Faculty of Engineering & Information Technology

Algorithms Analysis and Design

230213150

Thaer Thaher
Thaer.Thaher@aaup.edu

Fall 2023/2024

Chapter1: Introduction

Prerequisites

The course assumes that a student has gone through

- An introductory programming course.
- Fundamental data structures.
- Mathematics Background (summation formulas , recurrence relations …)

With such a background, student should be able to handle the course's material without undue difficulty

What is a computer program

- A computer program is a set of instructions written in a programming language to perform a specific task.
- An implementation of code to instruct a computer on how to execute a specific task.
- Some of the popular programming languages include **Python, Java, C/C++, C#, Visual Basic, JavaScript, PHP, and Ruby**.

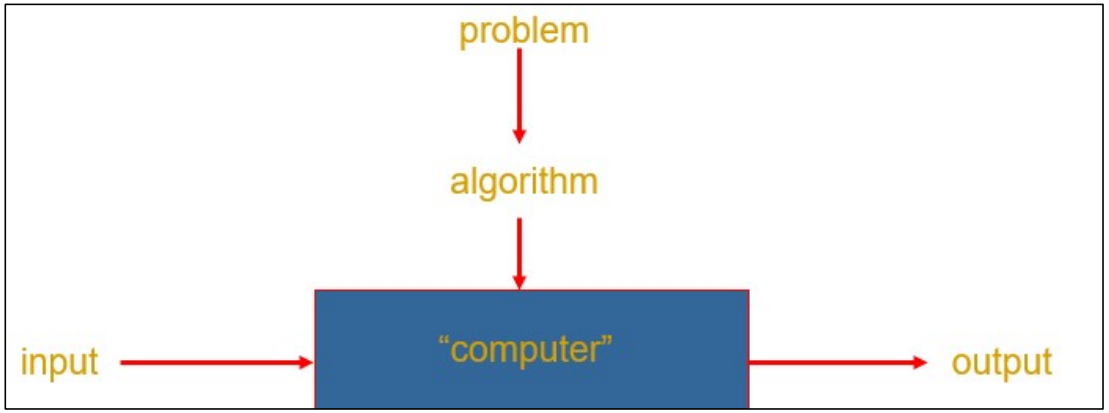
Software development process

❑ **Stages of program development:**

1. **Specify** the problem requirements: Understand what the software should do.
2. **Analyze** the problem (inputs, processing, outputs).
3. **Design** the algorithms (steps) to solve the problem.
4. **Implement** the algorithm (write code).
5. **Test** and verify the completed program.
6. **Maintain** and update the program.

What is an algorithm

- An **algorithm** is a finite set of precise instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
- Step-by-step problem-solving process where solution achieved in a finite amount of time.
- It is also defined as a systematic approach to solving a specific problem



What is an algorithm

- Algorithms are written according to a set of rules that define how a task is to be executed to get the expected results.
- Algorithms are conceptual and can be described using Pseudocode or flowcharts.
- An understanding of algorithms is essential for programmers to program more efficiently.
- For a specific problem, **several algorithms may exist**.
- We can implement algorithms in different programming languages.

Example

For example, here's an algorithm to add two numbers:

Start

Take two number inputs

Add both the numbers using the + operator

عرض

Display the result

End

Characteristics of Algorithm

Algorithms have the following main properties:

- **Input**
  - It should take valid and well-defined inputs (uses values from a specified list)
- **Output**:
  - It should produce the correct output given a valid input.
- **Precision / definiteness**:
  - the steps are precisely stated (should be unambiguous).
- **Finiteness**:
  - the algorithm terminates after a finite number of steps.
- **Effectiveness**:
  - steps are sufficiently simple and basic. You should not write unnecessary statements

Characteristics of Algorithm

Other properties

الحتمية

- **Determinism**:
  - the intermediate results of each step of execution are
    - unique and
    - are determined only by the inputs and results of the preceding steps

صحة

- **Correctness**:
  - the output is correct for each input as defined by the problem.
- **Generality**:
  - the algorithm should be applicable a set of inputs (not a special subset).

ملائم مجموعة فرعية

- It should be **language-independent**.
  - Write the line of codes independent of all language specific words, terms or notation.

المصطلح أو التدوين

Example

Example:

Problem: Finding the max of 3 numbers (a, b, c)

Algorithm:

1. x=a
2. If b>x, then x=b
3. If c>x, then x=c

Verify that our algorithm has the properties listed previously.

Example: Euclid's Algorithm

- **Problem**: Find gcd(*m,n*), the greatest common divisor of two nonnegative, not both zero integers *m* and *n*

Recall: The greatest common divisor (gcd) of two nonzero integers *m* and *n* is the greatest positive integer *d* such that *d* is a divisor of both *a* and *b*

- Examples: gcd (12, 8) = 4 gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

- **Euclid's algorithm** is based on repeated application of equality

  gcd(*m,n*) = gcd(*n*, *m* mod *n*)

  until the second number becomes 0, which makes the problem trivial.

- **Example**: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Two descriptions of Euclid's algorithm

- Step 1 If $n = 0$, return m and stop; otherwise go to Step 2
- Step 2 Divide m by n and assign the value fo the remainder to r
- Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

```
ALGORITHM Euclid ( m , n)
{
    WHILE n ≠ 0 do
        r ← m mod n
        m← n
        n ← r
    ENDWHILE
    RETURN m
}
```

Other methods for computing gcd(m,n)

Consecutive integer checking algorithm

- Step 1 Assign the value of min{m,n} to t
- Step 2 Divide m by t. If the remainder is 0, go to Step 3;
  otherwise, go to Step 4
- Step 3 Divide n by t. If the remainder is 0, return t and stop;
  otherwise, go to Step 4
- Step 4 Decrease t by 1 and go to Step 2

Other methods for gcd(m,n) [cont.]

Middle-school procedure

- Step 1 Find the prime factorization of m
- Step 2 Find the prime factorization of n
- Step 3 Find all the common prime factors
- Step 4 Compute the product of all the common prime factors

  and return it as gcd(m,n)

Note: Every integer greater than 1 is either a prime number or can be written as a product of its prime factors

```
60 = 2 . 2 . 3 . 5

24 = 2 . 2 . 2 . 3

gcd(60, 24) = 2 . 2 . 3 = 12.
```

For a specific problem, **several algorithms may exist**.

Important problem types

Main problem	
Sorting	Insertion sort, selection sort, bubble sort, heap sort, quick sort, merge sort …….
searching	Linear/sequential search, binary search …..
string processing	String matching, string sorting, text indexing …
graph problems	shortest path, travelling salesman problem (TSP), graph-coloring problem,…..
combinatorial problems	Knapsack, TSP, Graph coloring
geometric problems	closest-pair problem, convex-hull problem
numerical problems	solving system of equations, computing definite integrals ….

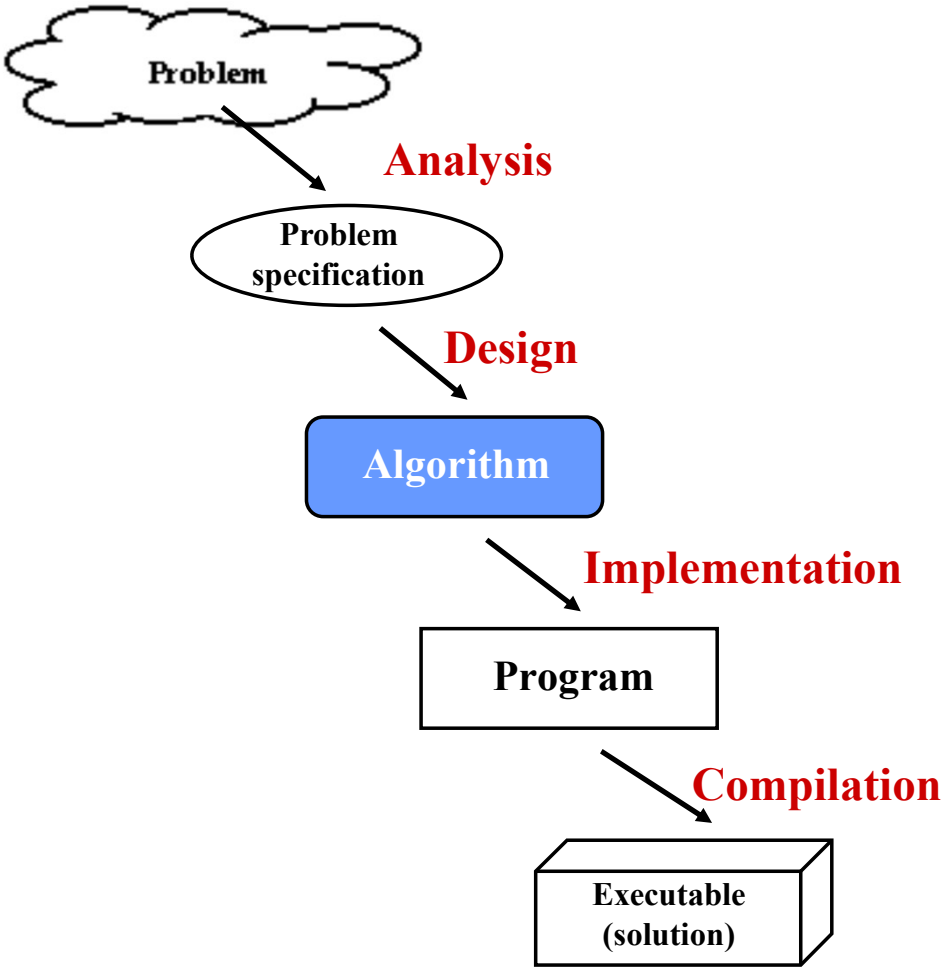
Differences Between Algorithm and Program

Algorithm	Program
Design phase	Implementation phase
Domain knowledge	programmer
Written using plain English language, mathematical notations, pseudocode (language-independent**)	Written in any programming language (C++, Java …)
Independent of hardware and OS	Depend on hardware and OS
can be understood by those from a non-programming background	can be understood by programmers
Analyzing	Testing

Recall software development life cycle

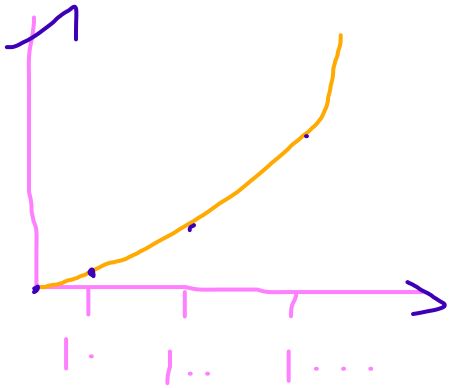
** An algorithm is the thing which stays the same whether the program is Java or C++…etc.

The Problem-solving Process



Two main aspects in the study of algorithms

- **Designing** an algorithm to solve a problem
  - How to design algorithms
- **Analyzing** algorithms
  - How to analyze algorithm efficiency
    - Theoretical Analysis → $T(n) = n^2$
    - Empirical Analysis → زي ما حملنا في assignment 1



Algorithms Analysis and Design

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Why Algorithm design techniques?

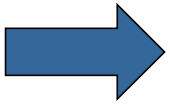
There are three principal reasons for emphasis on algorithm design techniques

- These techniques provide a student with tools for designing algorithms for new problems [General problem solving tools]
- algorithm design techniques have utility as general problem solving strategies.
- they seek to classify multitudes of known algorithms according to an underlying design idea.

Analysis of algorithms

In the analysis of algorithms, we ask the following questions:

- How good is the algorithm?
  - **Correctness**
    - Does the algorithm solve the problem?
  - **Termination**
    - Does the algorithm always stop after a finite number of steps?
  - **Time efficiency**
    - how many instructions does the algorithm execute?
  - **Space efficiency**
    - how much memory does the algorithm need to execute?



In this course, we are concerned primarily with the time analysis. Why?

Analysis of algorithms [cont.]

In the analysis of algorithms, we ask the following questions:

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Representing Algorithm

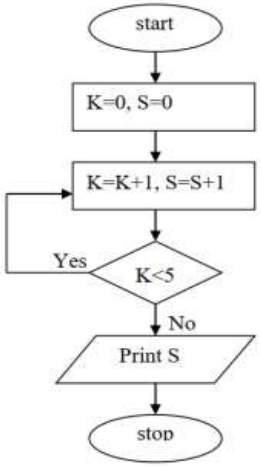
Representing Algorithms

There are two main ways that algorithms can be represented – **pseudocode** and **flowcharts**.

- **Pseudo code**
  - Steps written in a human language to describe the solution.
- **Flow chart:**
  - Graphical symbols that describes the flow of execution of the solution.

Example:

```
1. Set K and S to 0.
2. Let K=K+1.
3. Let S=S+K.
4. If K<5, go to step 2.
5. Print S.
6. Stop.
```

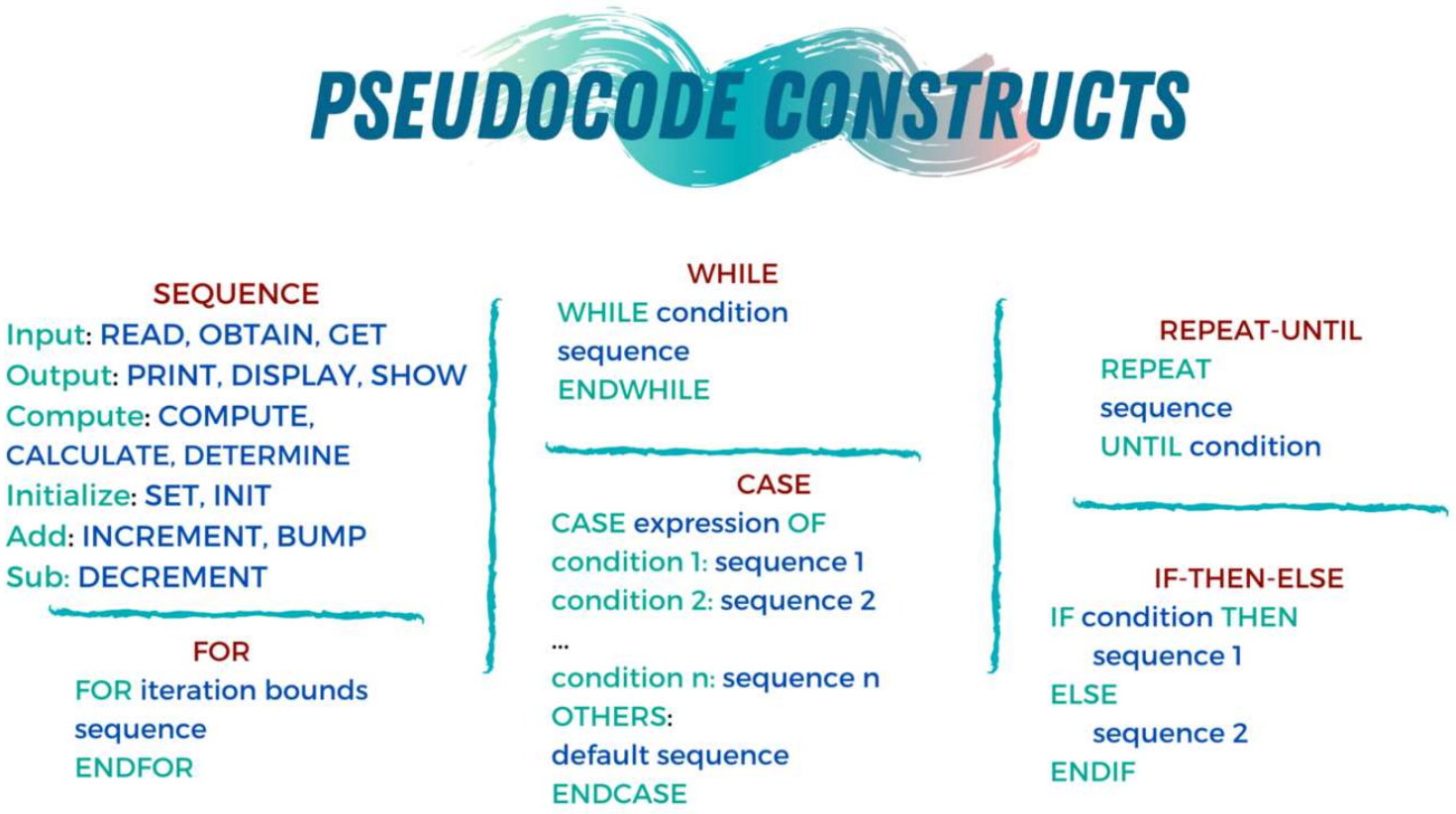


Pseudocode for algorithms

An algorithm can be written in many ways:

- English or any language
- **Pseudocode**
  - **A way of writing program descriptions that is similar to programming languages but may include English descriptions and does not have a precise syntax.**
- Why use Pseudocode?
  - **Ease up code construction**: It's one of the best approaches to start implementation of an algorithm.
  - **Better readability**: can be understood by those from a non-programming background.
  - **Act as a start point for documentation**

The main constructs of Pseudocode



Pseudocode example

Some Rules of writing Pseudocode

- **capitalize** the reserved commands (keywords) IF, ELSE, FOR ……..
- Keep it **simple**, **concise**, and **readable**.
- Have only **one** statement per line.
- **Indent** to show hierarchy: All statements showing. "dependency" are to be indented. These include while, do, for, if, switch.
- Always **end** multiline sections using any of the END keywords (*ENDIF*, *ENDWHILE*, etc.).

```
FOR X = 1 to 10
  FOR Y = 1 to 10
    IF gameBoard[X][Y] = 0
      Do nothing
    ELSE
      CALL theCall(X, Y) (recursively)
      counter += 1
    END IF
  END FOR
END FOR
```

Example: Finding the max of 3 numbers

Input: three numbers a, b , c

output: the maximum x

```
Max(a, b, c)
{
    x=a
    IF (b>x)  THEN
        x=b
    IF (c>x) THEN
        x=c
    RETURN x
}
```

```
Max(a, b, c)
Begin
    x=a
    IF (b>x)  THEN
        x=b
    IF (c>x) THEN
        x=c
    RETURN x
End
```

```
Max(a, b, c)
Begin
    x:=a
    IF (b>x)  THEN
        x:=b
    IF (c>x) THEN
        x:=c
    RETURN x
End
```

```
Max(a, b, c)
Begin
    x ← a
    IF (b>x)  Then
        x ← b
    IF (c>x) Then
        x ← c
    RETURN x
End
```

- BEGIN … END can be replaced with { } curly braces
- Different symbols can be used for assignment operator (= , := , ←)

Example: Finding the max value in an array

Input: array A **output**: max

```
Array_Max2(A, size)
{
    max=A[1]
    FOR i=2 to size
        IF (A[i]>max) THEN
            max=A[i]
        ENDIF
    ENDFOR
    RETURN max
}
```

```
Array_Max1(A, size)
{
    max=A[1]
    i=2
    WHILE  i <= size
        IF(A[i]>max) THEN
            max=A[i]
        ENDIF
        i++
    ENDWHILE
    RETURN max
}
```

Example: finding the summation of odd numbers from 1 to a given number > 1

Input: number n **output**: sum

```
Function Sum_odd_1_to_n ( n )
{
    sum = 0
    FOR count = 1 to n step 2
        add count to sum
    RETURN sum
}
```

As pseudocode does not follow a strict systematic or standard way of being written, so don't think of writing pseudocode as a strict rule

It should be written in such a manner to be easily comprehended.

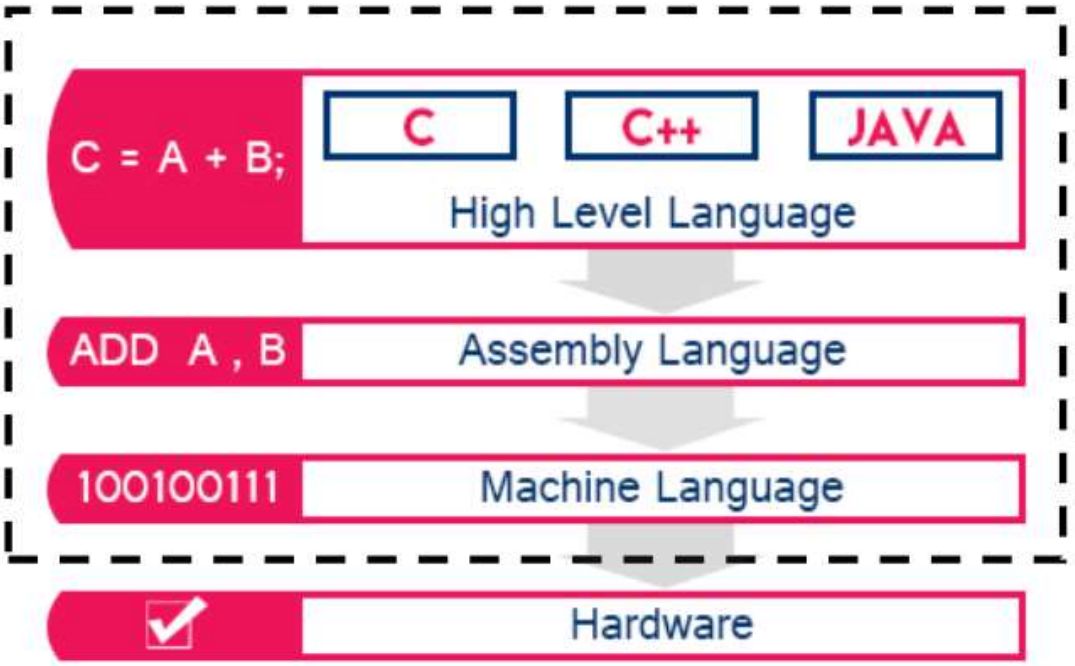
Exercises

1. Write a pseudocode that will calculate a running sum. A user will enter numbers that will be added to the sum and when a negative number is encountered, stop adding numbers and write out the final result.
2. Write a pseudocode of linear search algorithm.
3. Write a pseudocode of binary search algorithm.
4. Write a pseudocode to check whether a given array named ***A*** is sorted or not.
5. Write a pseudocode to find the sum and average of even numbers in a given array ***B***
6. Write pseudo code to print all multiples of 5 between 1 and 100

General Review

Programming languages

- Machine language (Low level language)
- Assembly language
- High level languages



Programming fundamentals

- Control structures.
  - **Selection structures** (if , if … else, if … else if …. else)
  - **Repetition** (iteration) structure (for , while, do … while)
- Functions
  - **Recursive functions**
- Arrays (one-dimensional , 2-dimensional)

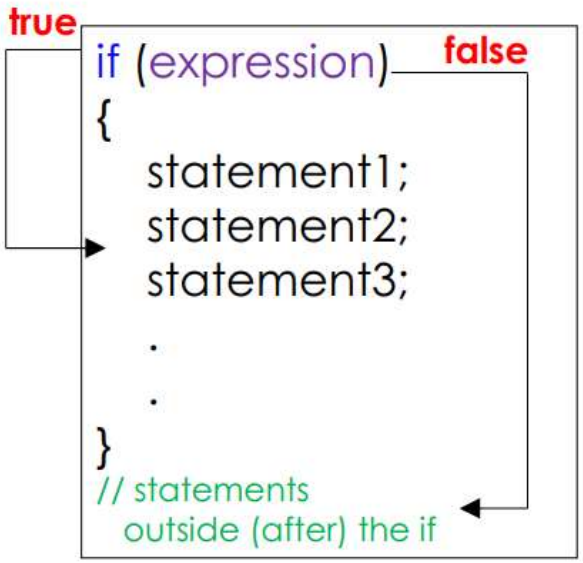
Selection Statements

Selection statements

- **Selection statements**: gives the ability to choose which set of statements (a block code) are executed according to a condition.
- There are different types of selection structures in C++ like:
  - if statement
  - if … else statement
  - if … else if … else statement (nested if statements)
  - Switch statement

if statement (cont.)

- **Compound statement (a number of statements):**



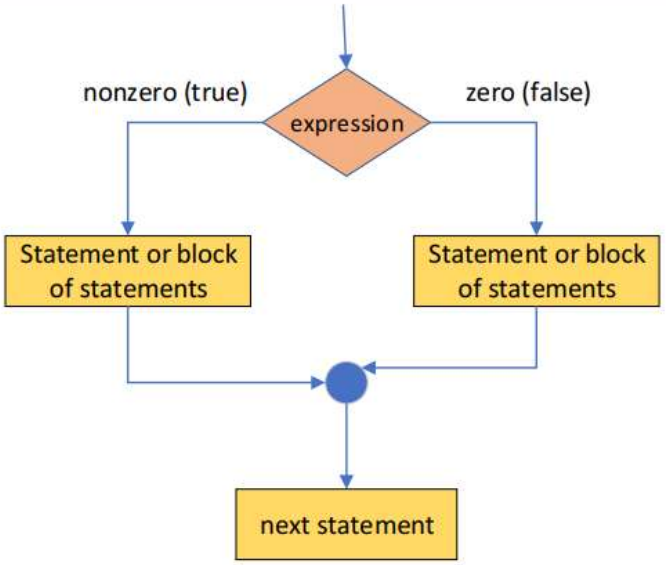
- If the *expression* evaluates *to* **true (nonzero)**, *statements* inside { } are executed;
- otherwise, *statements* inside { } are skipped and execution continues directly to the following statement

If … else Double-Selection Statement

- **if .. else**: Performs an action (or groups of actions) if the condition is **true** (nonzero), and a different action if the condition is **false (**so called double-selection**)**

- **General syntax:**

```
if ( expression )
{
        // block of code if expression is true
}
else
{
      // block of code if expression is false
{
// next statement
```



- **expression** is called statement **condition**

Extended-if Statement

- **if … else if … else (multiple selection statement)**
- **General form:**

```
if ( expression1 )
{
        // block of code-1
}
else if ( expression2 )
{
        // block of code-2
}
else if ( expression3 )
{
        // block of code-3
}
else
{
    // code that is executed of no condition is met
}
```

- The expressions are evaluated in the order of their appearance to determine the first expression that is **true**. The associated block of code is executed, and execution continues with the first statement following the entire **if-else-if** construct.
- If none of the expressions is true, the code associated with the **else** clause is executed, and execution then continues with the statement following the construct

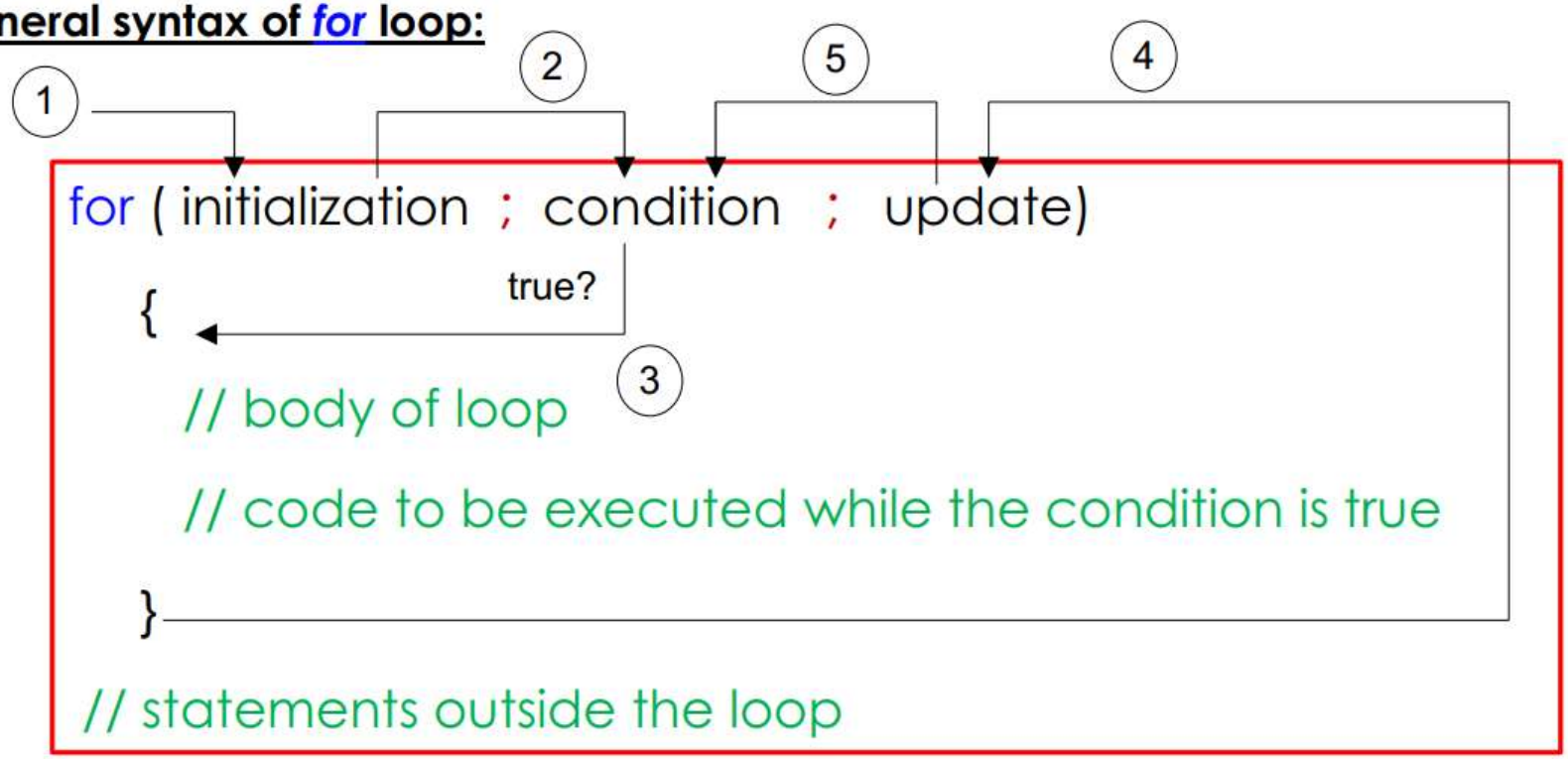
There are three types of loops:

1. ***for*** loop
2. ***while*** loop
3. ***do … while*** loop

Repetition structures

for loops

- **General syntax of *for* loop:**

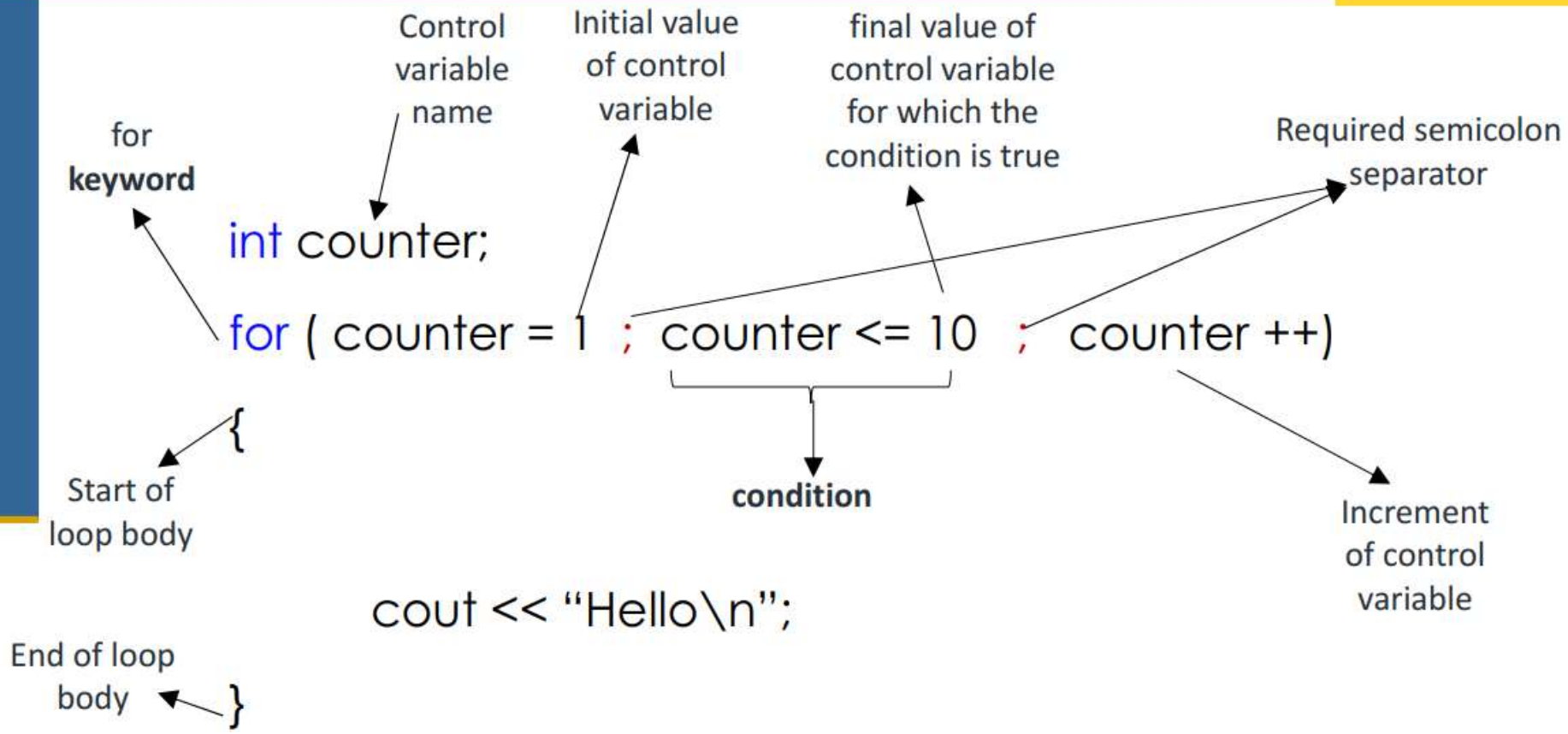


Essentials of counter-controlled loops

- Control variable (called **loop counter**)
  - is a variable used to control the loop.
  - It is given an **initial value**.
- Initialization: is usually used to give an initial value to the control variable. It is executed **executed only once** before the loop begins.
- The **update** portion is used to increment or decrement the control variable by a certain amount. (e.g. i++, i--, i+=2, i-=4)
- Condition: determines if looping should continue.

- if ***true***, the body of for loop is executed.
- if ***false***, the for loop is terminated

Essentials of counter-controlled loops



while loop

- **General syntax of *while* loop:**

```
Initialization;
while ( condition )
    {
       // body of loop
       // code to be executed while the condition is true
    }
 // statements outside the loop
```

Converting for to while

- **General syntax of *while* loop:**

```
for ( initialization ; condition ; update)
   {
     // body of loop
   }
 // statements outside the loop
```

```
initialization
while ( condition )
   {
     // body of loop
     update;
   }
 // statements outside the loop
```

do … while loop

- **General syntax of *do … while* loop:**

```
do {
   // body of loop
}
while ( condition );
```

- braces are not necessary when only one statement is being repeated.

What loop should you use??

- Any ***while*** loop can be converted into: **do-while** loop or **for** loop and vice versa.
- The loop to use depends on the problem"
- Should the loop always execute at least once?
  - Yes → **do-while** No → **while** or **for**
- Should the loop be **count controlled** or controlled by a **general condition**.
  - **Count** → **for** is the most common
  - **General condition** → **while** or **do-while** is most common
  - **Do-while** are **rarely** used.

User-defined functions

Create a function (Function Declaration)

- C++ allows the programmer to define their own function.
- **The general syntax to declare a function:**

```
return_type   function_name  ( parameter1, parameter2, …..)
{

   // function body
   Statements;

}
```

Function heading

Function body

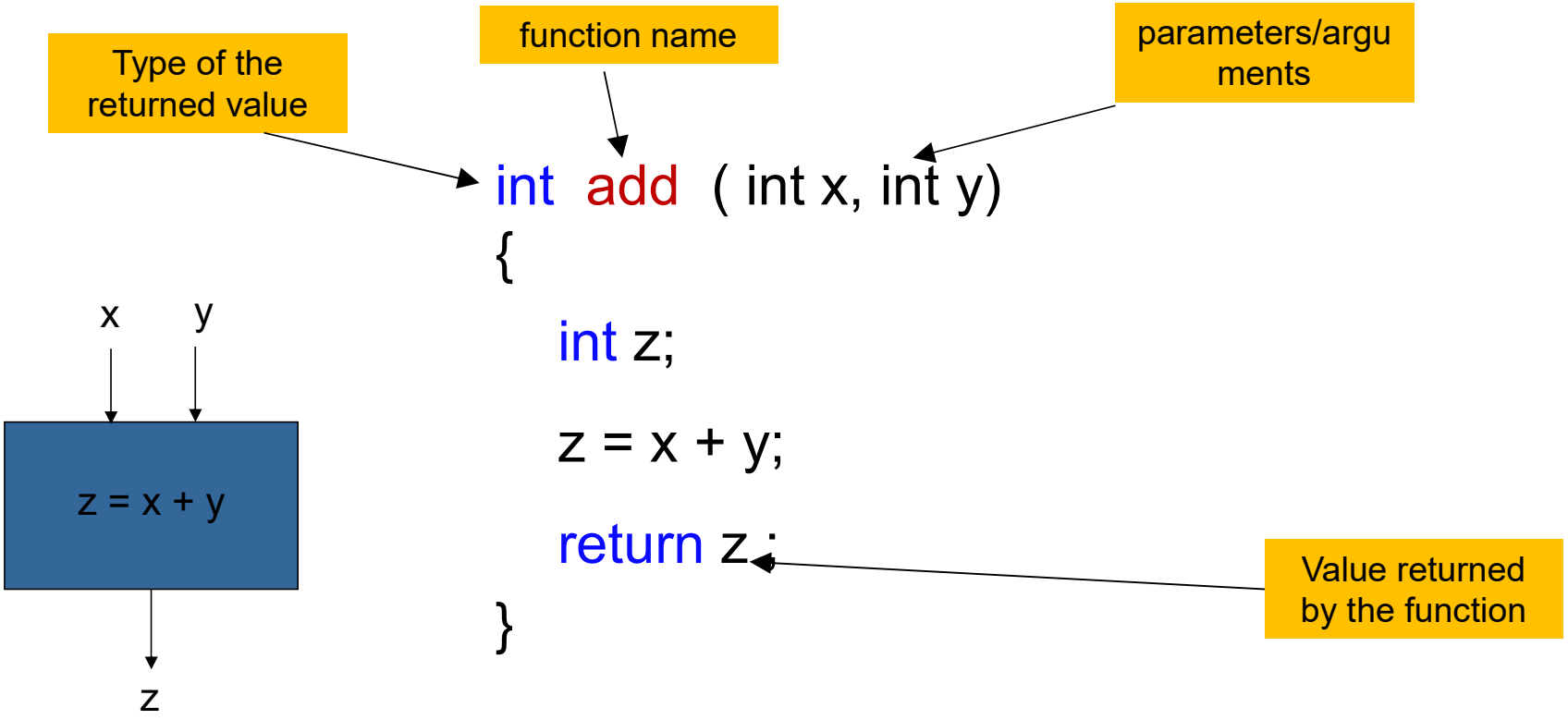
Create a function (Function Declaration)

- To declare a function, we need to specify:

1. Function **return type**.
   - Some functions return a value (int, double, float, char … etc).
   - Some function do not return a value (void)
2. Function **name**: should be valid (meet naming rules)
3. Function **parameters** (arguments): **input values** passed to the function.
4. Function **body**: The statements that construct the function.

Create a function (Function Declaration)

- **Declare/define a function to add two integer numbers and return the result.**



Return statement

- Some functions return a value and others do not.
  - **If the function return a result**, at least one statement will be a return statement indicating the function result.
  - **If the function does not return a result** then the return type is void. Optionally one or more return statements can exist.
- For functions that return a value, **it sends a single value back to the caller.**
- The type of the returned value must be compatible with the value in the function header.

- Return terminates the execution of statements in the function body.
- **Any statements after the return statement are not executed.**