CHAPTER 1

INTRODUCTION

Data Structure and Algorithm

CONTENTS

- Data structure definition,
  - ADT,
  - Classification of Data structures,
  - Array revision, pointer revision,
- Algorithm definition,
  - properties of algorithms,
  - expressing algorithms (natural language, flowchart, pseudocodes),
  - Algorithm complexity analysis (operation count, big-O, theta, omega), best case analysis, worst case analysis, average case analysis.

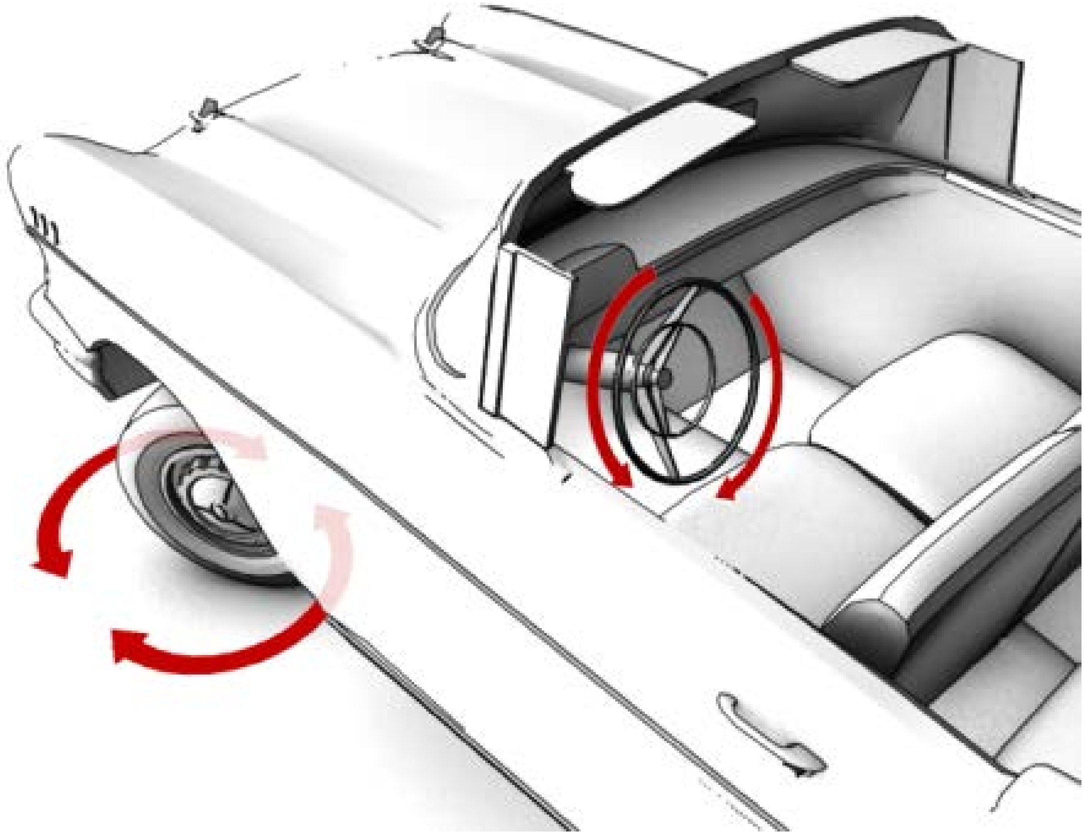
DATA STRUCTURE

- **Data**: is value or a set of values of deferent types (string, int, float…etc)
- **Structure**: a way of organizing information, so that it is easier to use
- So in a simple words we can define data structure as
  - It's a way organizing data in such a way so that data can be easier to use.
- Data Structure is a systematic way to organize data in order to use it efficiently.
- More precisely, a data structure is **a collection of data values**, the **relationship among them**, and the **operation** that can be applied to the data.

ABSTRACT DATA TYPES (ADT)

- ADT is a type defined in terms of its **data items and associated operations**, not its implementation.
- ADT is a type for objects whose behavior is defined by a **set of value** and a **set of operations**.
- The definition of ADT only mentions what operations are to be performed but not
  - how these operations will be implemented.
  - how data will be organized in memory and
  - what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation independent view.
- Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and Booleans are data types.
- We can think of ADT as a black box which hides the inner structure and design of the data type.

ADT EXAMPLE



STACK ADT

- A Stack contains elements of **same type** arranged in **sequential order**. All operations takes place at a single end that is **top** of the stack and following operations can be performed:
  - **push()** – Insert an element at one end of the stack called top.
  - **pop()** – Remove and return the element at the top of the stack, if it is not empty.
  - **peek()** – Return the element at the top of the stack without removing it, if the stack is not empty.
  - **size()** – Return the number of elements in the stack.
  - **isEmpty()** – Return true if the stack is empty, otherwise return false.
  - **isFull()** – Return true if the stack is full, otherwise return false.

QUEUE ADT

- A Queue contains elements of same type arranged in sequential order. Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:
  - **enqueue()** – Insert an element at the end of the queue.
  - **dequeue()** – Remove and return the first element of queue, if the queue is not empty.
  - **peek()** – Return the element of the queue without removing it, if the queue is not empty.
  - **size()** – Return the number of elements in the queue.
  - **isEmpty()** – Return true if the queue is empty, otherwise return false.
  - **isFull()** – Return true if the queue is full, otherwise return false.

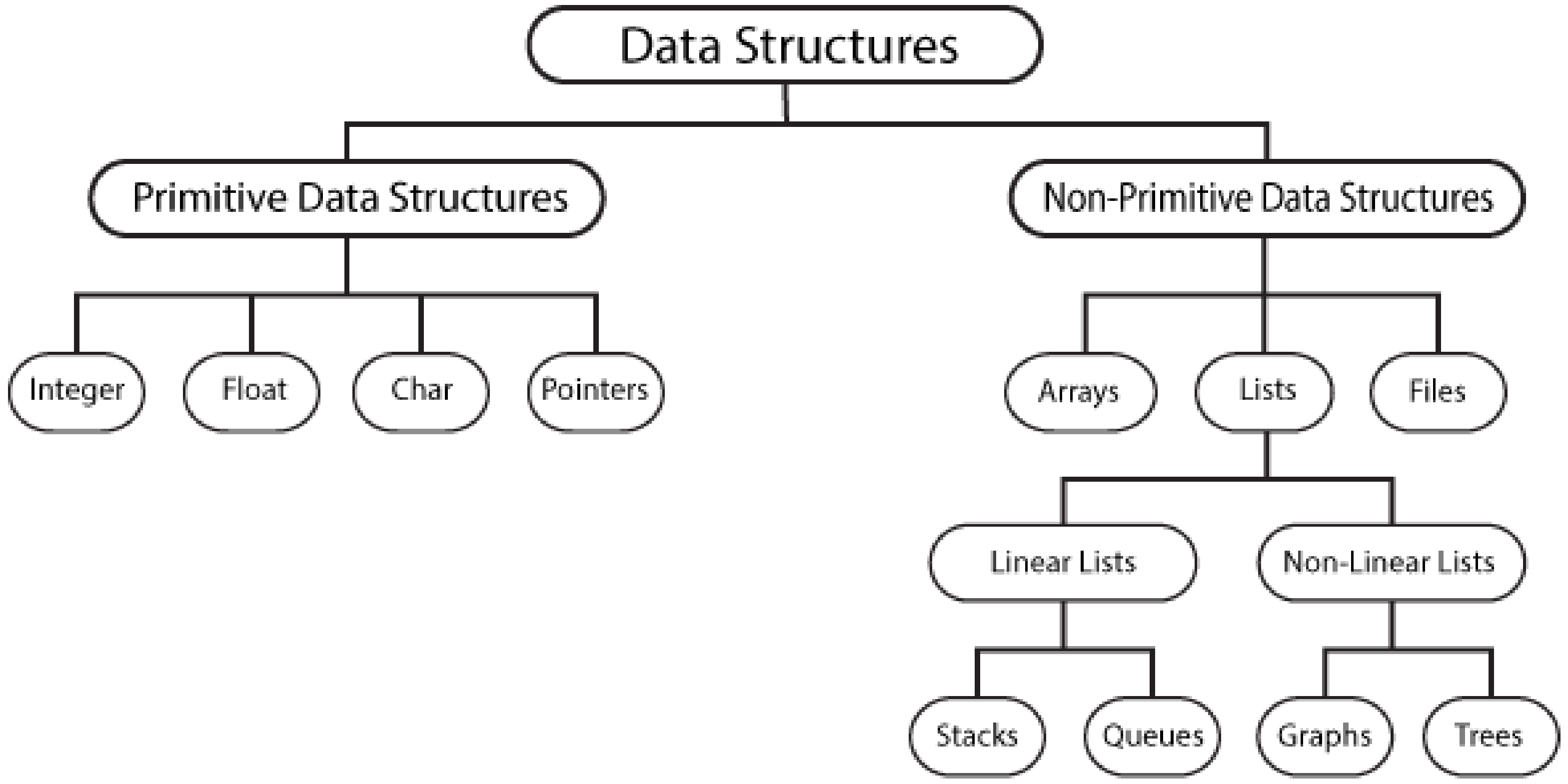
LIST ADT

- A list contains elements of same type arranged in sequential order and following operations can be performed on the list.
  - **get()** – Return an element from the list at any given position.
  - **insert()** – Insert an element at any position of the list.
  - **remove()** – Remove the first occurrence of any element from a non-empty list.
  - **removeAt()** – Remove the element at a specified location from a non-empty list.
  - **replace()** – Replace an element at any position by another element.
  - **size()** – Return the number of elements in the list.
  - **isEmpty()** – Return true if the list is empty, otherwise return false.
  - **isFull()** – Return true if the list is full, otherwise return false.

CLASSIFICATION OF DATA STRUCTURES

- **Primitive data structure**:- used to represent the standard data types of any one of computer language (int, char, float…..etc.)
  - Built in type
  - Used to represent single value
  - Directly operated upon by machine level instructions
- **Non-Primitive data structure**:- It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
  - can be constructed with the help of primitive data structure.
  - Used to store group of values
  - Arrays, linked list, stack, queue…etc.

CLASSIFICATION OF DATA STRUCTURES



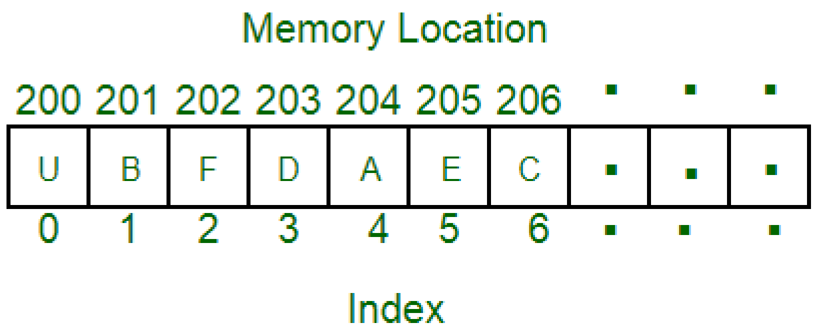
CLASSIFICATION OF DATA STRUCTURES

- Linear data structure: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists

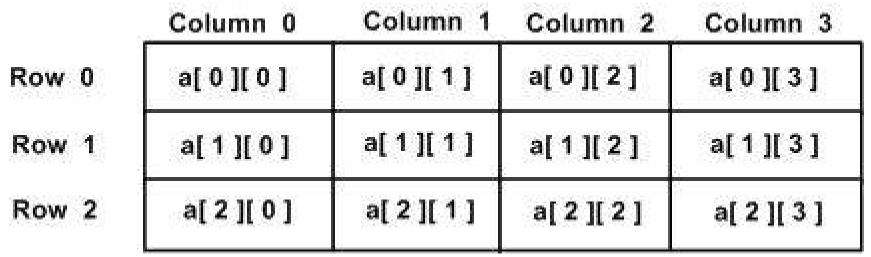
- Non-Linear data structure: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs

ARRAY REVISION

- Stores a fixed-size sequential collection of elements (stored at contiguous memory locations) of the same type.
- A specific element in an array is accessed by an index. (starts from 0)



- To declare an array in C++,
  - **type arrayName [arraySize];**
- The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type.
- C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration −
  - type name[size1][size2]...[sizeN];

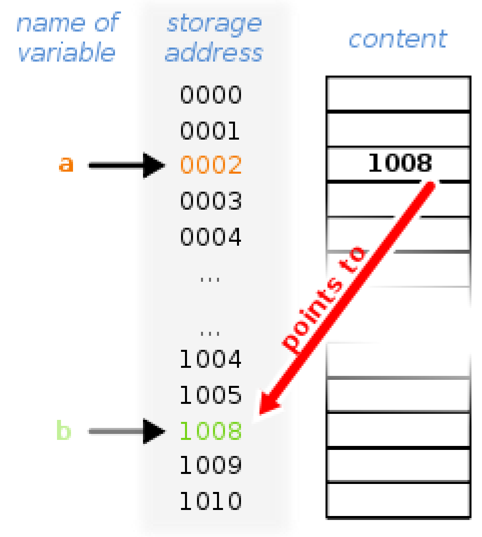


	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

POINTER REVISION

- a pointer is a data structure that stores the memory address of another value located in computer memory.
- type *var-name;
- The actual data type, is the same, a long

  hexadecimal number that represents a memory address.
- * that returns the value of the variable located at the address



ALGORITHMS

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.
- Algorithms are generally created **independent of underlying languages,** i.e. an algorithm can be implemented in more than one programming language.
  - Search, sort, insert, update, delete ………..
- For a single problem there could be more than one algorithms.

WRITE AN ALGORITHM FOR MAKING A CUP OF TEA ?

AN ALGORITHM FOR MAKING A CUP OF TEA

1. Put the teabag in a cup.
2. Fill the kettle with water.
3. Boil the water in the kettle.
4. Pour some of the boiled water into the cup.
5. Add sugar to the cup.
6. Stir the tea.
7. Drink the tea.

PROPERTIES OF AN ALGORITHM

- An algorithm should have the following characteristics −
  - **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
  - **Input** − An algorithm should have 0 or more well-defined inputs.
  - **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
  - **Finiteness** − Algorithms must terminate after a finite number of steps.
  - **Feasibility** − Should be feasible with the available resources.
  - **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.
  - **Correctness:** should produce the correct output

  - **Generality**: the algorithm should be applicable to all problems of a similar form

EXAMPLE 1

- Problem: Given a list of positive numbers, return the largest number on the list.

  Algorithm:

  1. Set `max` to 0.
  2. For each number x in the list L, compare it to Max. If x is larger, set Max to x.
  3. Max is now set to the largest number in the list.

- Does this meet the criteria for being an algorithm?

EXAMPLE 2

- Problem: Given two numbers from a user, return the division of the two numbers.

  Algorithm:

  1. Set Res to 0.
  2. Get number one from the user.
  3. Get number two from the user.
  4. Res now set to the result of number one divided by number two.
  5. Display Res to the user.

- Does this meet the criteria for being an algorithm?

EXAMPLE 3

- Problem: Given a name of person from user, Display the name five times.

  Algorithm:

  1. Set num to 5
  2. Declare a string variable Name.
  3. Ask the user to enter the name of person and store it into Name variable.
  4. While num is greater than or equal to zero, display the person name to user.
  5. Finished.

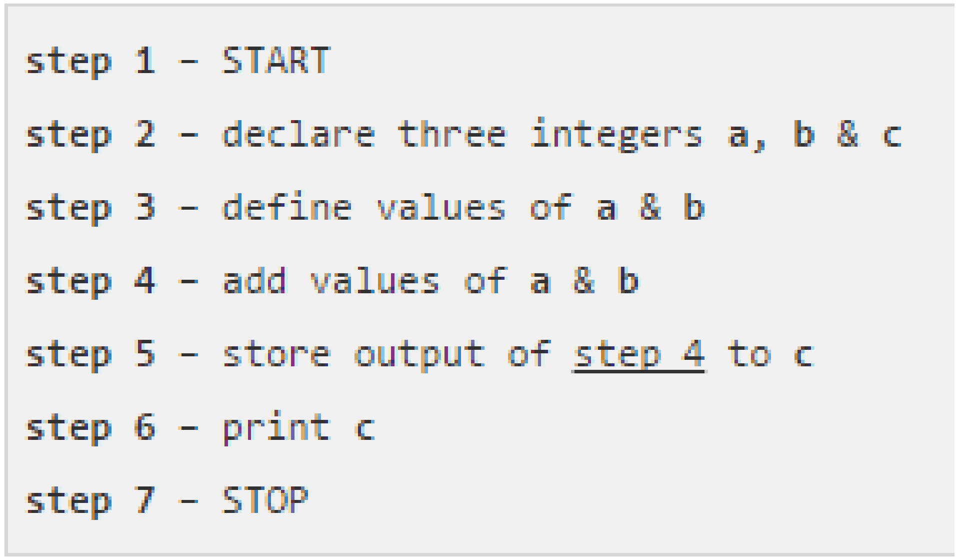
- Does this meet the criteria for being an algorithm?

HOW TO WRITE AN ALGORITHM?

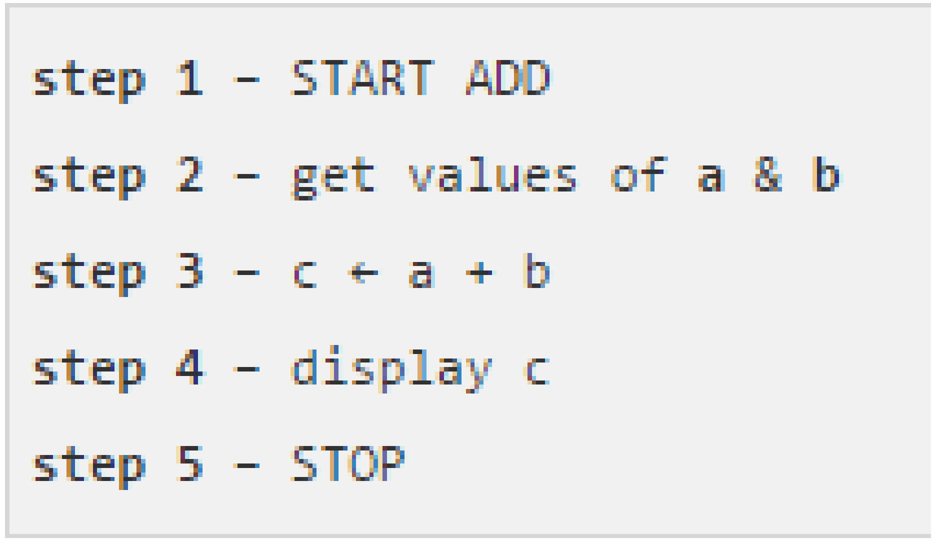
- There are no well-defined standards, Rather, it is problem and resource dependent.
- Algorithms are never written to support a particular programming code.
  - Common constructs can be used to write an algorithm.
- Algorithm writing is a process and is executed after the problem domain is well-defined.

Let's try to learn algorithm-writing by using an example.

- Problem − Design an algorithm to add two numbers and display the result.



```
step 1 - START
step 2 - declare three integers a, b & c
step 3 - define values of a & b
step 4 - add values of a & b
step 5 - store output of step 4 to c
step 6 - print c
step 7 - STOP
```



```
step 1 - START ADD
step 2 - get values of a & b
step 3 - c ← a + b
step 4 - display c
step 5 - STOP
```

COMMON ELEMENTS OF ALGORITHMS

- **Acquire data (input):**
  - Some means of reading values from an external source;
- **Computation:**
  - Some means of performing arithmetic computations, comparisons, testing logical conditions, and so forth...
- **Selection:**
  - Some means of choosing among two or more possible courses of action, based upon initial data, user input and/or computed results
- **Iteration:**
  - Some means of repeatedly executing a collection of instructions, for a fixed number of times or until some logical condition holds
- **Report results (output):**
  - Some means of reporting computed results to the user, or requesting additional data from the user

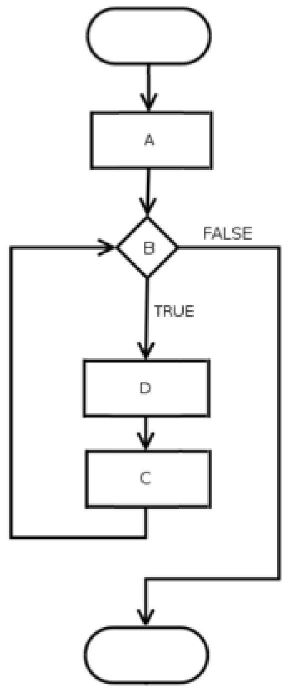
EXPRESSING ALGORITHMS

- Expressing the sequence of steps to performed can be done using
  - **Natural language**
    - Usually verbose and ambiguous
  - **Flow chart**
    - Avoid most (if not all) issues of ambiguity; difficult to modify w/o specialized tools; largely standardized
  - **Pseudocode**
    - Also avoids most issues of ambiguity;
    - Vaguely resembles common elements of programming languages;
    - No particular agreement on syntax
  - **Programming language**
    - Tend to require expressing low-level details that are not necessary for a high-level understanding

FLOWCHART

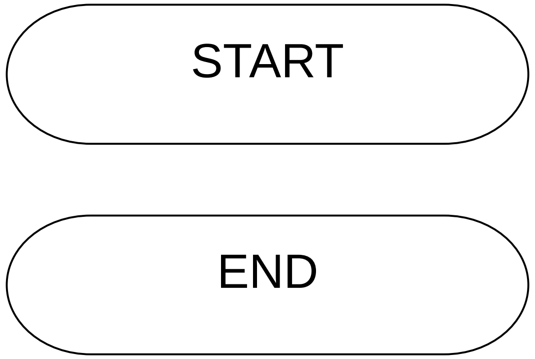
- A flowchart is a type of diagram that represents an algorithm, workflow or process.
- The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows.

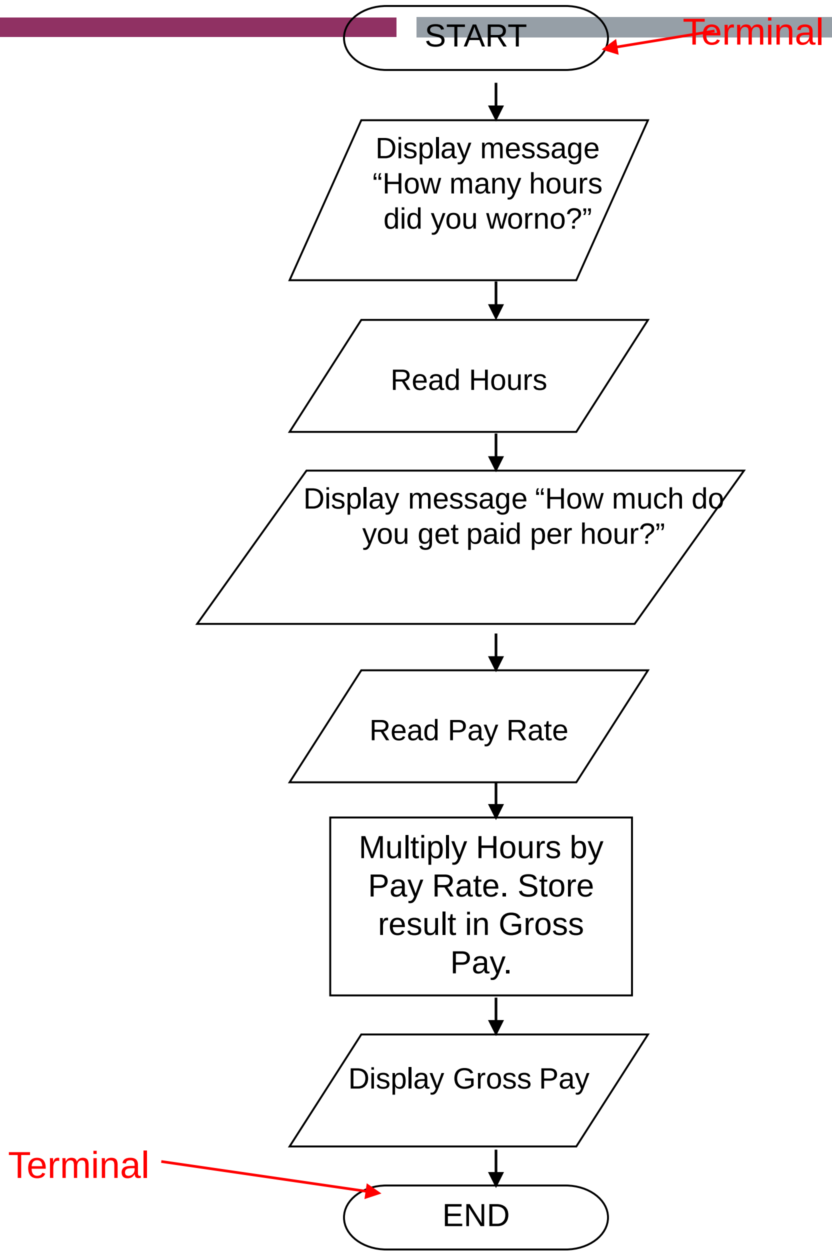
```
for(A;B;C)
  D;
```



BASIC FLOWCHART SYMBOLS

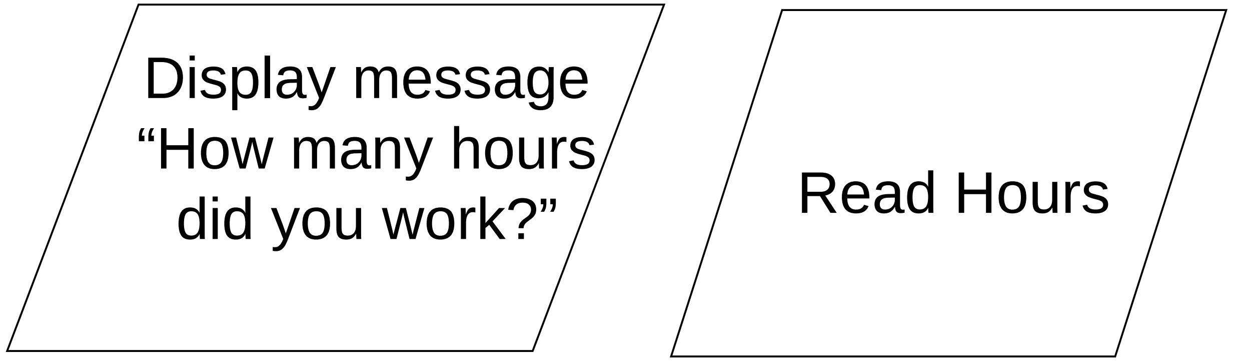
- Terminals
  - represented by rounded rectangles
  - indicate a starting or ending point

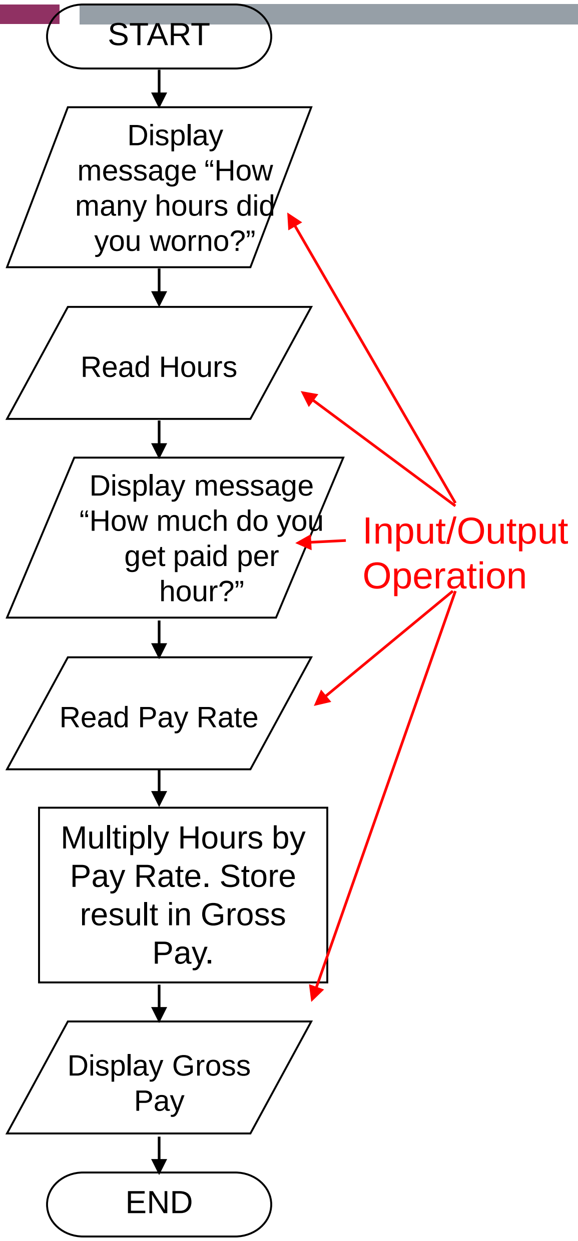



START
Terminal
Display message "How many hours did you worno?"
Read Hours
Display message "How much do you get paid per hour?"
Read Pay Rate
Multiply Hours by Pay Rate. Store result in Gross Pay.
Display Gross Pay
Terminal
END

BASIC FLOWCHART SYMBOLS

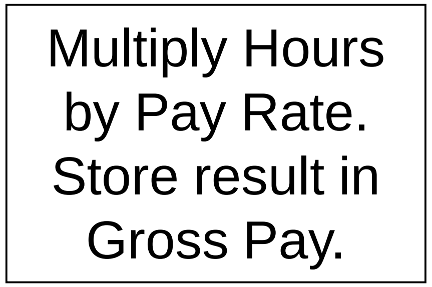
- Input/Output Operations
  - represented by parallelograms
  - indicate an input or output operation

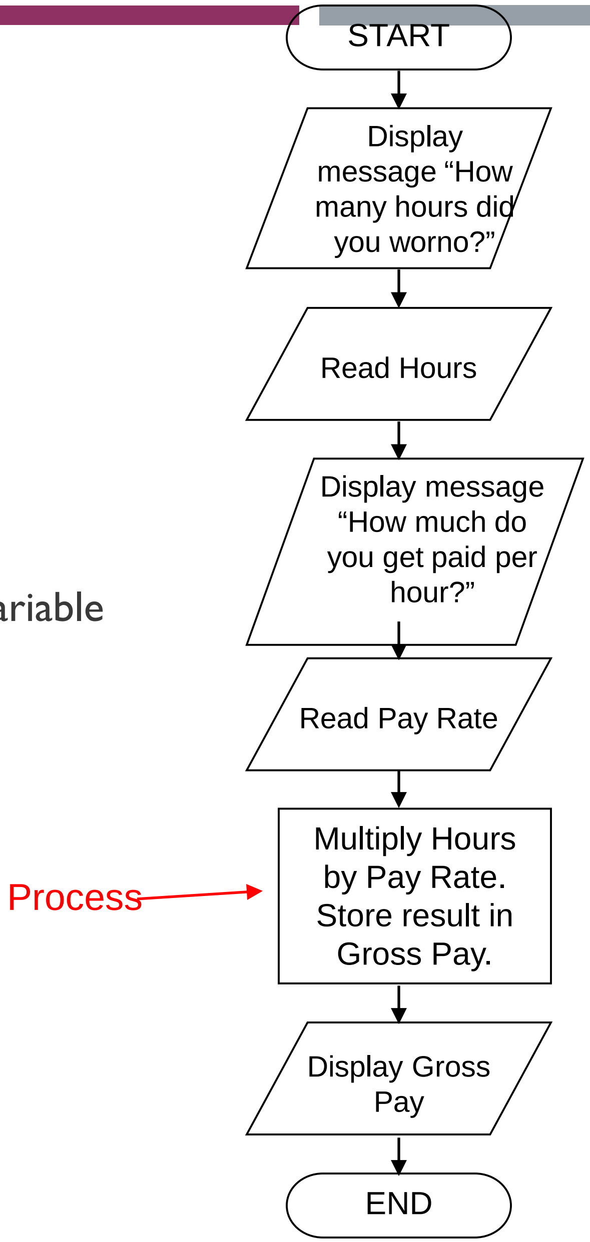



START
Display message "How many hours did you worno?"
Read Hours
Display message "How much do you get paid per hour?"
Read Pay Rate
Multiply Hours by Pay Rate. Store result in Gross Pay.
Display Gross Pay
END
Input/Output Operation

BASIC FLOWCHART SYMBOLS

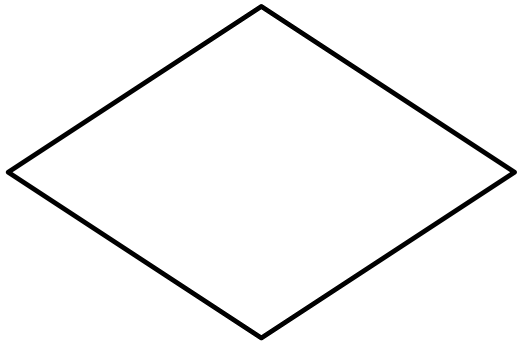
- Processes
  - represented by rectangles
  - indicates a process such as a mathematical computation or variable assignment

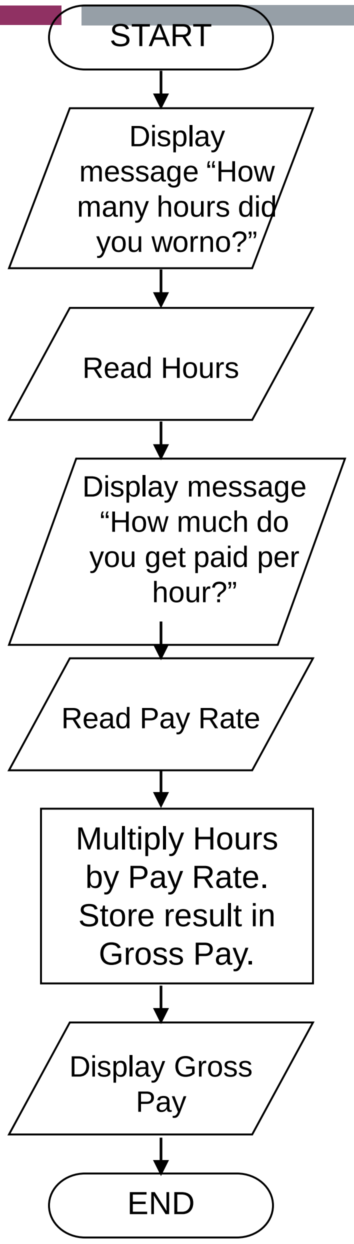



START
Display message “How many hours did you worno?”
Read Hours
Display message “How much do you get paid per hour?”
Read Pay Rate
Process
Multiply Hours by Pay Rate. Store result in Gross Pay.
Display Gross Pay
END

BASIC FLOWCHART SYMBOLS

- Decision
  - represented by Diamond
  - A diamond indicates a decision



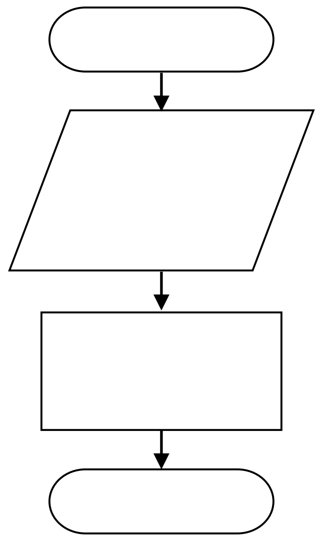

START
Display message "How many hours did you worno?"
Read Hours
Display message "How much do you get paid per hour?"
Read Pay Rate
Multiply Hours by Pay Rate. Store result in Gross Pay.
Display Gross Pay
END

FOUR FLOWCHART STRUCTURES

- Sequence
- Decision
- Repetition
- Case

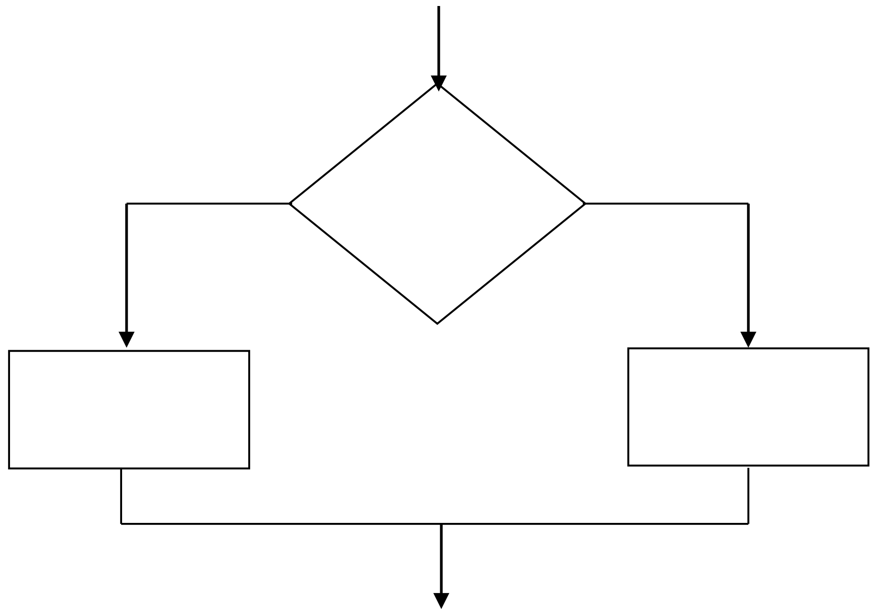
SEQUENCE STRUCTURE

- a series of actions are performed in sequence
- The pay-calculating example was a sequence flowchart.



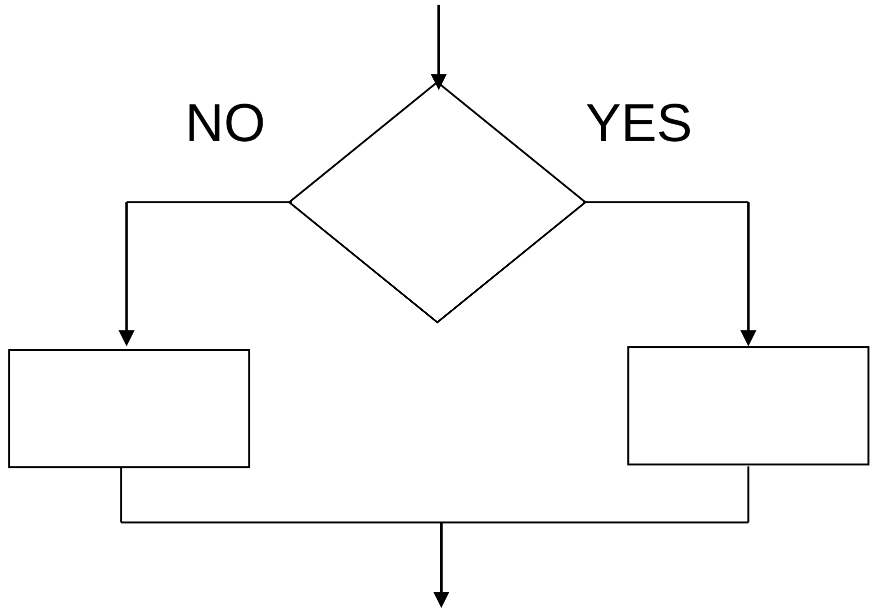
DECISION STRUCTURE

- One of two possible actions is selected, depending on a condition.



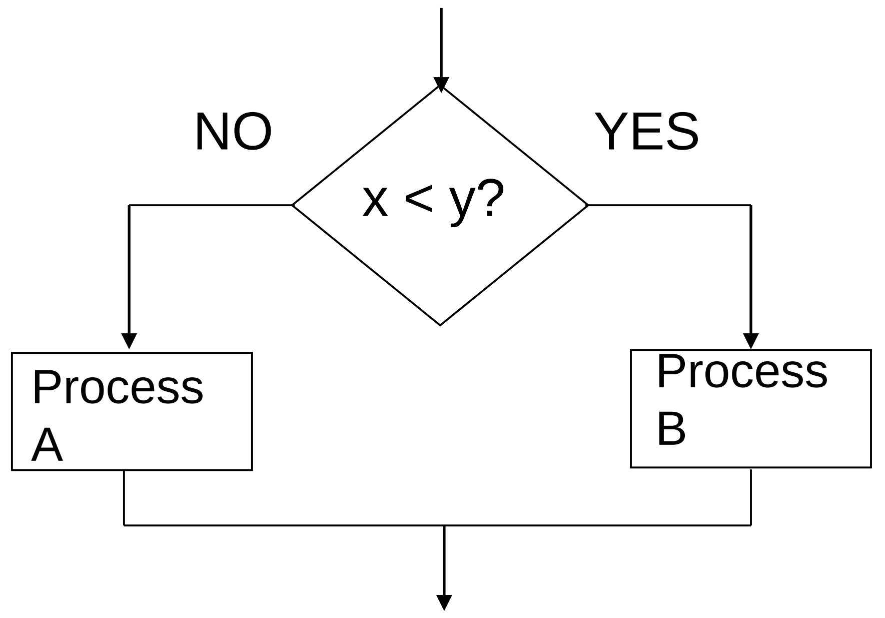
DECISION STRUCTURE

- the diamond symbol indicates a yes/no question. If the answer to the question is yes, the flow follows one path. If the answer is no, the flow follows another path



DECISION STRUCTURE

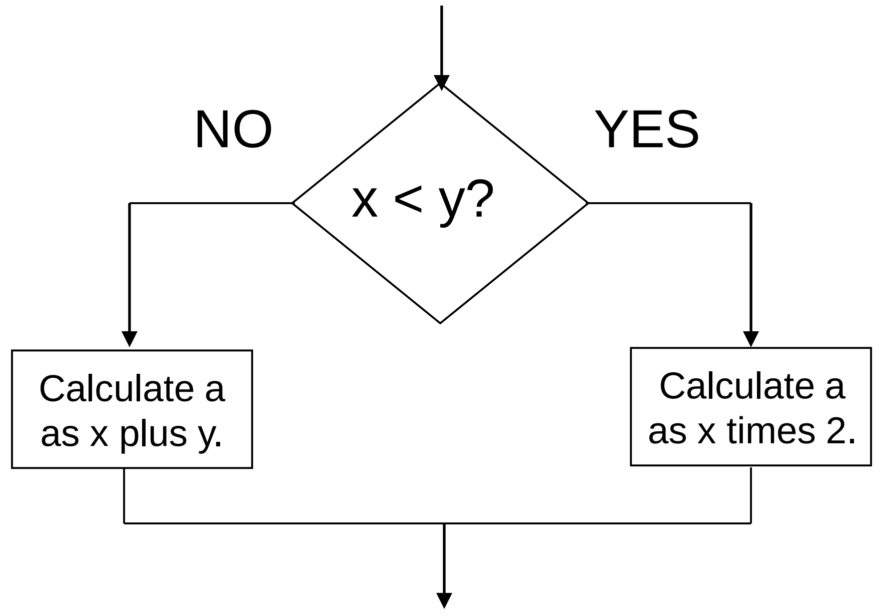
- In the flowchart segment below, the question "is x < y?" is answered. If the answer is no, then process A is performed. If the answer is yes, then process B is performed.



DECISION STRUCTURE

- The flowchart segment below shows how a decision structure is expressed in C++ as an if/else statement.

Flowchart



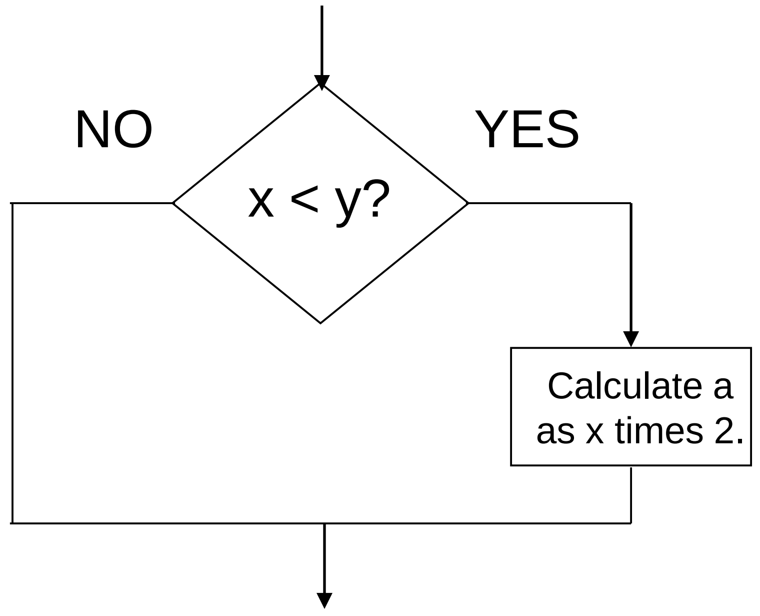
C++ Code

```
if (x < y)
      a = x * 2;
else
      a = x + y;
```

DECISION STRUCTURE

- The flowchart segment below shows a decision structure with only one action to perform. It is expressed as an if statement in C++ code.

Flowchart

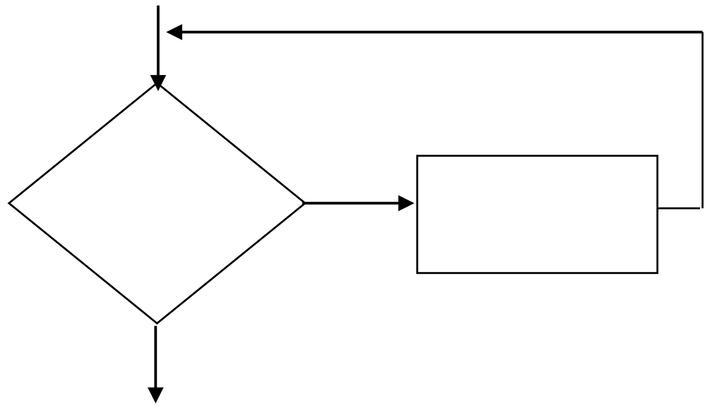


C++ Code

```
if (x < y)
    a = x * 2;
```

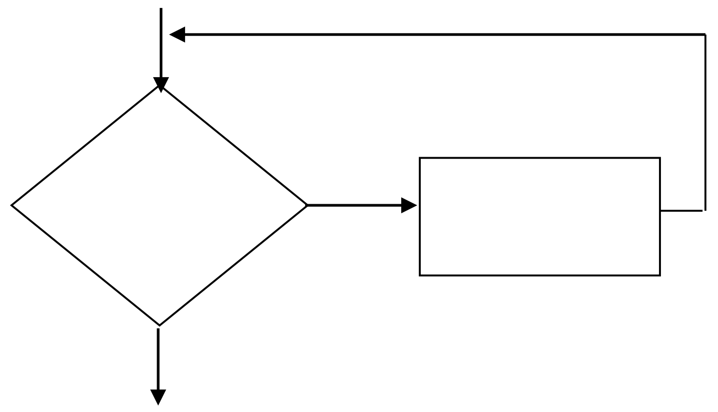
REPETITION STRUCTURE

- A repetition structure represents part of the program that repeats. This type of structure is commonly known as a loop.



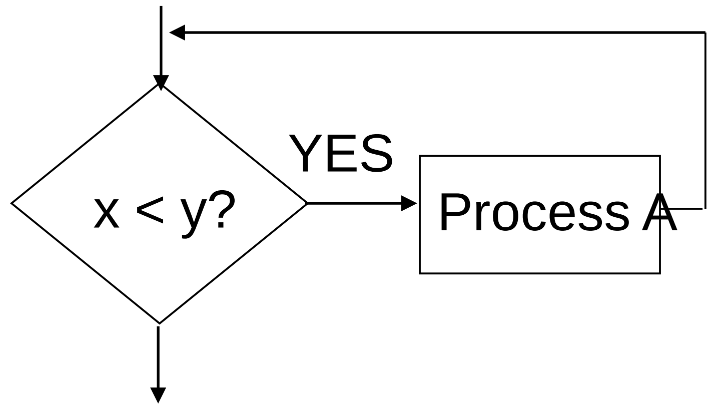
REPETITION STRUCTURE

- Notice the use of the diamond symbol. A loop tests a condition, and if the condition exists, it performs an action. Then it tests the condition again. If the condition still exists, the action is repeated. This continues until the condition no longer exists.



REPETITION STRUCTURE

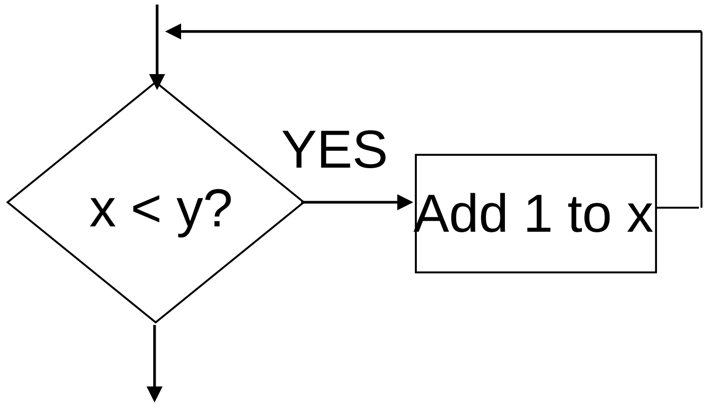
- In the flowchart segment, the question "is x < y?" is answered. If the answer is yes, then Process A is performed. The question "is x < y?" is answered again. Process A is repeated as long as x is less than y. When x is no longer less than y, the repetition stops and the structure is exited.



REPETITION STRUCTURE

- The flowchart segment below shows a repetition structure expressed in C++ as a while loop.

Flowchart

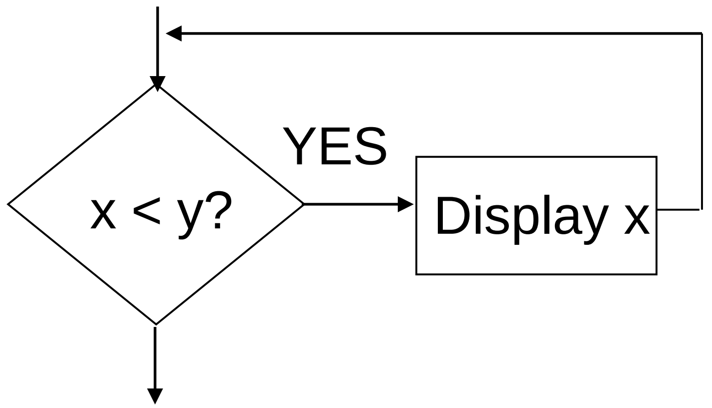


C++ Code

```
while (x < y)
    x++;
```

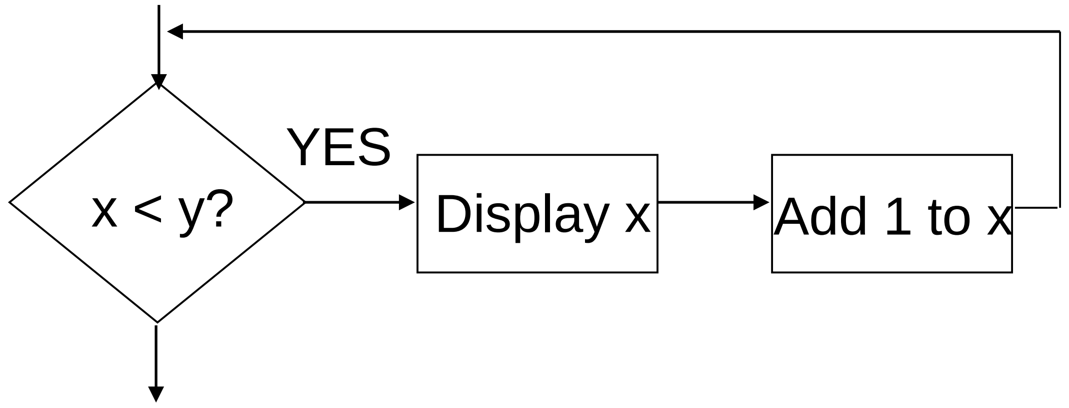
CONTROLLING A REPETITION STRUCTURE

- The action performed by a repetition structure must eventually cause the loop to terminate. Otherwise, an infinite loop is created.
- In this flowchart segment, x is never changed. Once the loop starts, it will never end.
- QUESTION: How can this flowchart be modified so it is no longer an infinite loop?



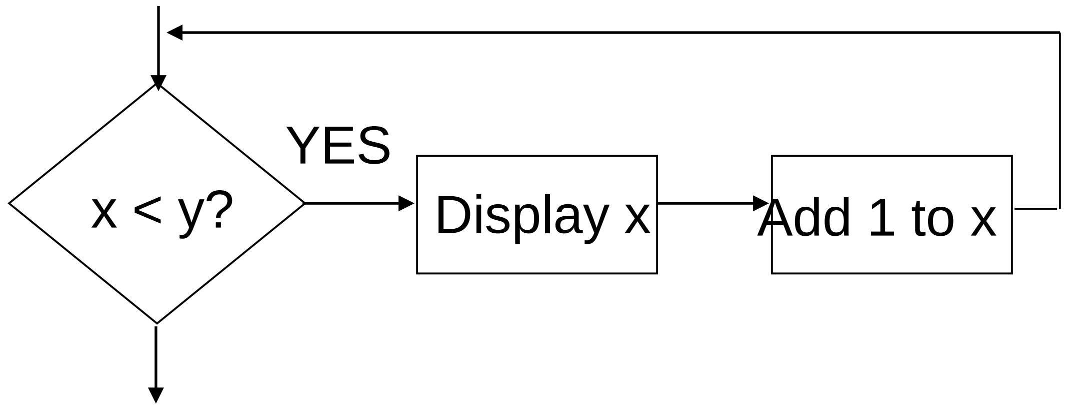
CONTROLLING A REPETITION STRUCTURE

- ANSWER: By adding an action within the repetition that changes the value of x.



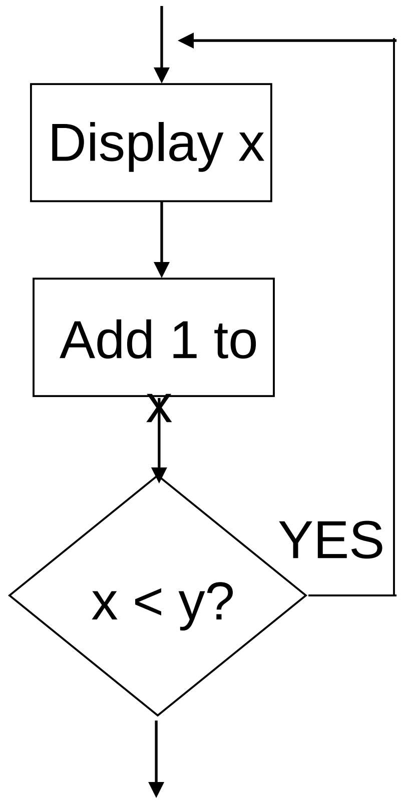
A PRE-TEST REPETITION STRUCTURE

- This type of structure is known as a pre-test repetition structure. The condition is tested *BEFORE* any actions are performed.
  - if the condition does not exist, the loop will never begin.



A POST-TEST REPETITION STRUCTURE

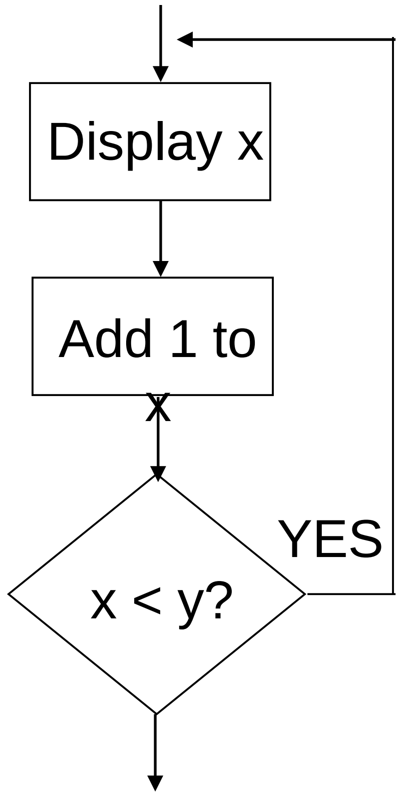
- This flowchart segment shows a post-test repetition structure.
- The condition is tested *AFTER* the actions are performed.
- A post-test repetition structure always performs its actions at least once.



A POST-TEST REPETITION STRUCTURE

- The flowchart segment below shows a post-test repetition structure expressed in C++ as a do-while loop.

Flowchart

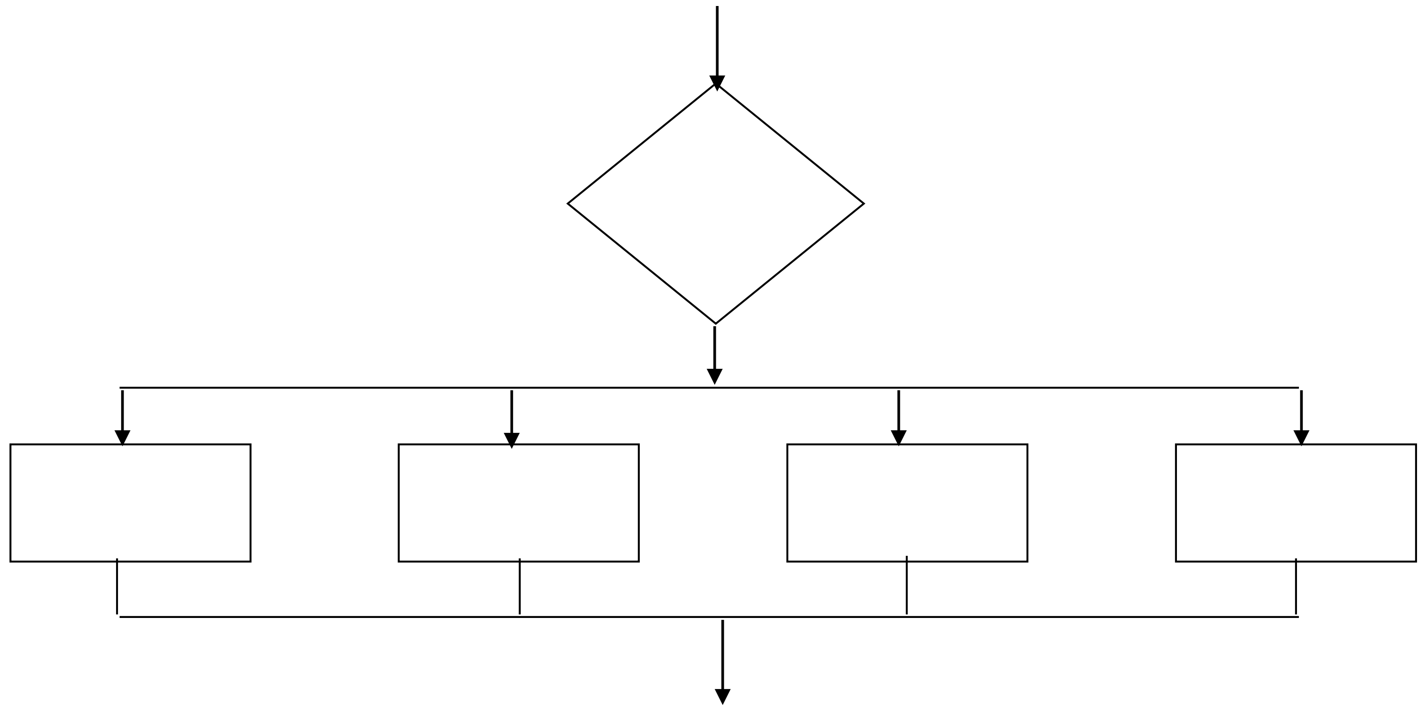


C++ Code

```
do
{
        cout << x << endl;
        x++;
} while (x < y);
```

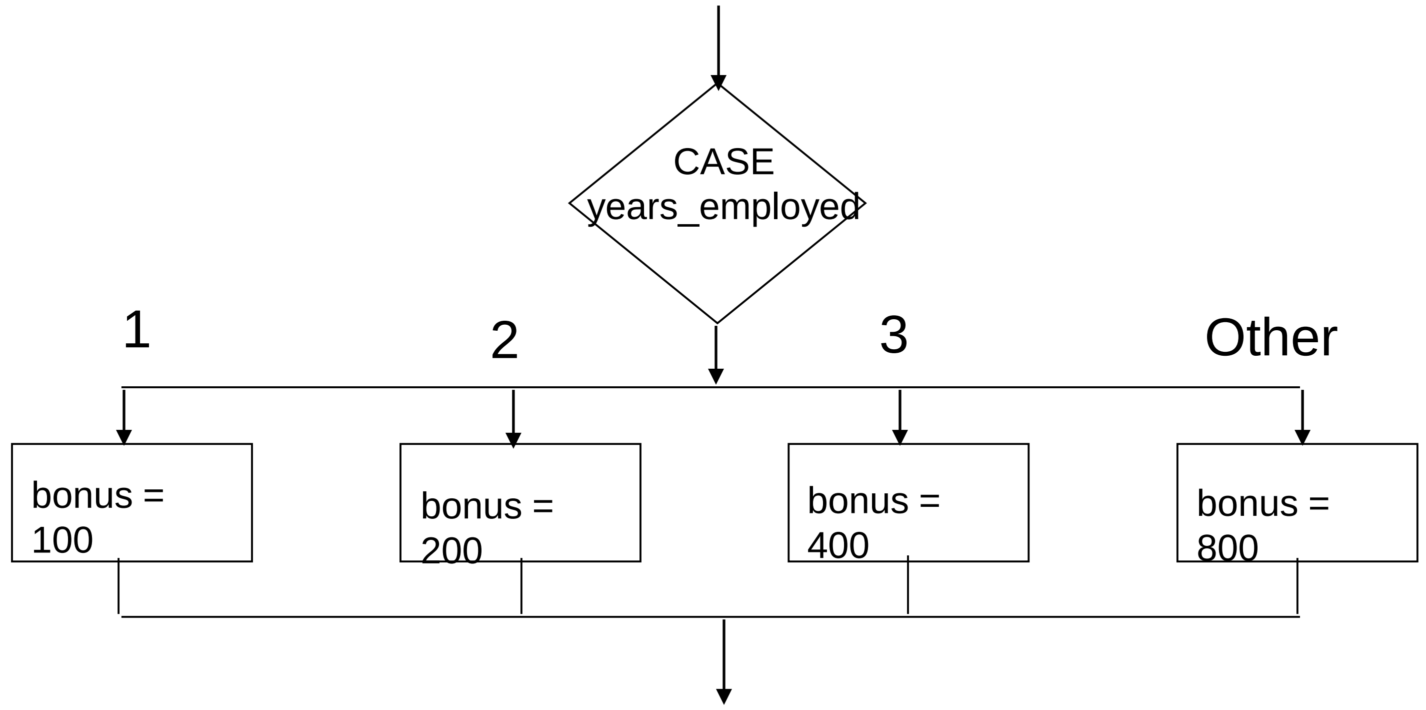
CASE STRUCTURE

- One of several possible actions is selected, depending on the contents of a variable.

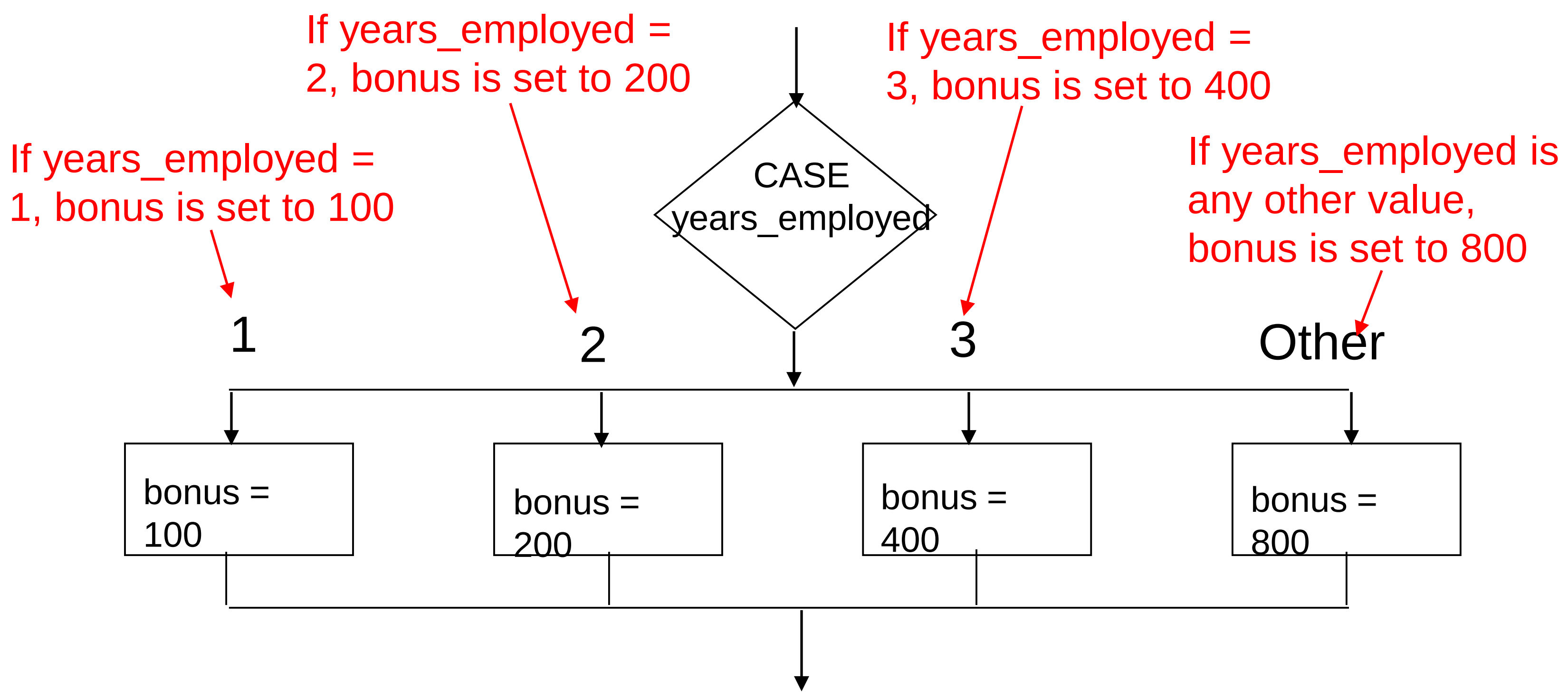


CASE STRUCTURE

- The structure below indicates actions to perform depending on the value in years_employed.

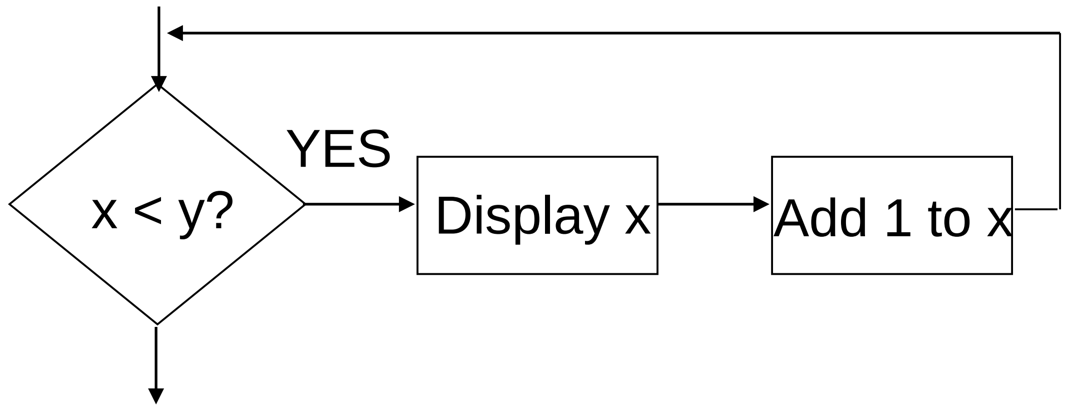


CASE STRUCTURE


If years_employed = 2, bonus is set to 200
If years_employed = 3, bonus is set to 400
If years_employed = 1, bonus is set to 100
If years_employed is any other value, bonus is set to 800
CASE years_employed
1
2
3
Other
bonus = 100
bonus = 200
bonus = 400
bonus = 800

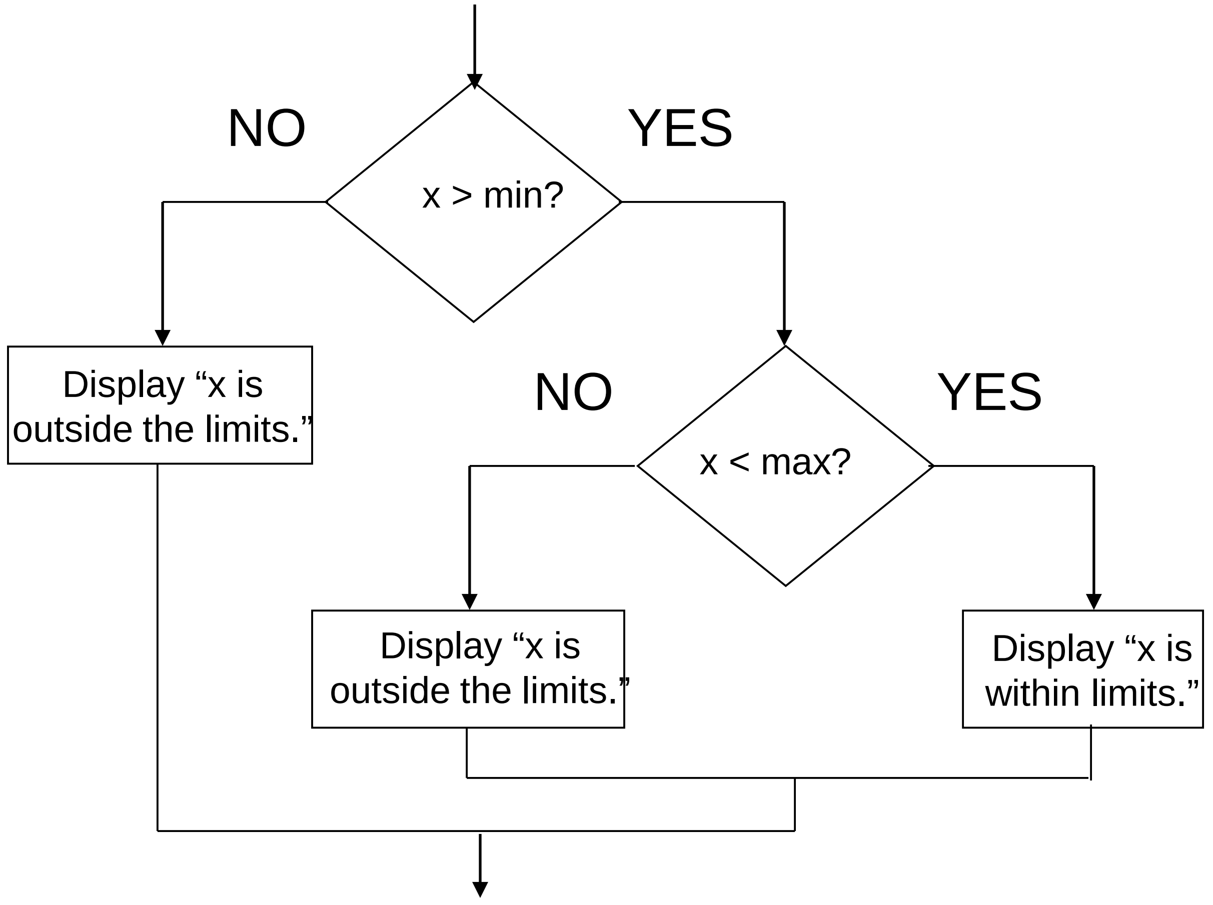
COMBINING STRUCTURES

- Structures are commonly combined to create more complex algorithms.
- The flowchart segment below combines a decision structure with a sequence structure.

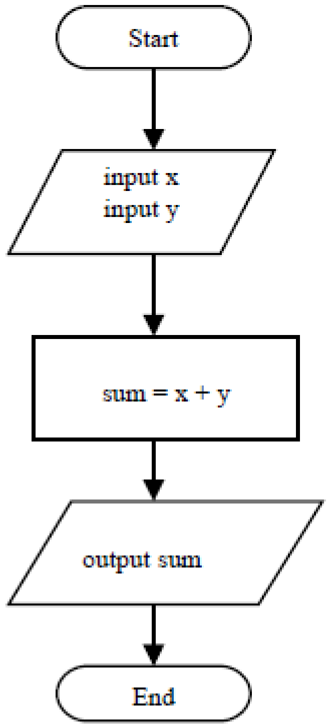


COMBINING STRUCTURES

- This flowchart segment shows two decision structures combined.

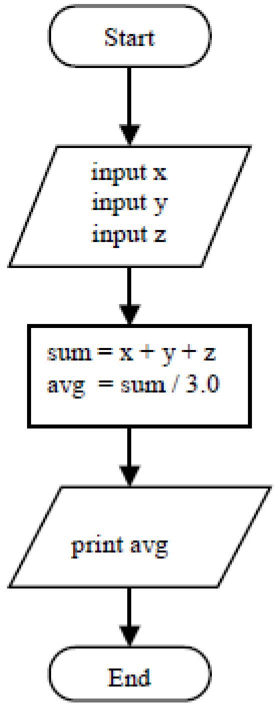

NO
x > min?
YES
Display "x is outside the limits."
NO
x < max?
YES
Display "x is outside the limits."
Display "x is within limits."

ALGORITHM FOR: **SUM OF 2 NUMBERS**



```
Begin
    input x, y
    sum = x + y
    print sum
End
```

ALGORITHM FOR: **AVERAGE OF 3 NUMBERS**



```
Begin
    input x
    input y
    input z
    sum = x + y + z
    avg = sum / 3.0
    print avg
End
```