

MATLAB Course

For Engineers

lecture 5



Algorithms and Control Structures

Algorithm: an ordered sequence of precisely defined instructions that performs some task in a finite amount of time. ***Ordered*** means that the instructions can be **numbered**, but an algorithm **must** have the **ability** to alter the order of its instructions using a ***control structure***. There are **three** categories of algorithmic operations:

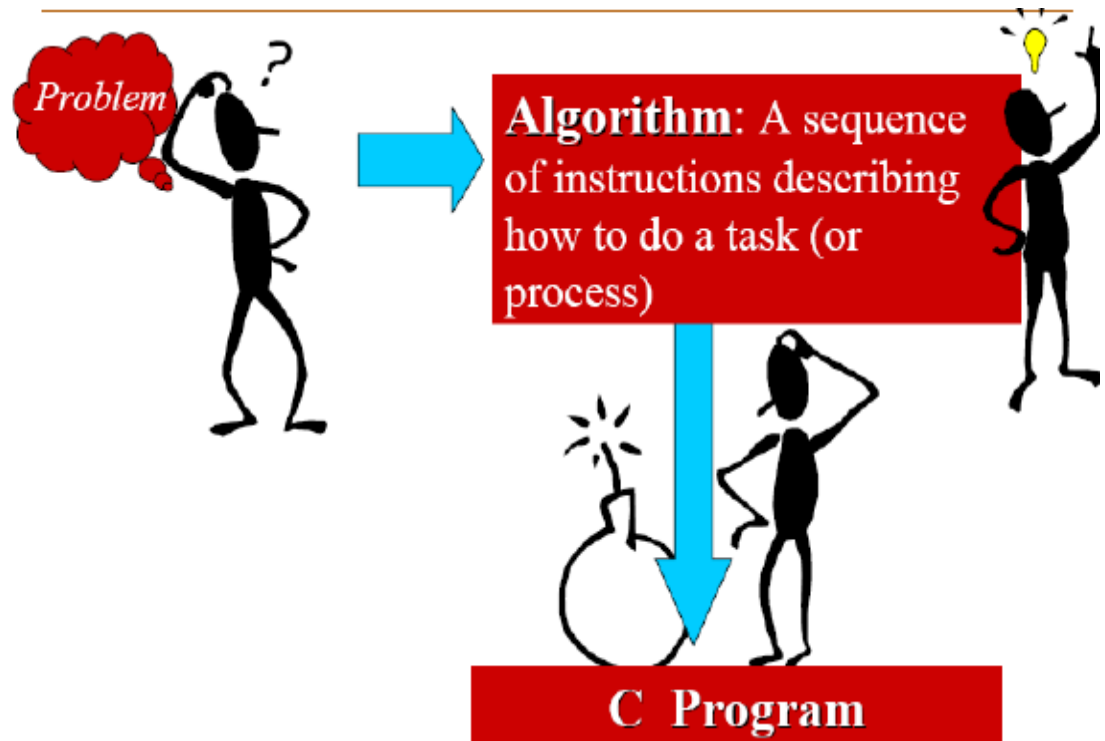
Sequential operations: Instructions executed in **order**.

Conditional operations: **Control** structures that **first** ask a **question** to be answered with a **true/false** answer and then select the next instruction based on the answer.

Iterative operations (loops): **Control** structures that **repeat** the execution of a block of instructions.

Components of an Algorithm

- Variables and values
- Instructions
- Sequences
- Procedures
- Selections
- Repetitions
- Documentation



Structured Programming From Algorithms to Programs

A technique for designing programs in which a hierarchy of *modules* is used, each having a single **entry** and a **single** exit point, and in which control is passed **downward** through the structure **without** unconditional branches to higher levels of the structure.

In MATLAB these modules can be **built-in or user-defined functions**.

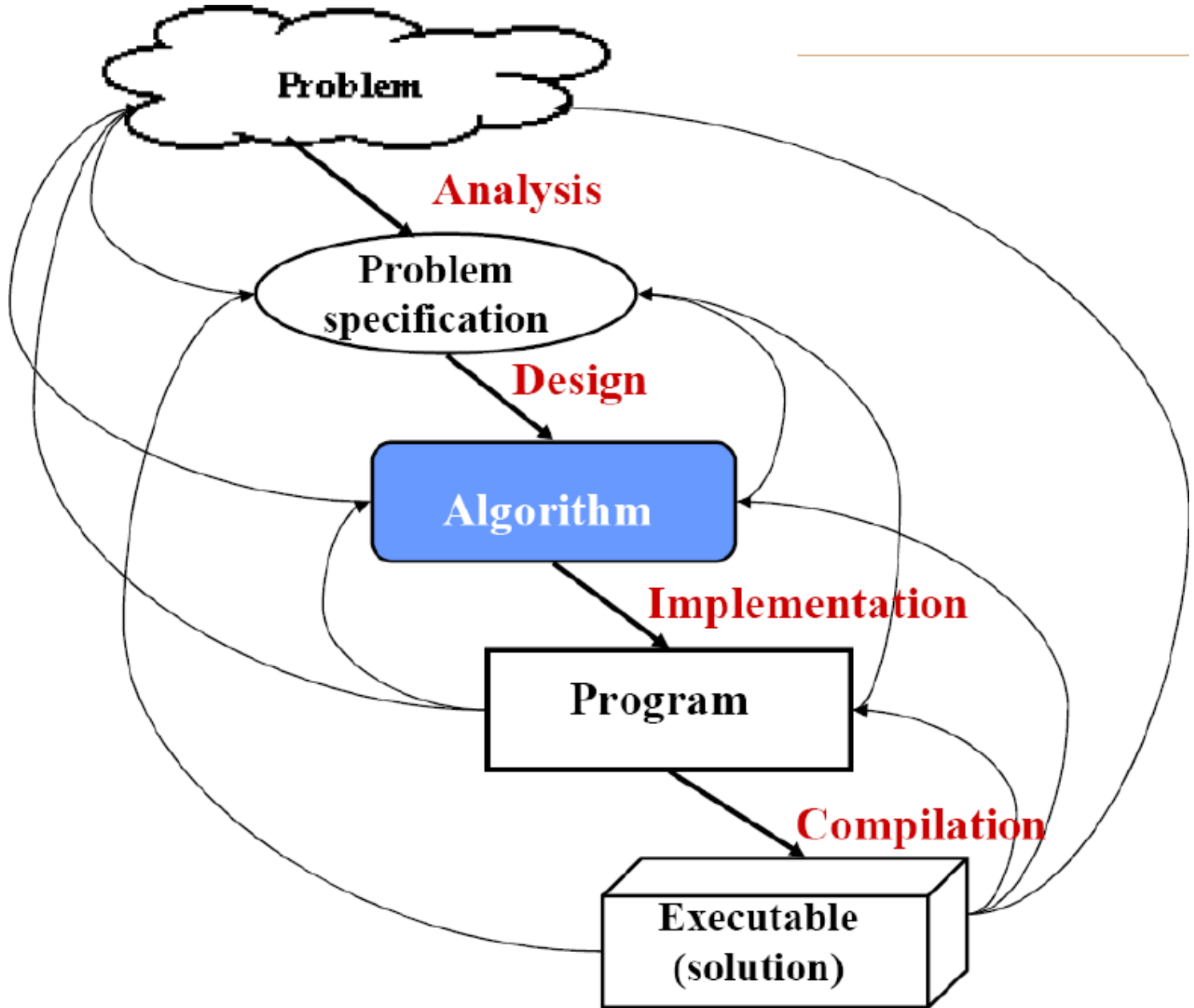
Advantages of structured programming

1. Structured programs are **easier** to write because the programmer **can study** the overall problem **first** and then deal with the details later.
2. Modules (functions) **written** for one application can be **used** for **other** applications (this is called *reusable code*).
3. Structured programs are **easier** to debug because each module is designed to perform just one task and thus it can be **tested separately** from the other modules.
4. Structured programming is **effective** in a **teamwork** environment because **several people** can work on a common program, each person developing one or more modules.
5. Structured programs are **easier** to **understand** and **modify**, especially if **meaningful** names are chosen for the modules and if the **documentation** clearly identifies the module's task.

Steps for developing a computer solution:

1. **State** the problem concisely.
2. **Specify** the data to be **used** by the program. This is the “**input**.”
3. **Specify** the information to be **generated** by the program. This is the “**output**.”
4. Work through the solution **steps** by **hand** or with a calculator; use a simpler set of data if necessary.
5. **Write** and **run** the program.
6. **Check** the **output** of the program with **your hand** solution.
7. **Run** the program with your **input** data and perform a reality check on the output.
8. If you will use the program as a **general** tool in the future, **test** it by running it for a range of reasonable data values; perform a reality check on the results.

The Problem solving Process



Effective documentation can be accomplished with the use of

1. Proper selection of variable **names** to reflect the quantities they represent.
2. Use of **comments** within the program.
3. Use of **structure charts**.
4. Use of **flowcharts**.
5. A **verbal** description of the program, often in *pseudocode*. (in which natural language and mathematical expressions are used to construct statements that look like computer statements but without detailed syntax).

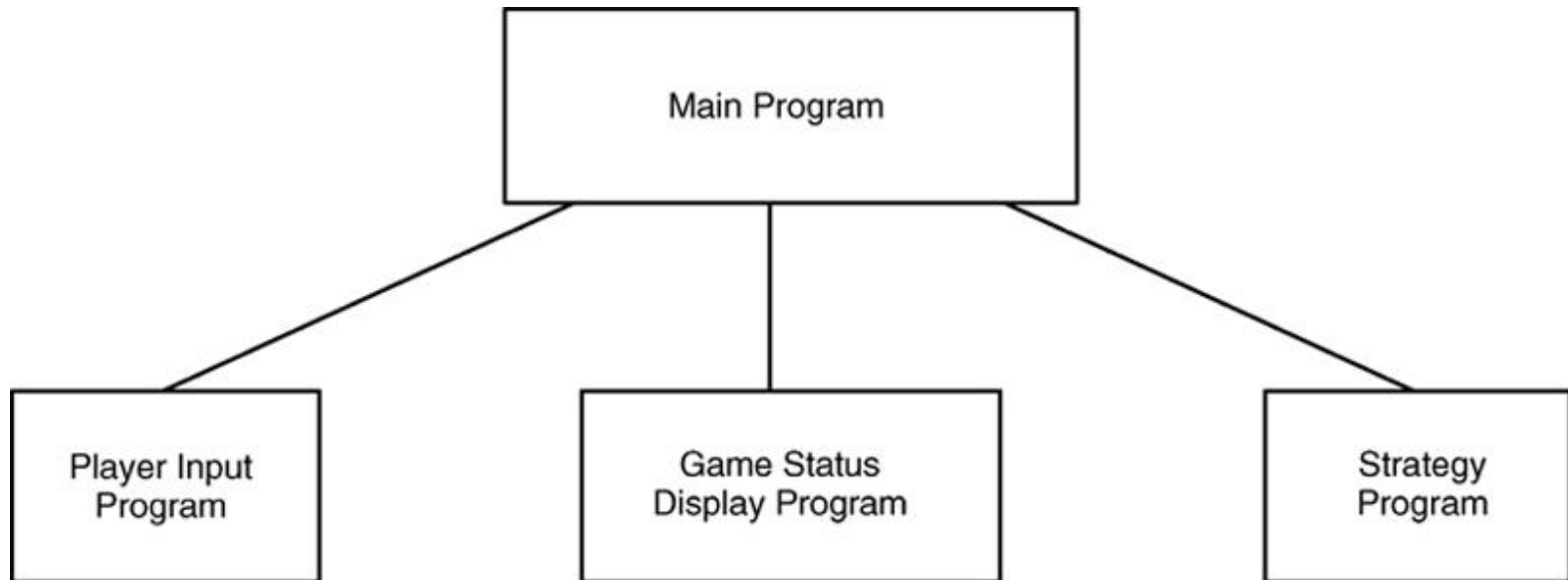
Documenting with Charts

Two types of charts aid in developing structured programs and in documenting them.

These are structure charts and flowcharts.

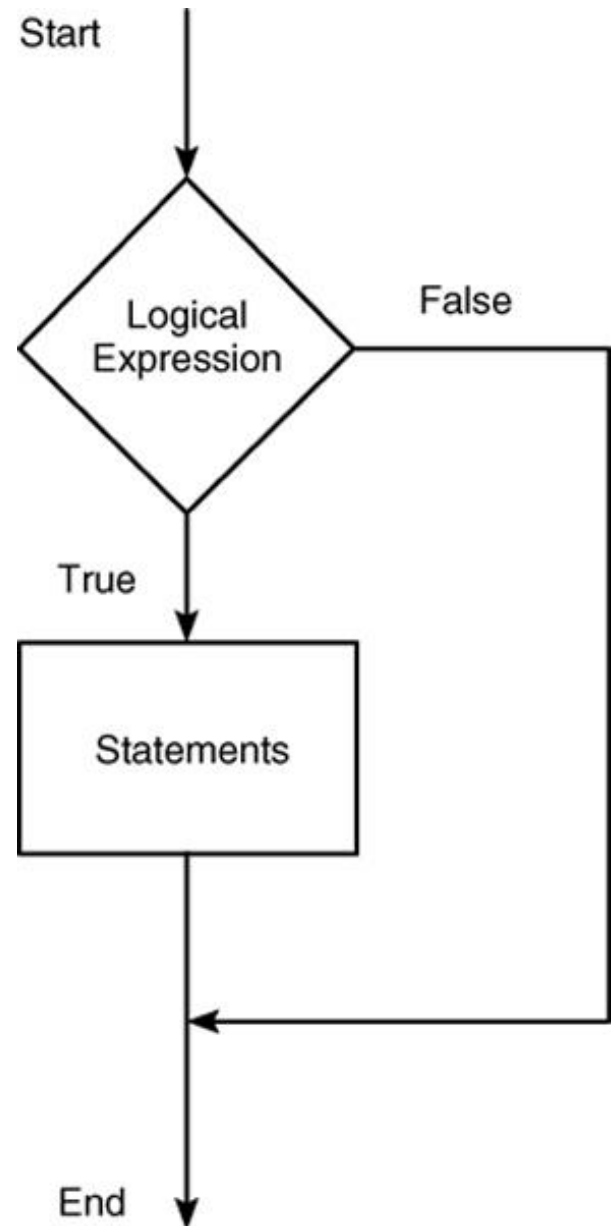
A structure chart is a **graphical** description showing how the different parts of the program are **connected** together.

Structure chart of a game program.



Flowcharts are useful for developing and documenting programs that contain **conditional** statements, because they **can** display the various **paths** (called “**branches**”) that a program can take, depending on how the conditional statements are executed.

Flowchart representation of the **if** statement.



Finding Bugs

Debugging a program is the process of **finding** and **removing** the “**bugs**,” or **errors**, in a program. Such errors usually fall into one of the following categories.

1. **Syntax errors** such as omitting a **parenthesis** or **comma**, or **spelling** a command name incorrectly. MATLAB usually **detects** the more obvious errors and displays a message describing the error and its location.
2. Errors due to an **incorrect** mathematical procedure, called ***runtime errors***. Their occurrence often depends on the particular input data. A common example is **division by zero**.

To locate a runtime error, try the following:

- 1. Always** test your program with a **simple** version of the problem, whose **answers** can be **checked** by hand calculations.
- 2. Display** any **intermediate** calculations by removing semicolons at the end of statements.
- 3. To test user-defined functions**, try commenting out the **function** line and running the file as a script.
- 4. Use the debugging features of the Editor/Debugger.**

Relational operators

Operator

Meaning

$<$

Less than.

$<=$

Less than or equal to.

$>$

Greater than.

$>=$

Greater than or equal to.

$==$

Equal to.

\neq

Not equal to.

For example, suppose that $\mathbf{x}=[6,3,9]$ and $\mathbf{y}=[14,2,9]$.
The following MATLAB session shows some examples.

```
>>z = (x < y)
```

```
z =  
    1    0    0
```

```
>>z = (x ~= y)
```

```
z =  
    1    1    0
```

```
>>z = (x > 8)
```

```
z =  
    0    0    1
```

The **relational** operators can be used for array **addressing**.

For example, with $x = [6, 3, 9]$ and $y = [14, 2, 9]$, typing

$z = x(x < y)$

Finds all the elements in x that are **less** than the corresponding elements in y . The result is $z = 6$.

The **arithmetic** operators $+$, $-$, $*$, $/$, and \backslash have **precedence** over the **relational** operators. Thus the statement

$z = 5 > 2 + 7$

is equivalent to

$z = 5 > (2 + 7)$

and returns the result $z = 0$.

We can use parentheses to change the order of precedence;

for example, $z = (5 > 2) + 7$ evaluates to $z = 8$.

The logical Class

When the **relational** operators are used, such as `x = (5 > 2)` they create a **logical** variable, in this case, **x logical** variables may have only the values **1 (true)** and **0 (false)**.

Just because an **array** contains only **0s** and **1s**, however, it is not necessarily a logical array. **For example**, in the following session **k** and **w** appear the same, but **k** is a **logical** array and **w** is a **numeric** array, and thus an error message is issued.

```
>>x = [-2:2]; k = (abs(x)>1)
```

```
k =
```

```
    1    0    0    0    1
```

```
>>z = x(k)
```

```
z =
```

```
   -2    2
```

```
>>w = [1,0,0,0,1]; v = x(w)
```

```
??? Subscript indices must either be real  
positive... integers or logicals.
```

Accessing Arrays Using Logical Arrays

When a **logical** array is used to address another array, it extracts from that array the elements in the locations where the **logical** array has **1s**.

So typing **A(B)**, where **B** is a **logical** array of the **same** size as **A**, returns the values of **A** at the **indices** where **B** is **1**.

Specifying array subscripts with **logical** arrays extracts the elements that correspond to the **true (1)** elements in the **logical** array.

Given **A = [5, 6, 7; 8, 9, 10; 11, 12, 13]** and **B = logical(eye(3))**, we can extract the **diagonal** elements of **A** by typing **C = A(B)** to obtain **C = [5; 9; 13]**.

Logical operators

Operator	Name	Definition
~	NOT	$\sim A$ returns an array the same dimension as A ; the new array has ones where A is zero and zeros where A is nonzero .
&	AND	$A \& B$ returns an array the same dimension as A and B ; the new array has ones where both A and B have nonzero elements and zeros where either A or B is zero .
	OR	$A B$ returns an array the same dimension as A and B ; the new array has ones where at <u>least one</u> element in A or B is nonzero and zeros where A and B are both zero .

Operator	Name	Definition
&&	Short-Circuit AND	Operator for scalar logical expressions. A && B returns true if both A and B evaluate to true , and false if they do not.
 	Short-Circuit OR	Operator for scalar logical expressions. A B returns true if either A or B or both evaluate to true , and false if they do not.

Order of precedence for operator types.

Precedence	Operator type
First	Parentheses ; evaluated starting with the innermost pair.
Second	Exponentiation operators (^) and logical NOT (~)
Third	Mathematical operations (*,/,+,-)
Fourth	Relational operators; evaluated from left to right .
Fifth	Logical AND .
Sixth	Logical OR .

Logical functions

Logical function

Definition

all(x)

Returns a **scalar**, which is **1** if **all** the elements in the vector **x** are **nonzero** and **0 otherwise**.

all(A)

Returns a **row** vector having the **same** number of columns as the matrix **A** and containing **ones** and **zeros**, depending on whether or not the corresponding column of **A** has all **nonzero** elements.

any(x)

Returns a **scalar**, which is **1** if any of the elements in the vector **x** is **nonzero** and **0 otherwise**.

any(A)

Returns a **row** vector having the **same** number of columns as **A** and containing **ones** and **zeros**, depending on whether or not the corresponding column of the matrix **A** contains any **nonzero** elements.

finite(A)

Returns an **array** of the same dimension as **A** with **ones** where the elements of **A** are finite and **zeros** elsewhere.

Logical function

Definition

ischar(A)

Returns a **1** if **A** is a character array and **0** otherwise.

isempty(A)

Returns a **1** if **A** is an empty matrix and **0** otherwise.

isinf(A)

Returns an array of the same dimension as **A**, with ones where **A** has '**inf**' and **zeros** elsewhere.

isnan(A)

Returns an array of the same dimension as **A** with **ones** where **A** has '**NaN**' and **zeros** elsewhere. ('**NaN**' stands for "**not a number**," which means an undefined result.)

isnumeric(A)

Returns a **1** if A is a **numeric** array and **0** otherwise.

isreal(A)

Returns a **1** if A has **no elements** with **imaginary** parts and **0** otherwise.

logical(A)

Converts the elements of the array **A** into **logical** values.

xor(A,B)

Returns an array the same dimension as **A** and **B**; the new array has **ones** where either **A** or **B** is **nonzero**, but **not both**, and **zeros** where **A** and **B** are either both **nonzero** or **both zero**.

Logical Operators & the `find` Function

Consider the session

```
>>x = [5, -3, 0, 0, 8]; y = [2, 4, 0, 5, 7];  
>>z = find(x&y)  
z =  
     1     2     5
```

Note that the `find` function returns the *indices*, and **not** the *values*.

Remember, the find function returns the *indices*, and not the *values*. In the following session, note the difference between the result obtained by `y(x&y)` and the result obtained by `find(x&y)` in the previous slide.

```
>>x = [5, -3, 0, 0, 8]; y = [2, 4, 0, 5, 7];  
>>values = y(x&y)  
values =  
     2     4     7  
>>how_many = length(values)  
how_many =  
     3
```

The **if** Statement

The **if** statement's basic form is

الشرطية البسيطة

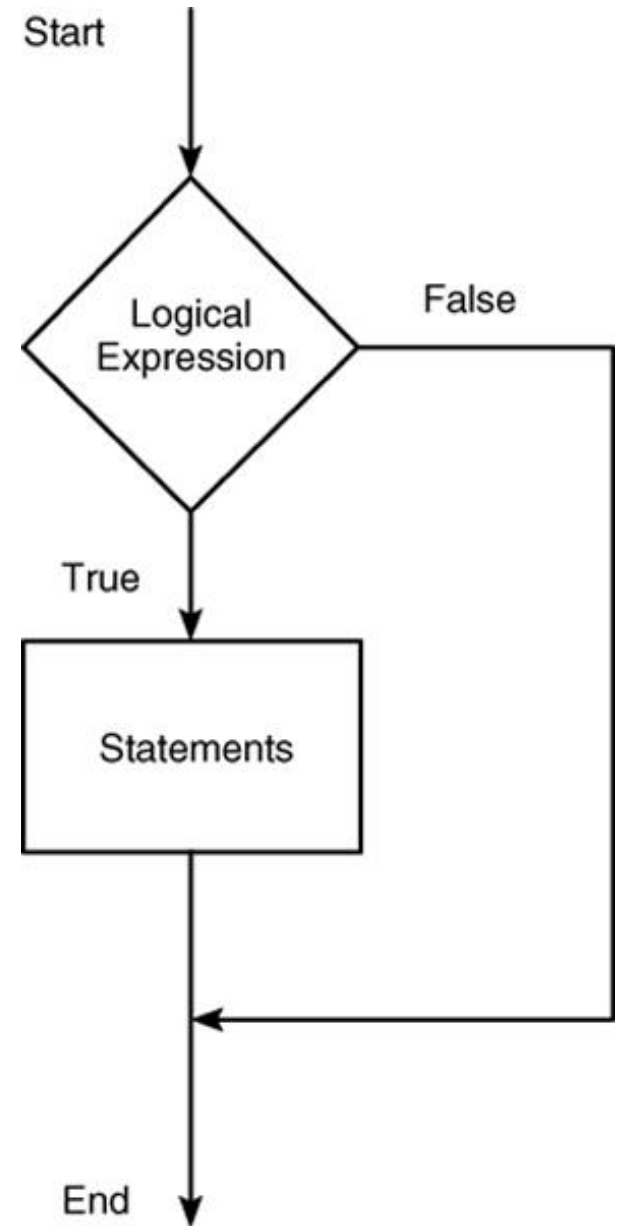
إذا تحقق الشرط نفذ

إذا لم يتحقق الشرط لا تنفذ و انهي

if logical expression
statements

end

Every **if** statement **must** have an accompanying **end** statement. The end statement marks the end of the *statements* that are to be executed if the *logical expression* is true.



The **else** Statement

الشرطية الاختيارية

هنا يجري الاختيار بين مسارين للتنفيذ

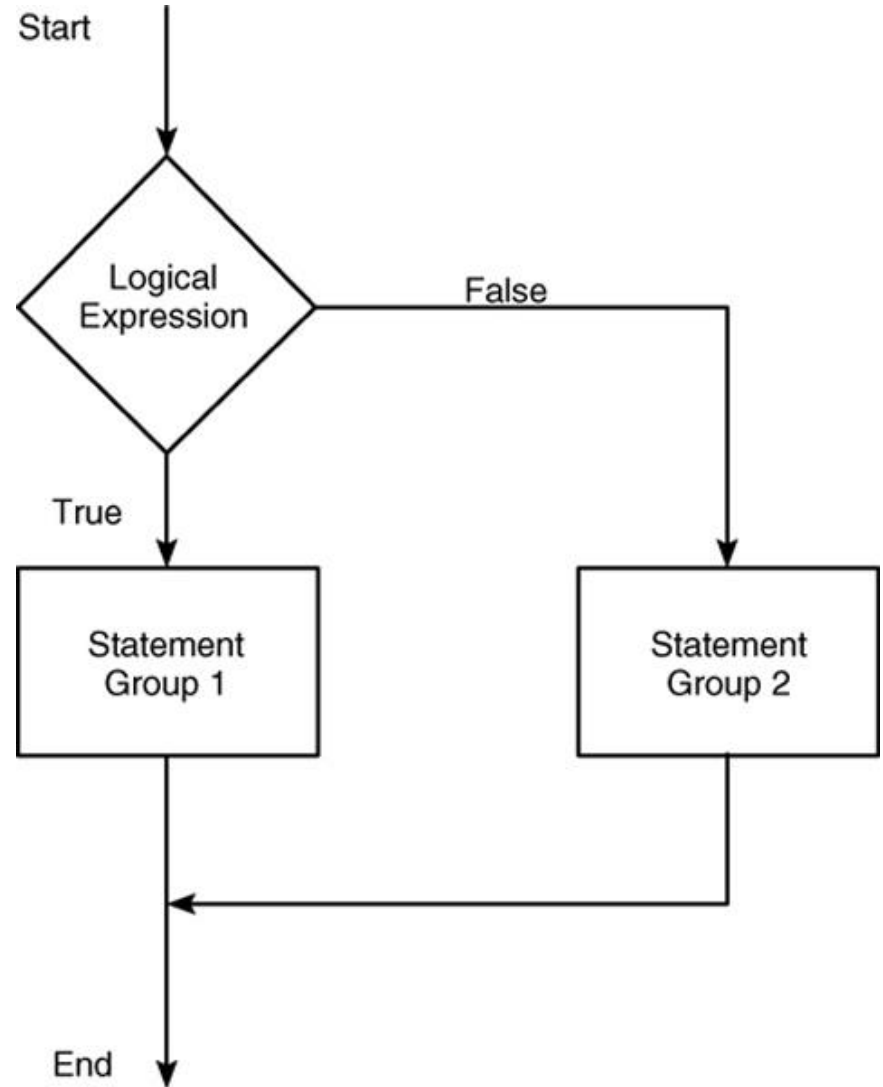
1- تحقق الشرط نفذ المسار 1

2- لم يتحقق الشرط نفذ المسار 2

The basic structure for the use of the else statement is

```
if logical expression
    statement group 1
else
    statement group 2
end
```

When the test, if **logical expression**, is performed, where the logical expression may be an **array**, the test returns a value of true only if **all** the elements of the logical expression are true!



The following statements

```
if logical expression 1
    if logical expression 2
        statements
    end
end
```

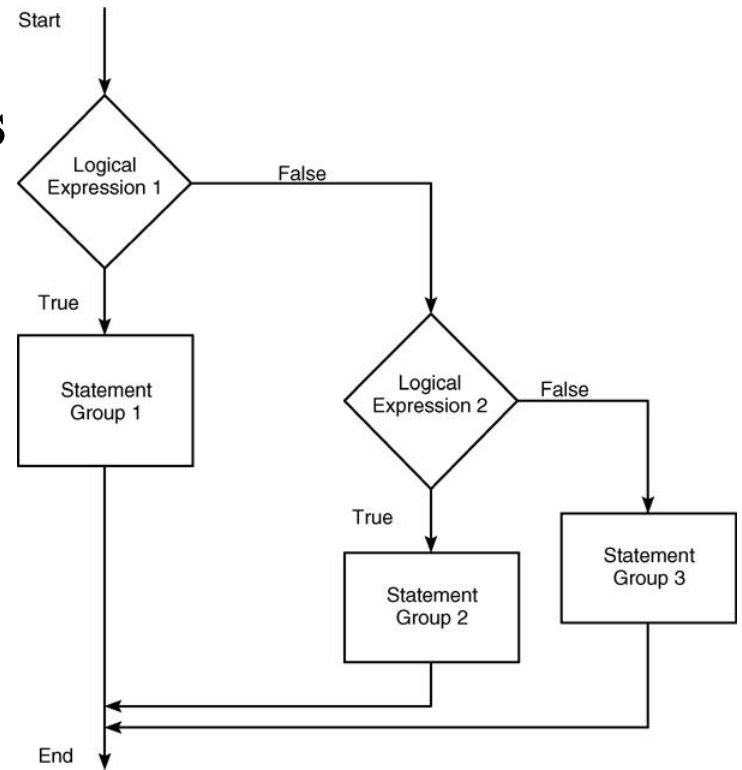
Can be replaced with the more concise program

```
if logical expression 1 & logical expression 2
    statements
end
```

The `elseif` Statement

The general form of the if statement is

```
if logical expression 1
    statement group 1
elseif logical expression 2
    statement group 2
else
    statement group 3
end
```



The `else` and `elseif` statements may be **omitted** if not required. However, if both are used, the `else` statement must come after the `elseif` statement to take care of all conditions that might be unaccounted for.

For example, suppose that $y = \log(x)$ for $x > 10$, $y = \text{sqrt}(x)$ for $0 \leq x \leq 10$, and $y = \exp(x) - 1$ for $x < 0$. The following statements will compute y if x already has a **scalar value**.

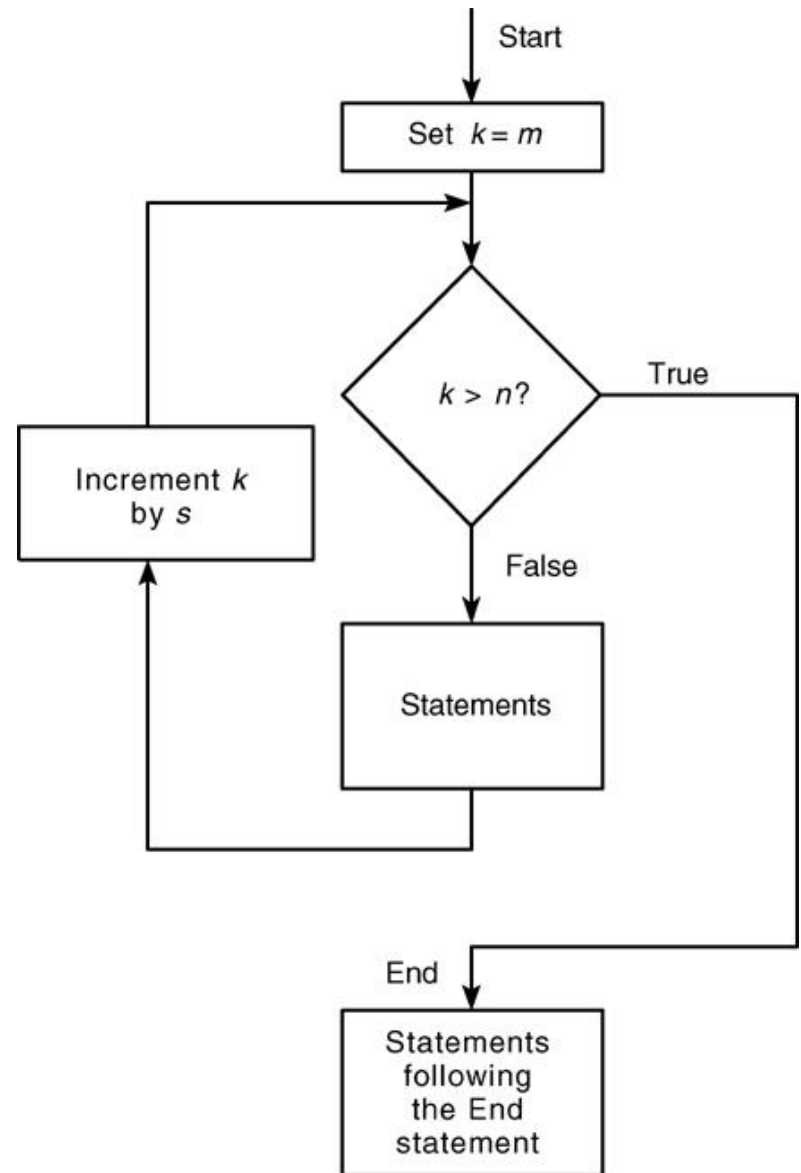
```
if x > 10
    y = log(x)
elseif x >= 0
    y = sqrt(x)
else
    y = exp(x) - 1
end
```

for Loops

A simple example of a for loop is

```
for k = 5:10:35
    x = k^2
end
```

The *loop variable* **k** is initially assigned the value **5**, and **x** is calculated from $x = k^2$. Each successive pass through the loop increments **k** by **10** and calculates **x** until **k** exceeds **35**. Thus **k** takes on the values **5, 15, 25, and 35**, and **x** takes on the values **25, 225, 625, and 1225**. The program then continues to execute any statements following the **end** statement.



Note the following rules when using **for** loops with the loop variable expression **k = m:s:n**

- The **step** value **s** may be **negative**.
Example: **k = 10:-2:4** produces **k = 10, 8, 6, 4**.
- If **s** is **omitted**, the **step** value defaults to **1**.
- If **s** is **positive**, the loop will **not** be executed if **m** is **greater** than **n**.
- If **s** is negative, the loop will **not** be executed if **m** is **less** than **n**.
- If **m** equals **n**, the loop will be executed **only once**.
- If the **step** value **s** is **not** an **integer**, round-off **errors** can cause the loop to execute a different number of passes than intended.

The `continue` Statement

The following code uses a `continue` statement to **avoid** computing the logarithm of a **negative number**.

```
x = [10,1000,-10,100];  
y = NaN*x;  
for k = 1:length(x)  
    if x(k) < 0  
        continue  
    end  
    y(k) = log10(x(k));  
end  
disp(y)
```

The result is `y = 1, 3, NaN, 2`.

while Loops

ما دام الشرط قائم كرر العملية

The **while** loop is used when the looping process **terminates** because a specified **condition** is satisfied, and thus the number of passes is **not** known in advance. A **simple example** of a while loop is

```
x = 5;
while x < 25
    disp(x)
    x = 2*x - 1;
end
```

The results displayed by the disp statement are 5, 9, and 17.

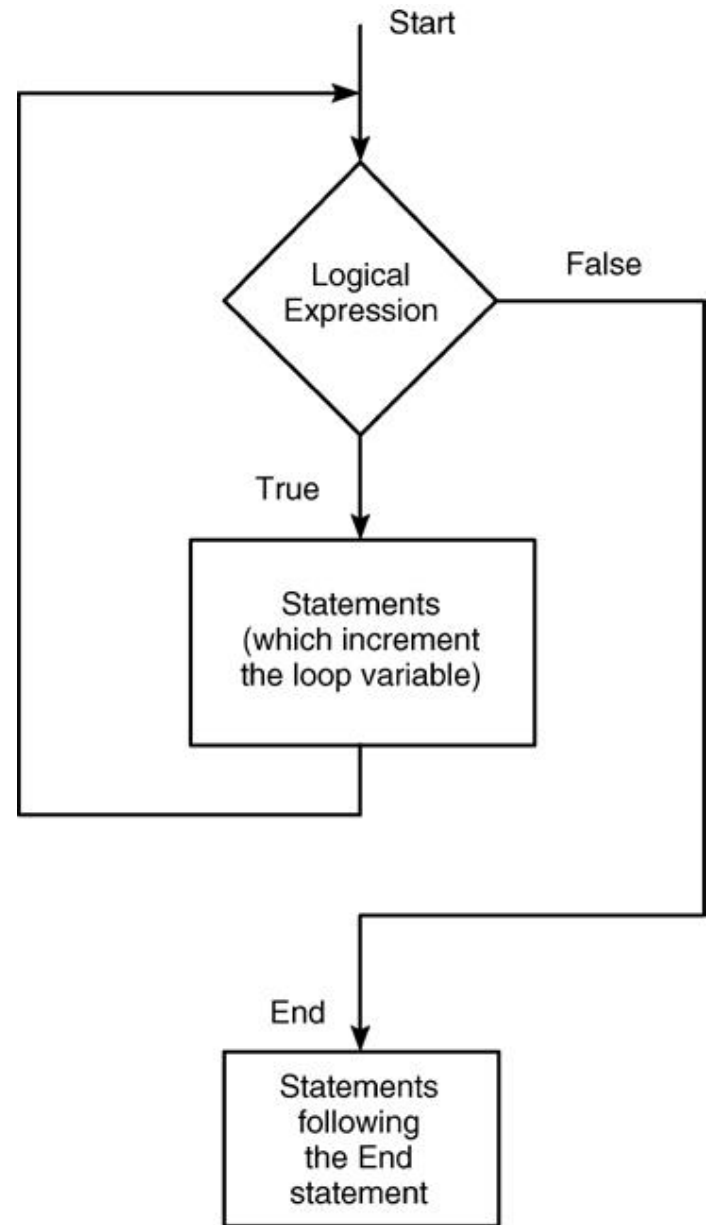
The **typical** structure of a **while** loop follows.

```
while logical expression  
    statements  
end
```

For the while loop to function properly, the following two

conditions must occur:

1. The loop variable **must** have a value **before** the while statement is executed.
2. The loop variable **must** be **changed** somehow by the *statements*.



A simple **example** of a **while** loop is

```
x = 5;k = 0;
```

```
while x < 25
```

```
    k = k + 1;
```

```
    y(k) = 3*x;
```

```
    x = 2*x-1;
```

```
end
```

The loop variable **x** is initially assigned the value **5**, and it keeps this value until the statement **x = 2*x - 1** is encountered the **first time**. Its value then changes to **9**. Before each pass through the loop, **x** is checked to see if its value is **less** than **25**. If so, the pass is made. If not, the loop is skipped.

Another **Example** of a **while** Loop

Write a **script file** to determine how many terms are required for the sum of the series $5k^2 - 2k$, $k = 1, 2, 3, \dots$ to **exceed 10,000**. What is the **sum** for this many terms?

```
total = 0;k = 0;
while total < 10000
    k = k + 1;
    total = 5*k^2 - 2*k + total;
end
disp('The number of terms is:')
disp(k)
disp('The sum is:')
disp(total)
```

The sum is 10,203 after 18 terms.

The **switch** Structure

The **switch** structure provides an **alternative** to using the **if**, **elseif**, and **else** commands. Anything programmed using **switch** can also be programmed using **if** structures.

However, for some applications the switch structure is more **readable** than code using the if structure.

The `switch` Structure

```
switch input expression
  case value1
    statement group 1
  case value2
    statement group 2
  .
  .
  .
  otherwise
    statement group n
end
```

Note: *expression* (which can be a scalar or string).

The following **switch** block displays the point on the compass that corresponds to that angle.

```
switch angle
  case 45
    disp('Northeast')
  case 135
    disp('Southeast')
  case 225
    disp('Southwest')
  case 315
    disp('Northwest')
  otherwise
    disp('Direction Unknown')
end
```

