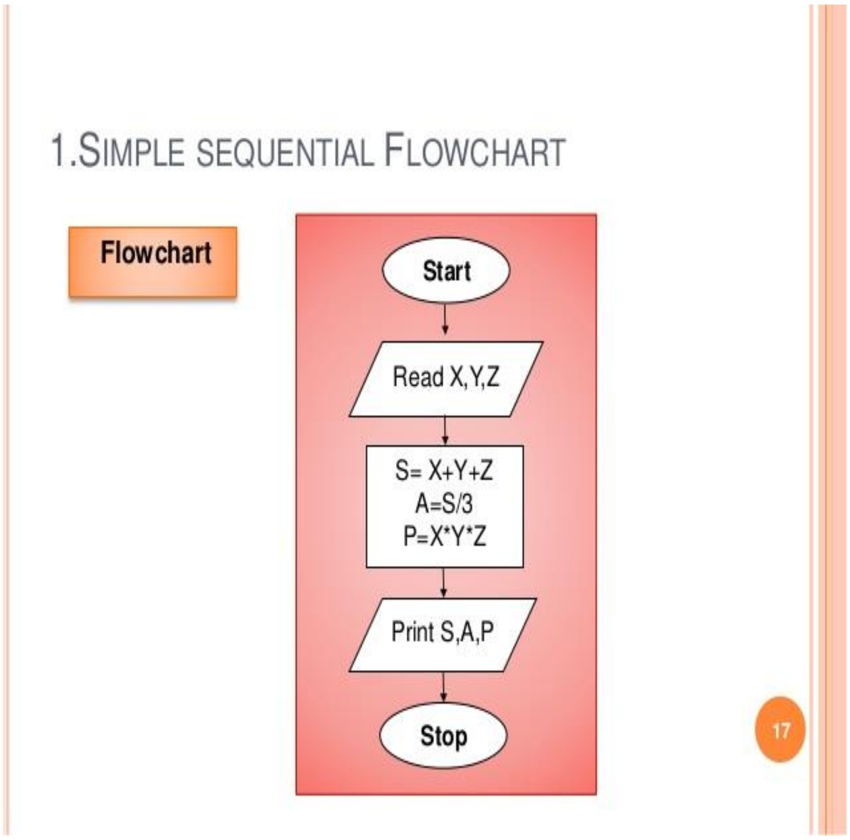
Simple Sequential Flowchart: Example 1:

Construct a flowchart that finds sum, average and product of three numbers.



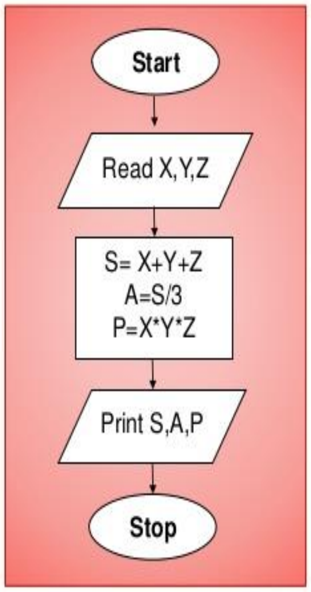
1.Simple sequential Flowchart

Variables	Algorithm
X: First Number	Step 1- Start
Y: Second Number	Step 2- Read X, Y, Z
Z: Third Number	Step 3- Calculate S = X+Y+Z
S: Sum (X+Y+Z)	Step 4- Calculate A = S/3
A: Average (S/3)	Step 5- Calculate P = X*Y*Z
P: Product (X*Y*Z)	Step 6- Print S, A, P
	Step 7- Stop



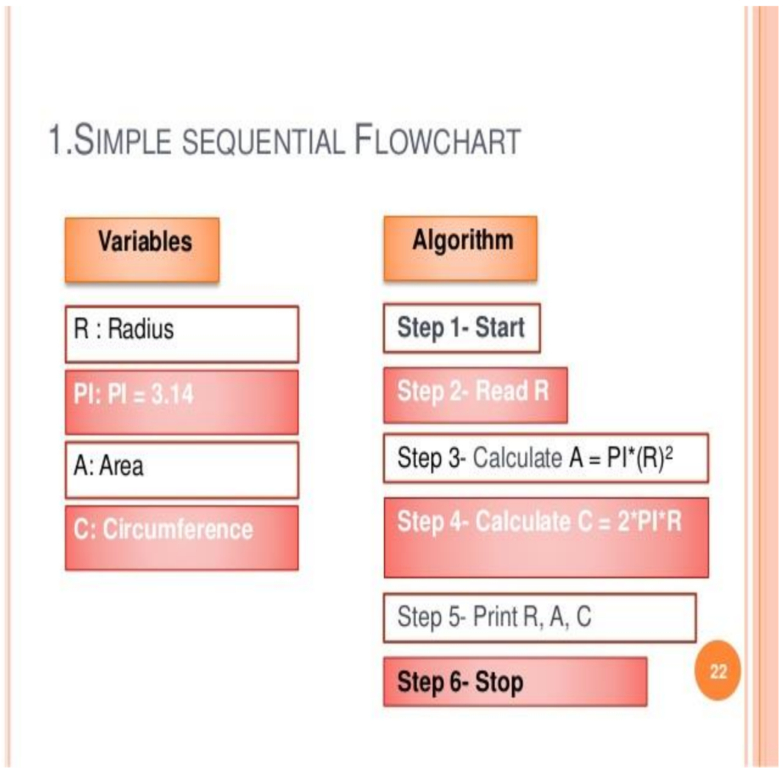
1.Simple sequential Flowchart

Flowchart



Simple sequential flow: Example 2:

Construct a flowchart that finds the area and circumference of a circle where R (radius) is given.

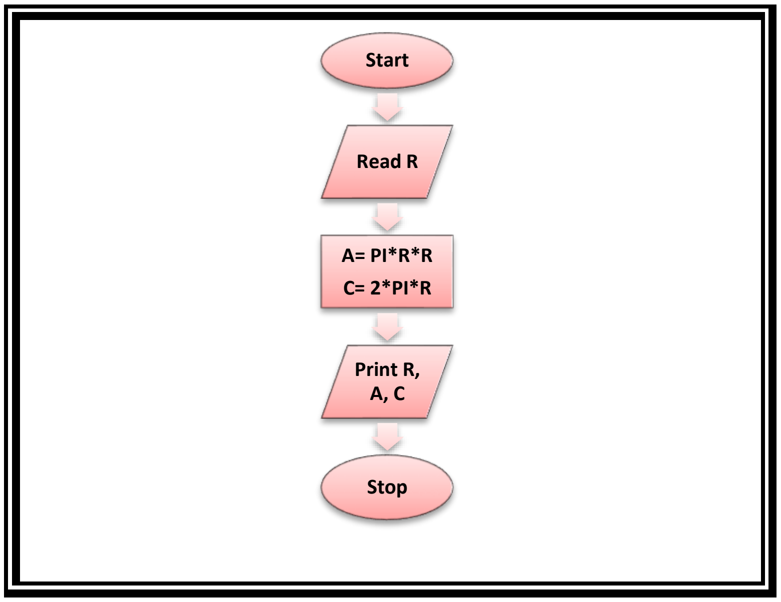


1.Simple sequential Flowchart

Variables	Algorithm
R : Radius	Step 1- Start
PI: PI = 3.14	Step 2- Read R
A: Area	Step 3- Calculate A = PI*(R)^2
C: Circumference	Step 4- Calculate C = 2*PI*R
	Step 5- Print R, A, C
	Step 6- Stop

22

Flowchart:



2. Branched Flowcharts

Example 1

Construct a flow chart for the following function

$$F(x) = \begin{cases} X & X>=0 \\ -X & X<0 \end{cases}$$

2. Branched Flowcharts

Variables

X : Number

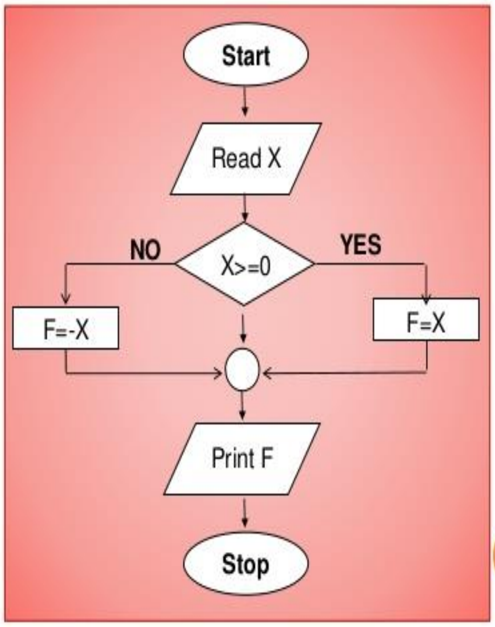
F: function of X

Algorithm

```
Step 1- Start
Step 2- Read X
Step 3- if X >=0 then F =X
Step 4- if X <0 then F =-X
Step 5- Print F
Step 6- Stop
```

2. Branched Flowcharts

Flowchart



2. Branched Flowcharts

Example 2

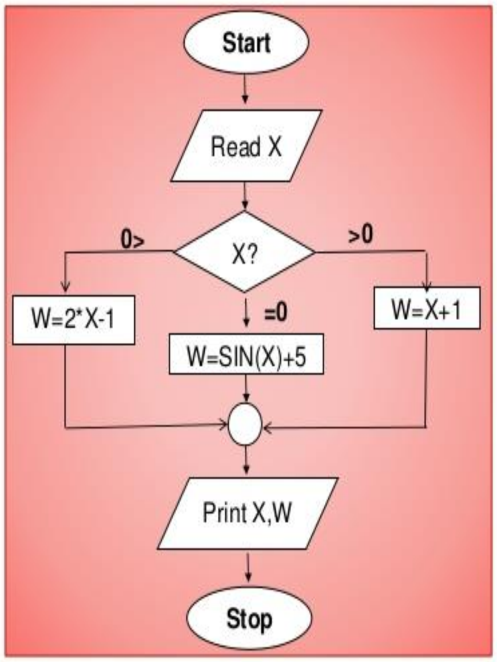
Trace the following flowchart and write the output of it.

1. When X = 20

2. When X = -10

2. Branched Flowcharts

Flowchart



2. Branched Flowcharts

Result

When X=20

X= 20
W= 21

When X=-10

X= -10
W= -21

2. Branched Flowcharts

Example 3

Exercise

Draw a flowchart that shows the traffic light processing

2. Branched Flowcharts

Variables

C : Traffic light color

Algorithm

```
Step 1- Start
Step 2- Read C
Step 3- make a Decision (what is c)
Step 4- if C is RED then Print STOP
Step 5- if C is YELLOW then Print WAIT
Step 6- if C is GREEN then Print PASS
Step 7- Stop
```

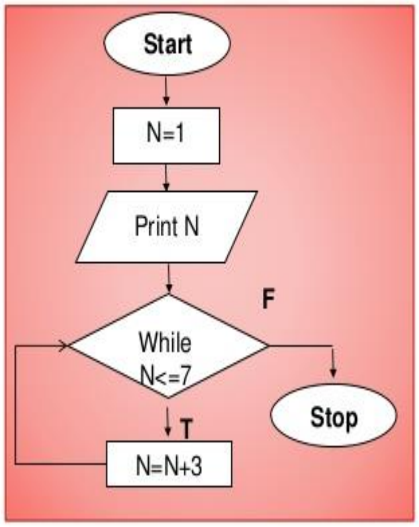
3. Loop Flowcharts

Example 1

Trace the following flowchart and write the output of it.

3. Loop Flowcharts

Flowchart



3. Loop Flowcharts

Result

N	Loop
1	1
4	2
7	3

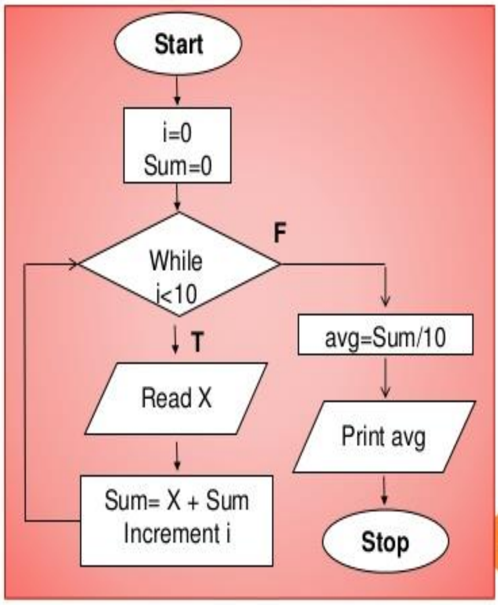
3. Loop Flowcharts

Example 2

Trace the following flowchart and write the output of it.

3. Loop Flowcharts

Flowchart



3. Loop Flowcharts

Result

Loop	Read X	Sum	i
1	3	3	1
2	4	7	2
3	1	8	3
4	10	18	4
5	7	25	5
6	5	30	6
7	3	33	7
8	8	41	8
9	4	45	9
10	5	50	10

Avg =50/10 =5

Lecture 5

7. The Process of Writing a C++ Program

- Programming: is a process of problem-solving.
- Different people use Different techniques to solve problems.
- Some techniques are nicely outlined and easy to follow. They not only solve the problem, but also give insight into how the solution was reached.
- These problem-solving techniques can be easily modified if the domain of the problem changes.
- To be a good problem solver and a good programmer, you must follow good problem-solving techniques.

Algorithm: a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

In a programming environment (PDLC), the problem-solving process requires, after the problem has been defined, the following Three basic steps:

1. Analyze the problem and outline the problem and its solution requirements.

2. Design an algorithm to solve the problem.

3. Implement the algorithm in a programming language, such as C++, and maintain the program by using and modifying it if the problem domain changes.

So, To develop a program to solve a problem, you start by analyzing the problem. You then design the algorithm; write the program instructions in a high-level language (or code the program); and enter the program into a computer system. Figure 7 summarizes this three-step programming process:

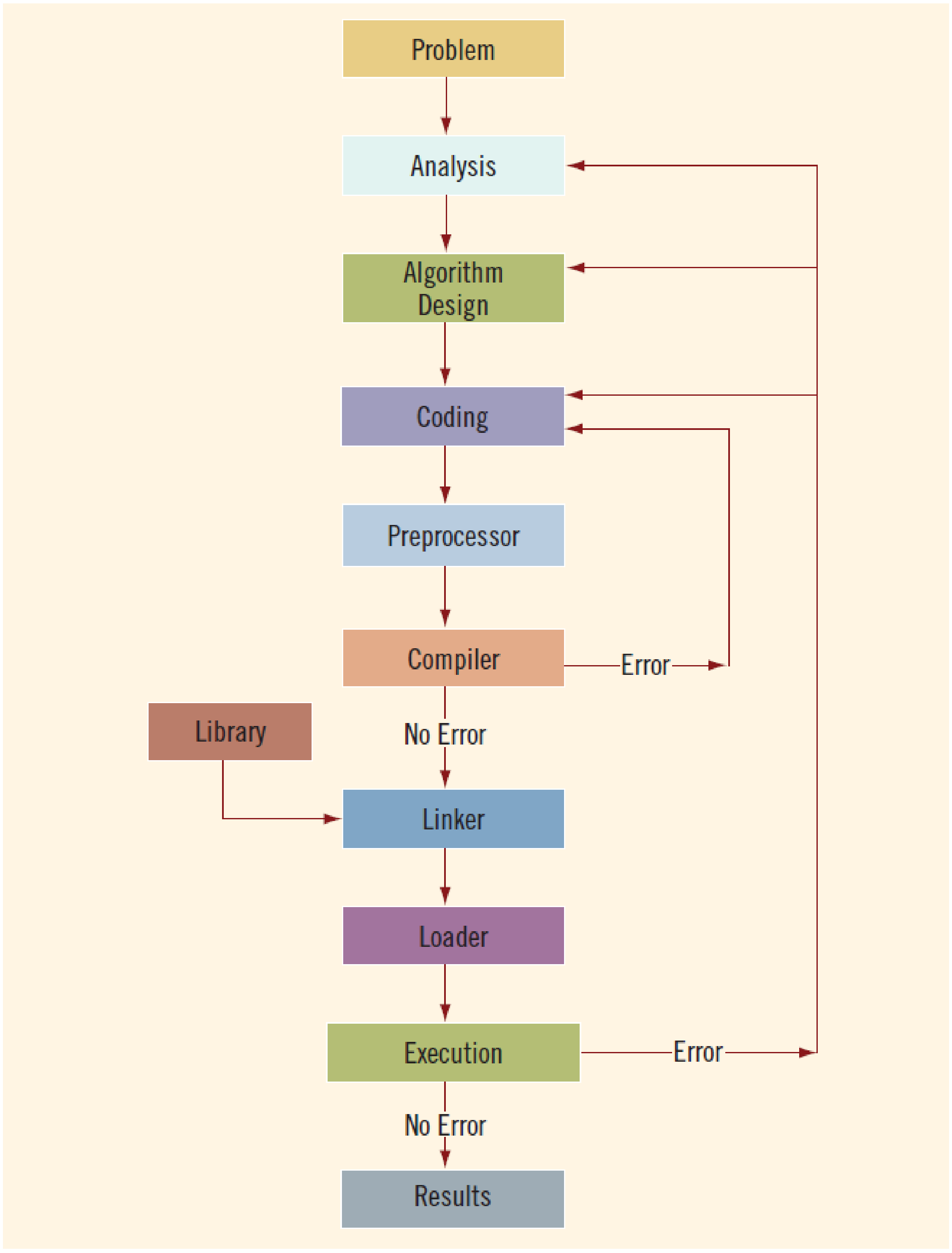


Figure 7 Problem analysis–coding–execution cycle

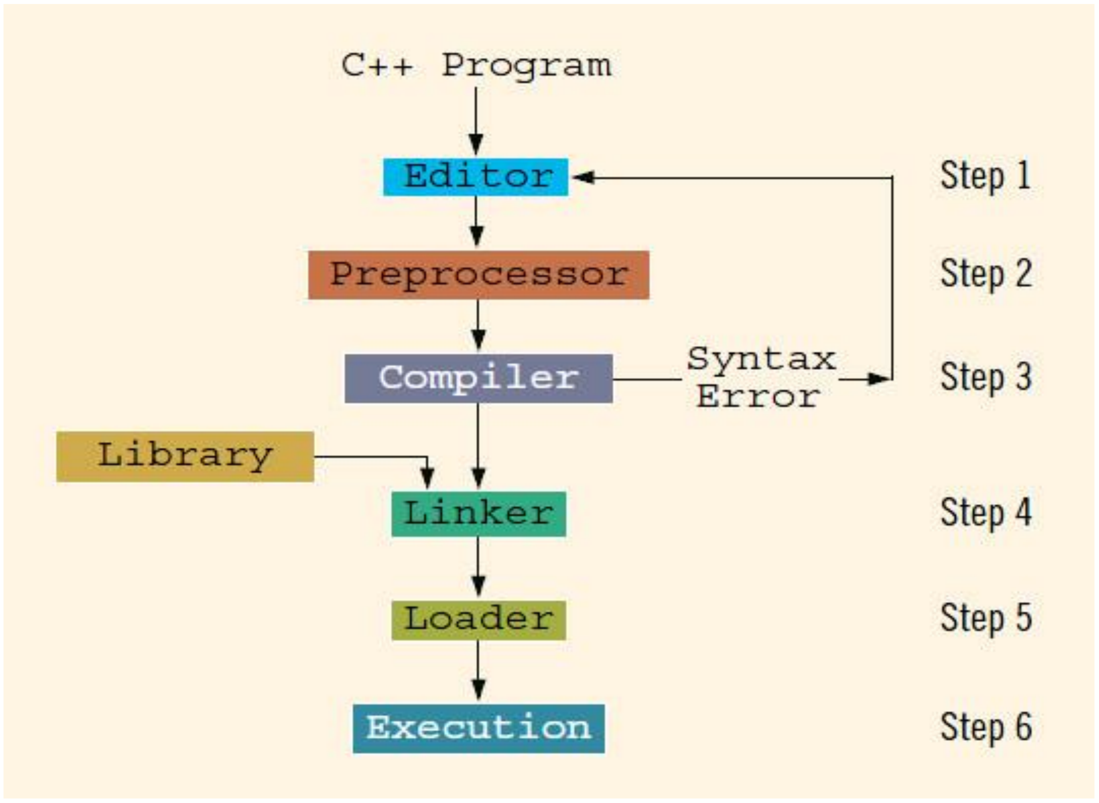


Figure 8 Processing a C++ program

As illustrated in Figure 8, we are ready to review the steps (**Six Steps**) required to process a program written in C++.

Step 1: Write the source codes (.cpp) and header files.

Step 2: Pre-process the source codes according to the *preprocessor directives*. Preprocessor directives begin with a hash sign (#), e.g., #include and #define. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.

Step 3: Compile the pre-processed source codes into object codes (.obj, .o).

Step 4: Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).

Step 5: Load the executable code (.exe) into computer memory.

Step 6: Run the executable code, with the input to produce the desired output.

- ✓ Source Program (.cpp): consists of the program statements comprising a C++ or other programming language program.
- ✓ Object Program (.obj): is the result of compiling a Source Program.
- ✓ Header files: contain constant, variable, and function declarations needed by a program.
- ✓ Linker: A program that combines the Object Program (.obj) with other programs in the library and is used in the program to create the executable code.
- ✓ Loader: A program that loads an Executable Program (.exe) into main memory.
- ✓ Executable Program (.exe): is a program that can be run by a computer.

Lecture 6

7.1 Compiling and Linking Errors

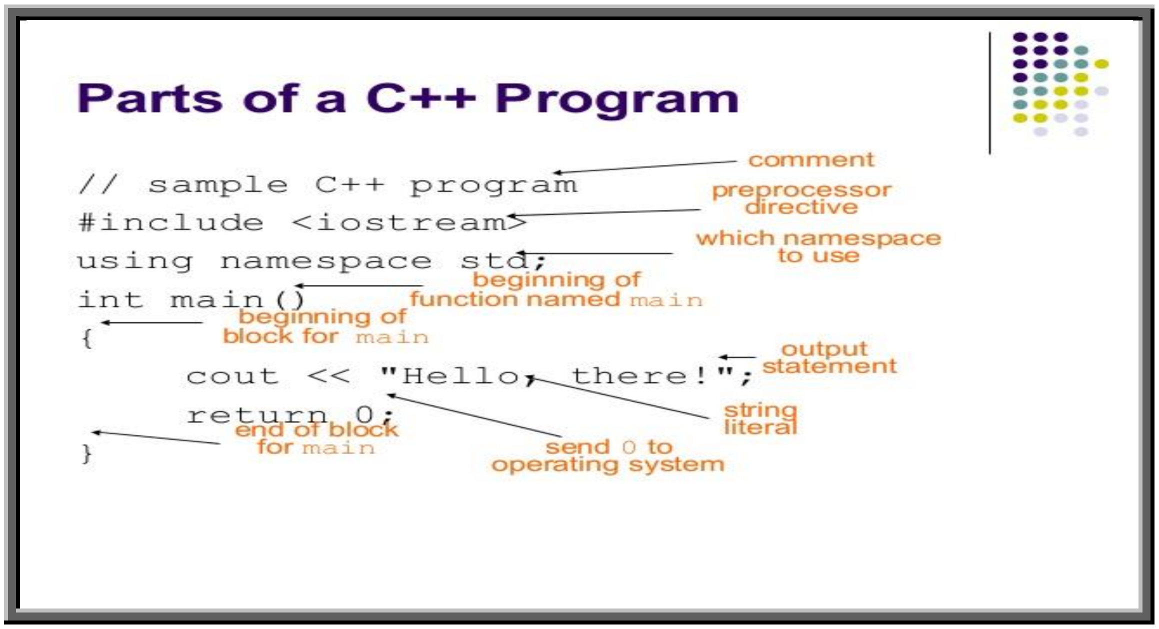
During compilation, if there are any errors that will be listing by the compiler. The errors may be any one of the following:

- **Syntax error** This error occurs due to mistake in writing the syntax of a C++ statement or wrong use of reserved words, improper variable names, using variables without declaration etc. Examples are: missing semi colon or parenthesis, etc. Appropriate error message and the statement number will be displayed. You can see the statement and make correction to the program file, save and recompile it.
- **Logical error** This error occurs due to the flaw in the logic. This will not be identified by the compiler. However it can be traced using the debug tool in the editor. First identify the variable which you suspect creating the error and add them to watch list. This tool helps in knowing the values taken by the variable at each step. You can compare the expected value with the actual value to identify the error.

- **Linker error** This error occur when the files during linking are missing or misspelt.
- **Runtime error** This error occurs if the program encounters division by zero, accessing a null pointer etc. during execution of the program.

7.2 Parts of C++ Program: Brief Explanation

See the following C++ program:



Parts of a C++ Program

```
// sample C++ program
#include <iostream>
using namespace std;
int main()
{
      cout << "Hello, there!";
      return 0;
}
```

Comments

/* ...... */

// ... until the end of the line

These are called *comments*. Comments are NOT executable and are ignored by the compiler; but they provide useful explanation and documentation to your readers. There are two kinds of comments:

1. *Multi-line Comment*: begins with `/*` and ends with `*/`. It may span more than one line.

2. *End-of-line Comment*: begins with `//` and lasts until the end of the current line.

Preprocessor directives

#include <iostream>

using namespace std;

The "`#include`" is called a *preprocessor directive*. Preprocessor directives begin with a `#` sign. They are processed before compilation. The directive "`#include <iostream>`" tells the preprocessor to include the "`iostream`" header file to support input/output operations. The "`using namespace std;`" statement declares `std` as the *default namespace* used in this program. The names `cout` and `endl`, which is used in this program, belong to the `std` namespace. These two lines shall be present in all our programs.

Main function

int main() { ... *body* ... }

defines the so-called `main()` *function*. The `main()` function is the *entry point* of program execution. `main()` is required to return an `int` (integer).

cout << "hello, world" << endl;

"`cout`" refers to the standard output (or Console OUTput). The symbol `<<` is called the *stream insertion operator* (or *put-to operator*), which is used to put the string "hello, world" to the console. "`endl`" denotes the END-of-Line or newline, which is put to the console to bring the cursor to the beginning of the next line.

Return Statement

return 0;

terminates the `main()` function and returns a value of `0` to the operating system. Typically, return value of 0 signals normal termination; whereas value of non-zero (usually 1) signals abnormal termination. This line is optional. C++ compiler will implicitly insert a "`return 0;`" to the end of the `main()` function.

Lecture 7

8. Programming Methodologies

Two popular approaches to programming design are:

1- Structured approach (Structured Programming)

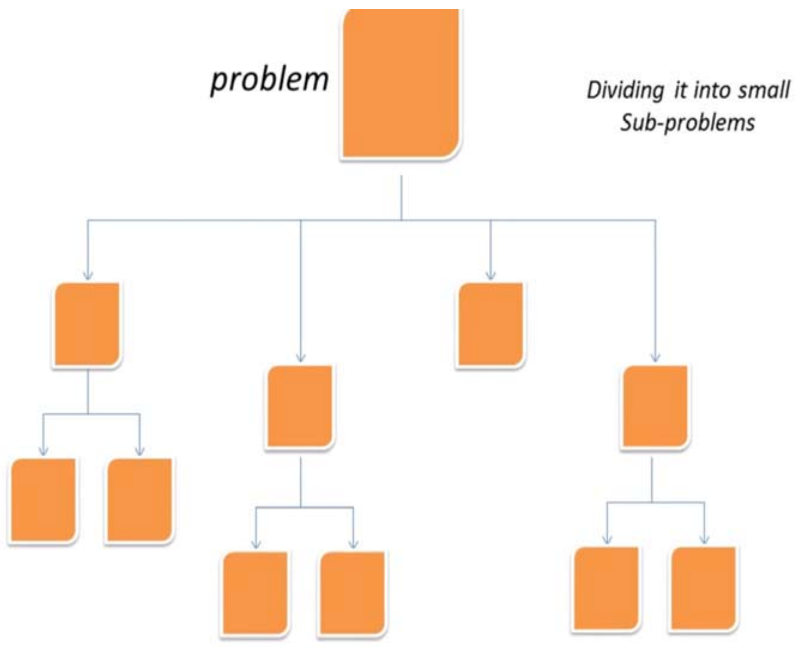
2- Object-Oriented approach (Object-Oriented Programming)

8.1 Structured Programming:

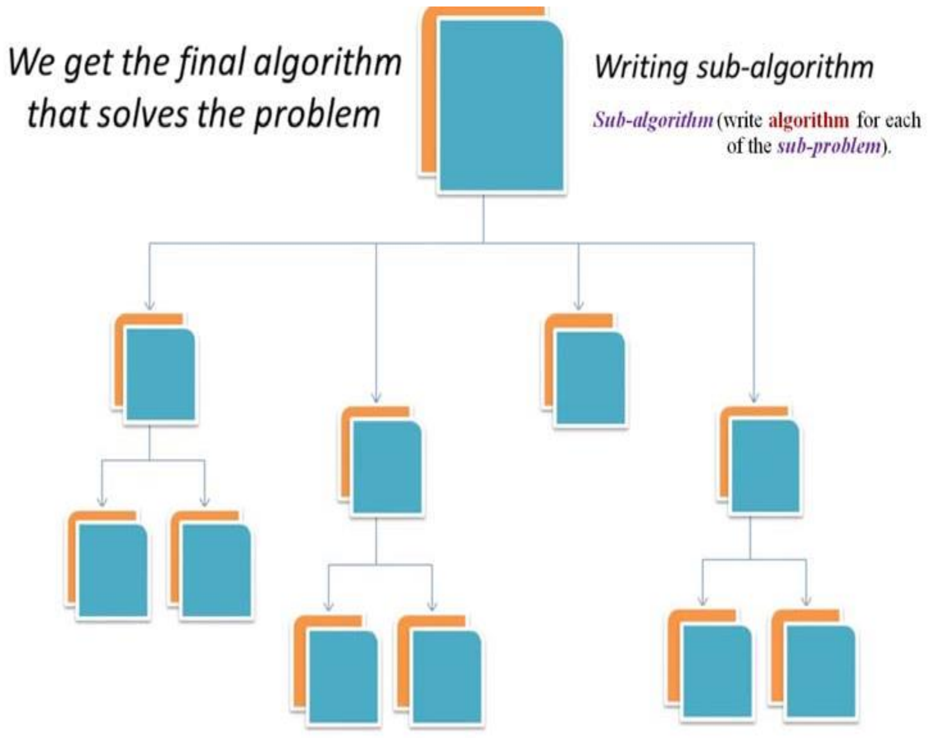
Dividing a problem into smaller sub-problems is called structured-design. Each sub-problem is then analyzed, and a solution is obtained to solve the sub- problem. The solutions to all of the sub-problems are then combined to solve the overall problem. The process of implementing a structured design is called **Structured Programming**, see Figure 9. The structured-design approach is also known as Top-down design or Bottom-up design.

Tip: Programs written using the **Structured-design** approach are;

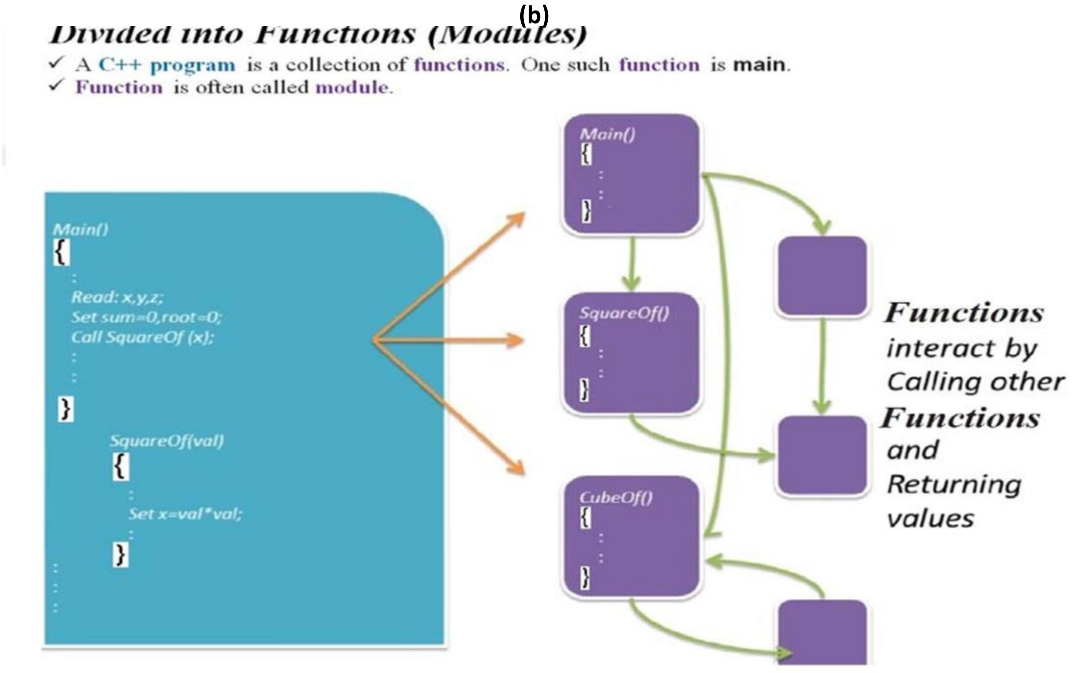
- ✓ easier to understand,
- ✓ easier to test and debug, and
- ✓ easier to modify.



(a)


We get the final algorithm that solves the problem
Writing sub-algorithm
Sub-algorithm (write algorithm for each of the sub-problem).

(b)


Divided into Functions (Modules)

- ✓ A C++ program is a collection of functions. One such function is main.
- ✓ Function is often called module.

```
Main()
{
   :
   Read: x,y,z;
   Set sum=0,root=0;
   Call SquareOf (x);
   :
   :
}
        SquareOf(val)
        {
           :
           Set x=val*val;
           :
        }
:
:
:
```

Main()
SquareOf()
CubeOf()
Functions interact by Calling other Functions and Returning values

(c)

Figure 9: Structured Programming

8.2 Control Structures in structured design (within C++): An Introduction

The programs in C++ executes its statements one by one from beginning to end. There are different approaches that can be used to solve same problems. The format of program should be such that it is easy to trace the flow of execution of statements. By making the flow of execution traceable, it is easy to develop accurate, error-free and maintainable code. As the statements are executed, their execution may vary depending upon the circumstances. The programs not only store data but also perform manipulative tasks and to perform such tasks C++ has provided tools which help in performing repetitive tasks and other manipulative actions(decision making). The tools comes in the form of **Control Structures.** So **Control Structures in C++** tells about the order in which the statements are executed and helps to perform manipulative, repetitive and decision making actions.

The **structured-design** approach uses **Three** fundamental types of program flow:

- **Sequence**: Instructions are executed in the order in which they appear.
- **Selection**: The program branches to different instructions depending on whether a condition is met or conditions. Within C++; if, if..else, ... or switch.
- **Repetition**: The program repeats particular statements a certain number of times based on some condition(s). Within C++; for, while or do…while.

1. Selection Structure (Branching Statements)

As the name suggests they alter the sequential execution of the statements of program depending upon the condition. Branching Statements are included in this control structure. Following branching statements are supported by C++ :

- *if* statement.
- *if-else* statement.

- *switch* statement.
- *goto* statement.

Except *goto* statement which is unconditional branching statement, all the other ones are conditional branching statements.

Lecture 8

if Statement

if Statement helps in controlling the flow of the program by providing programmer by an ability to make decisions like what part of code to execute thus controlling the sequence of execution of statements. Syntax of the *if*-statement is as follows:

if (test-expression) statement;

As mentioned in the syntax of the statement, the test expression is always mentioned in parentheses. In this statement if the test-expression is true then the statement which is followed immediately is executed. If the test-expression is false then the control will transfer to the next statement following the *if* construct. Multiple statements can also be included in the *if-statement* construct. These multiple statements must be enclosed in curly braces {}, these statements are known as Compound Statements.

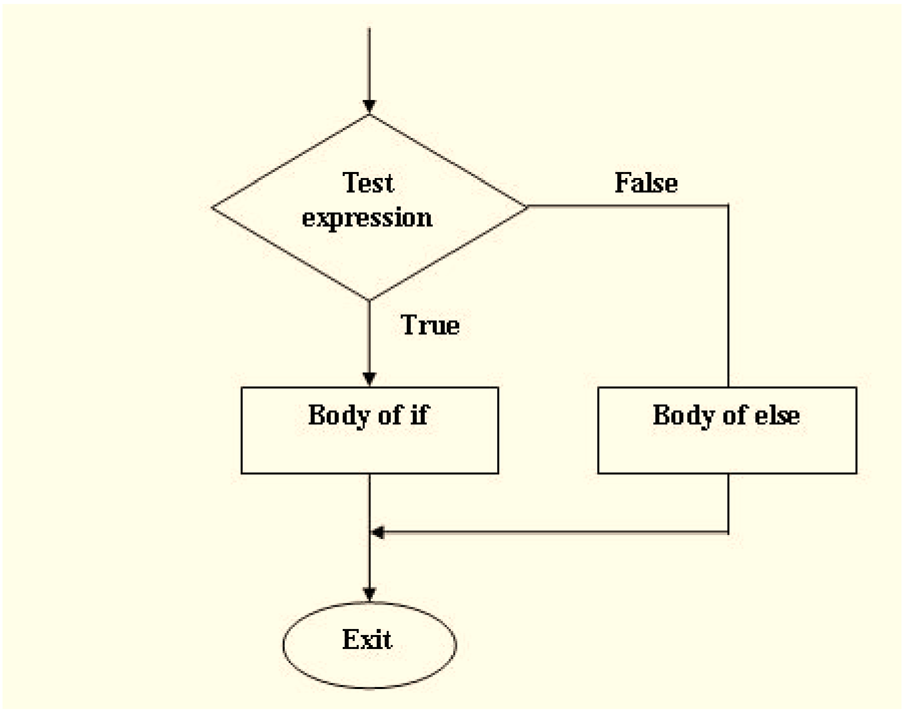
if-else Statement

This statement is quite similar to *if-statement*, this statement also specify what happens when the condition is not satisfied by providing *else* as an alternative statement. If you want to perform some action even if the test-expression fails i.e. condition is false then if-*else* statement is used. Syntax of *if-else statement* is as follows:

If (test-expression) { Statement1; } else { Statement2; }

As mentioned in syntax of the *if-else statement*, if the test expression is true then statement 1 is executed else statement 2 is executed if the test expression fails. The ***else*** clause of an *if…else* statement is associated with the closest previous ***if*** statement.

In an *if-else statement* only the code associated with *if* or the code associated with *else* is executed, never both. The following figure explains the flow of *if-else statement*:



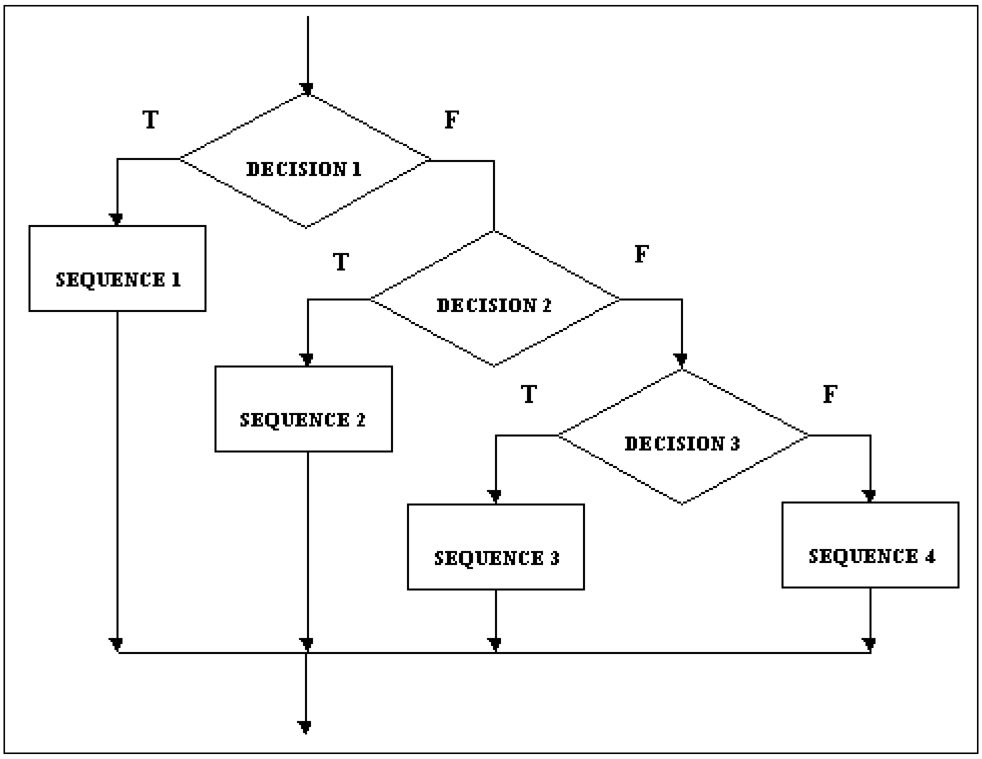
In the above example if the condition fails then else part of the program executes and if the condition is true then first part i.e. if part is executed. Similar to *if-statement*, *if-else statement* can also be used with compound statements.

Nested if else Statement

As the name suggests this statement contains multiple *if-else statements* which are used in multi-way decisions that arise when there are multiple conditions and different actions are required depending upon the conditions. The syntax of *Nested if-else statement* is as follows:

If (test-expression1) statement1; else if (test-expression2) statement2; else if (test-expression3) statement3;

The following figure explain the flow of *Nested if-else statement*:



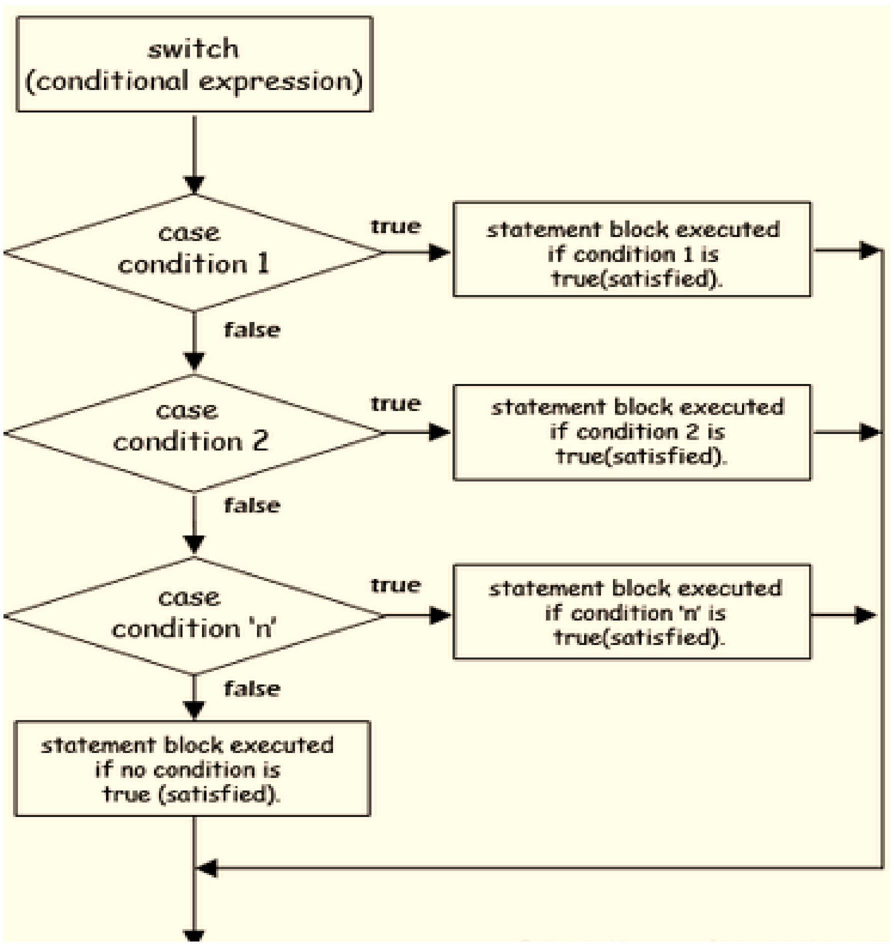
As the above flowchart explains, if the test-expression1 i.e. decision1 is true, the whole chain is terminated. The chain will only continue if the test-expression1 or decision1 fails, similar is the case for decision2 and so on. In nested if-else statements, the else statement is associated with corresponding inner most if statement.

Lecture 9

switch Statement

This statement is alternative to *nested if-else statement.* It is multi-way decision making construct which allows you to choose different statements depending upon the conditions. This switch statement successfully tests the expression against the integer or character constant and when there is a match, the statement associated with the constant is executed. *break statement* is very important in a switch statement structure. It causes exit from the switch structure after the case statements are executed. If the *break statement* is not mentioned then the control will transfer to the next case statement. When the condition in case is true the corresponding statement will get executed. The *default* part is optional in the switch statement, if none of the case is true then default statement is executed. The syntax and flow of *switch-statement* are explained in the following figures:

```
switch(test-expression) {
    case constant 1   : statement sequence 1; break;
    case constant 2   : statement sequence 2; break;
    case constant 3   : statement sequence 3; break;
    : :
    case constant n-1: statement sequence n-1; break;
   [default          :       statement sequence n]; }
```


switch (conditional expression)
case condition 1
true
statement block executed if condition 1 is true(satisfied).
false
case condition 2
true
statement block executed if condition 2 is true(satisfied).
false
case condition 'n'
true
statement block executed if condition 'n' is true(satisfied).
false
statement block executed if no condition is true (satisfied).

switch vs if-else

- ***switch*** can only test the expression for equality whereas ***if-else*** can evaluate logical and relational expressions.
- ***if-else*** statement is more flexible and versatile than ***switch*** statement as ***if-else*** can handle range of values whereas ***switch*** case can have single value.
- cases in ***switch*** statement cannot handle floating-point values whereas ***if-else*** statements can handle floating point values apart from integer and character.

A ***switch*** statement is more efficient than nested ***if-else*** statement A switch statement can only work for equality comparisons. A switch statement can also be nested.

2. Looping Structure(Iterative Statements)

With the help of Loops, a section of code can be executed repeatedly until a termination condition is met. Loops or the looping statements are also called iteration statements. There are three kinds of loops in C++:

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

<u>Elements of Looping Structure</u>

There are four elements in the looping structure that control its execution. These elements are given below:

- **Initialization Expression:** The loop control variable must be initialized for the looping structure. This expression is executed only once in the beginning of the loop.
- **Test Expression:** As the name suggests this element test the condition which decides whether the loop body will be executed or not. If the condition is false the loop gets terminated else the loop statements get executed. ***for-loop*** and ***while-loop*** are entry controlled loops where the test expression is evaluated before entering whereas ***do-while*** is an exit controlled loop.
- **Update Expression:** This element updates the value of loop variable. This element is executed after executing the body of loop.

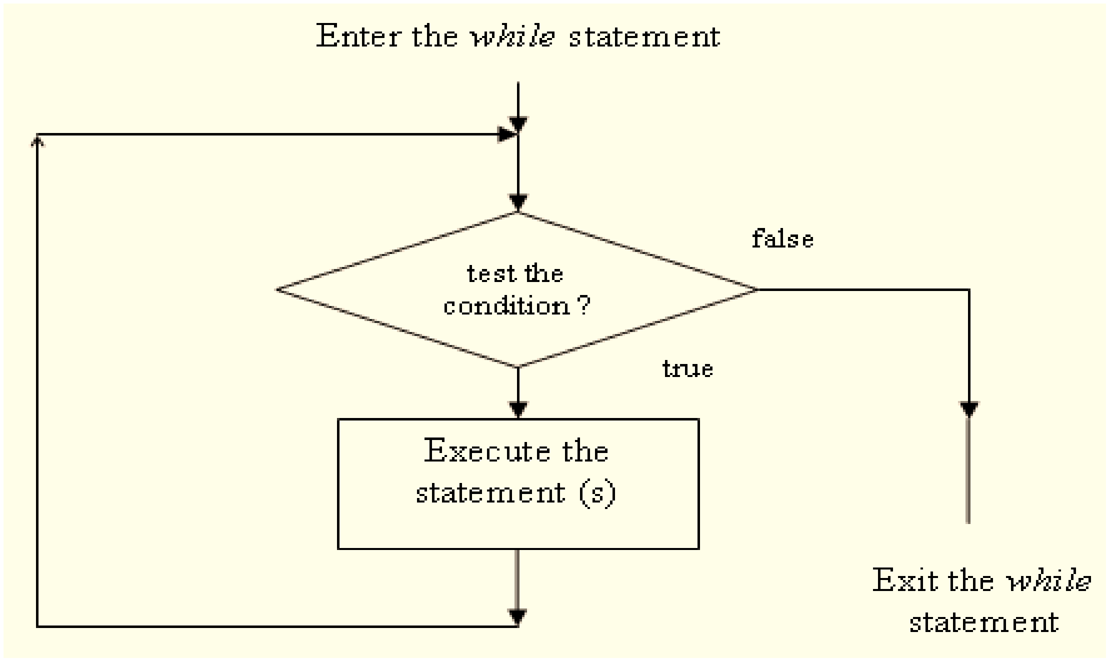
- **Loop Body:** The body of the loop contains the statements which are executed repeatedly as long as the test expression is true. In entry controlled loop the body of the loop is executed after the test expression whereas in the exit controlled loop the body of the loop gets executed before the test-expression.

while-loop

what if the number of iterations to be performed are not known in advance, *while-loop* can be used in such a case. The statements inside the while loop keeps on executing until the test condition is true. The syntax of *while-loop* is as follows :

while (test-condition) { statement(s); }

The flowchart of *while-loop* is as follows :


Enter the *while* statement
test the condition ?
false
true
Execute the statement (s)
Exit the *while* statement

while-loop evaluates the test-expression before entering into the loop. *while-loop* can also be infinite loop, only if the loop variable is not updated inside the body. *while-loop* can also be empty loop, only if it contains a null statement in its body.

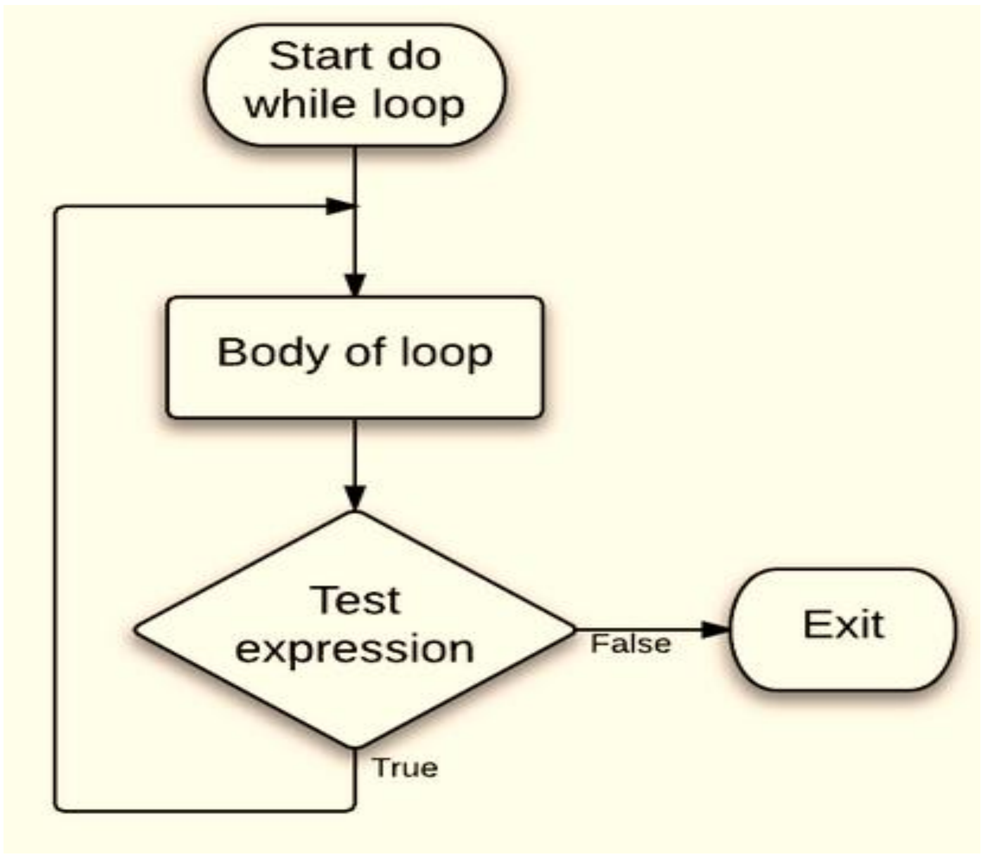
Lecture 10

do-while loop

As stated earlier, *do-while* loop is an exit-controlled loop i.e. the test-expression is evaluated after the loop-body gets executed. Whereas the *for-loop* and *while-loop* is evaluated at the top of the loop, *do-while* is evaluated at the bottom of the loop. This looping structure is used where it is desirable to execute the body of a *while-loop* at least once, even if the test-expression is false. The syntax of *do-while* loop is as follows:

```
do  { statement; } while (test-expression);
```

The flowchart of *do-while* loop is as follows:



First the statement inside the body of the loop is executed and then the test expression is evaluated. Example of *do-while* loop is as follows:

```
int i=0;
do
{
cout<<i<<endl;
i++;
}
while (i<5);
```

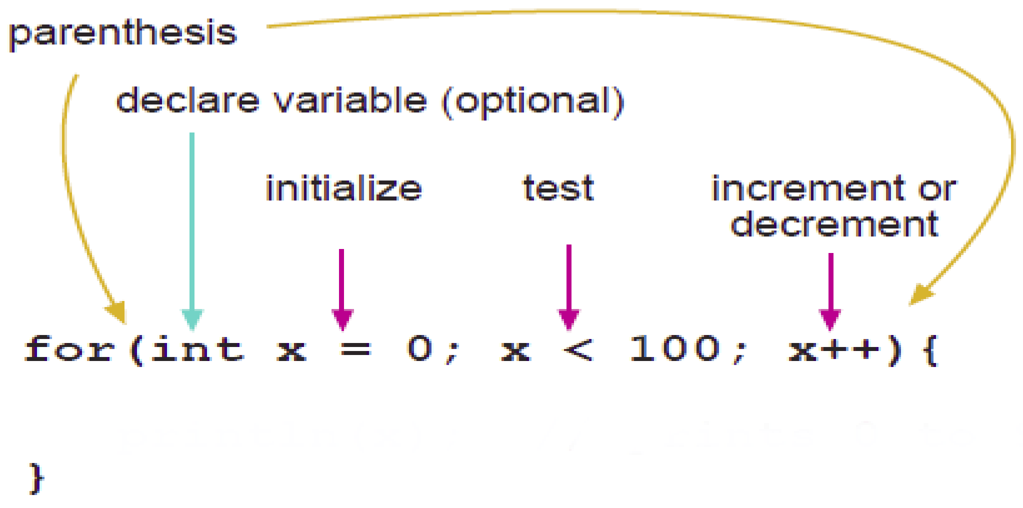
Output of the program: 0 1 2 3 4 The *do-while* loop is used commonly in a menu selection routine, where the menu is displayed at least once and then depending upon the user’s response, it is either repeated or terminated.

for-loop

The *for-loop* is used where a statement is executed a fixed number of times. Compound statements can also be used in the body of for loop. The syntax of *for-loop* is as follows :

for (initialization expression; test-expression; update expression) {

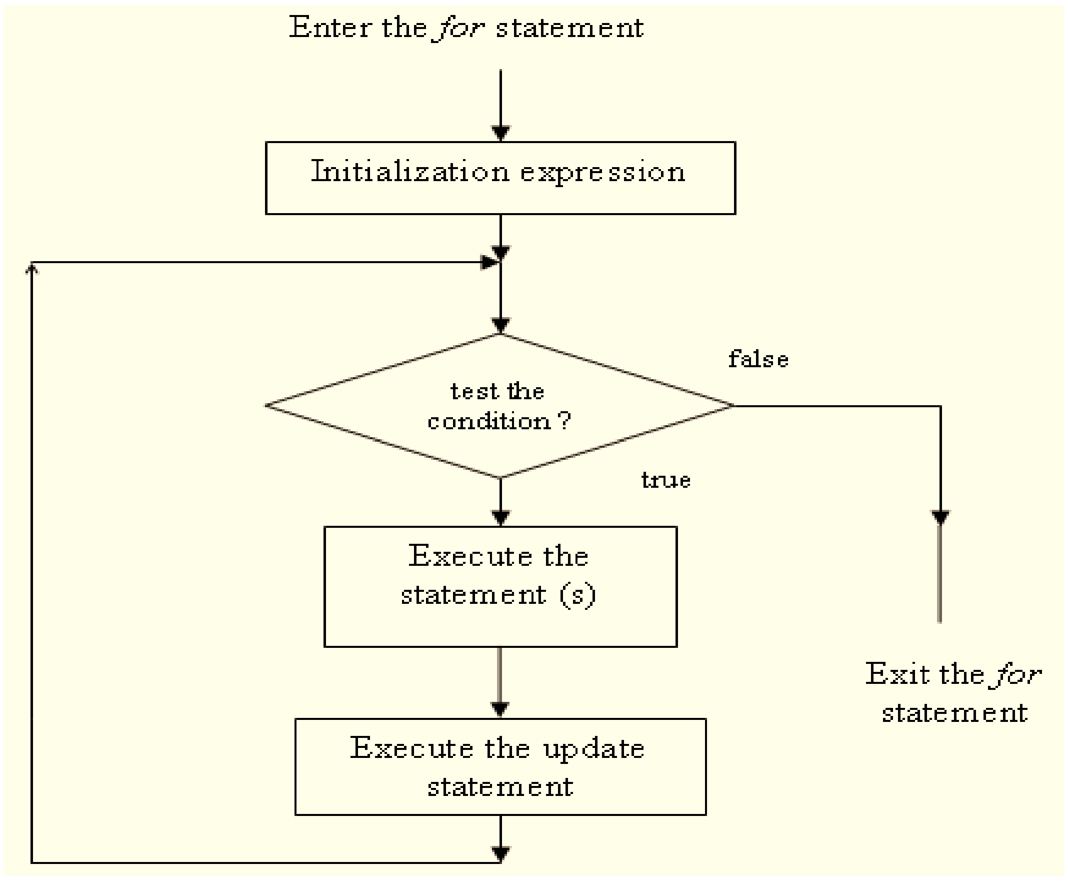
body of for-loop; }



```
for(int x = 0; x < 100; x++){

}
```

The flowchart of the for-loop is as follows:


Enter the *for* statement
Initialization expression
test the condition ?
false
true
Execute the statement (s)
Execute the update statement
Exit the *for* statement

- The **initialization expression** step is executed first, and only once.
- Then, the **test-condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the **update** statement is executed to update the loop variable.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself.