# Programming and Data Structure

**Sujoy Ghose**

**Sudeshna Sarkar**

**Jayanta Mukhopadhyay**

**Dept. of Computer Science & Engineering.**

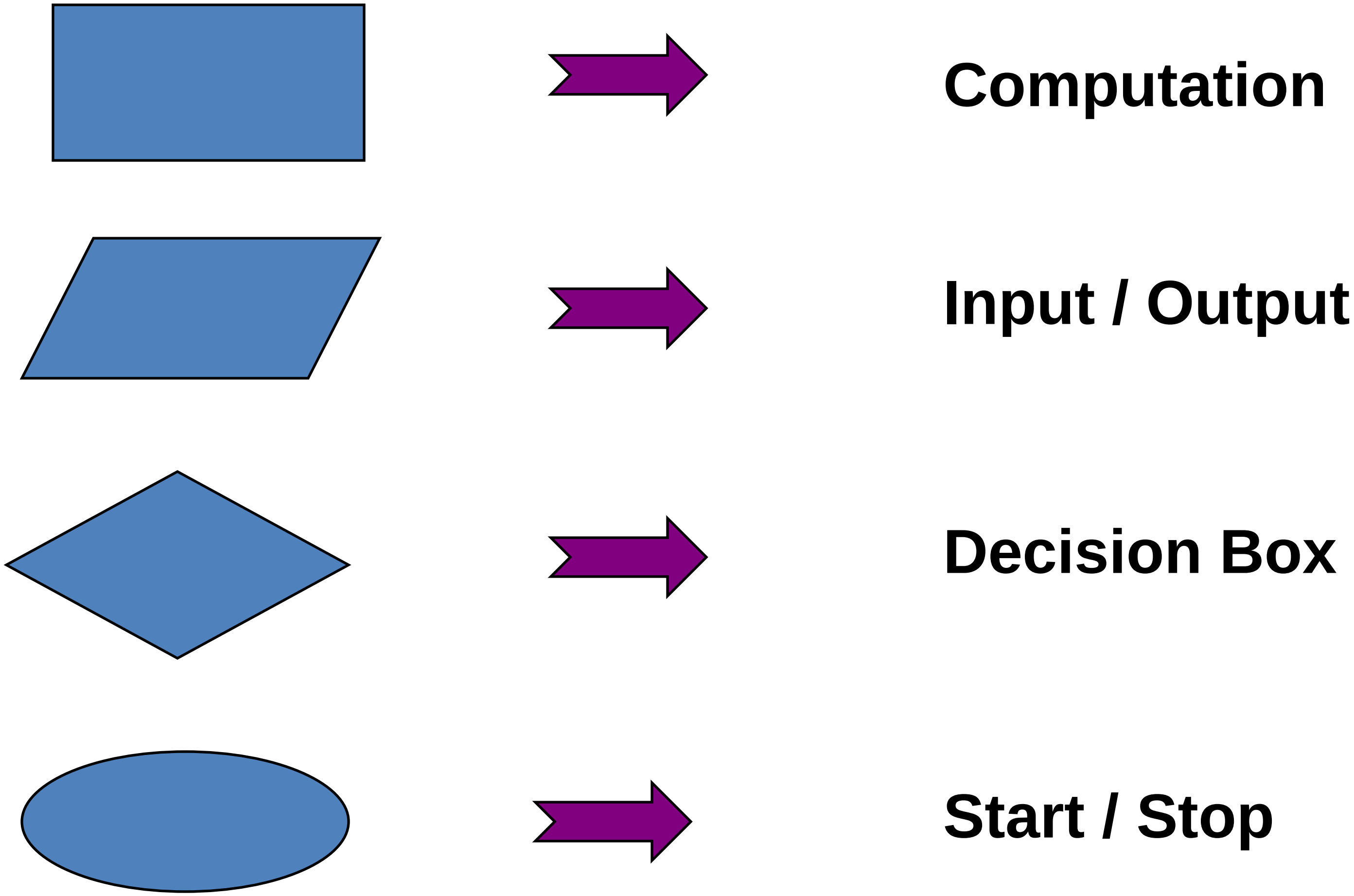
**Indian Institute of Technology**

**Kharagpur**

# Problem solving

- Step 1:
  - Clearly specify the problem to be solved.
- Step 2:
  - Draw flowchart / write algorithm
- Step 3:
  - Convert flowchart / algorithm into program code.
- Step 4:
  - Compile the program into object code.
- Step 5:
  - Execute the program.

# Flowchart: basic symbols



**Computation**

**Input / Output**

**Decision Box**

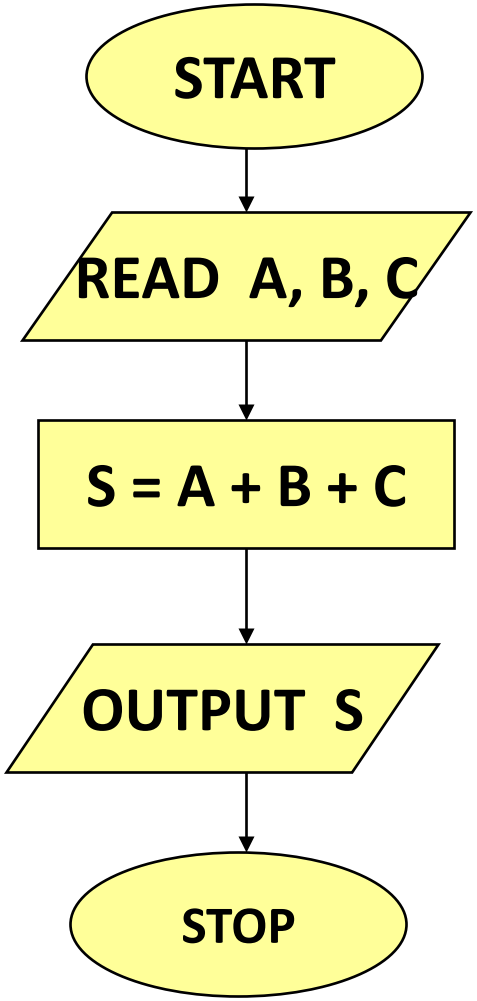
**Start / Stop**

# Contd.

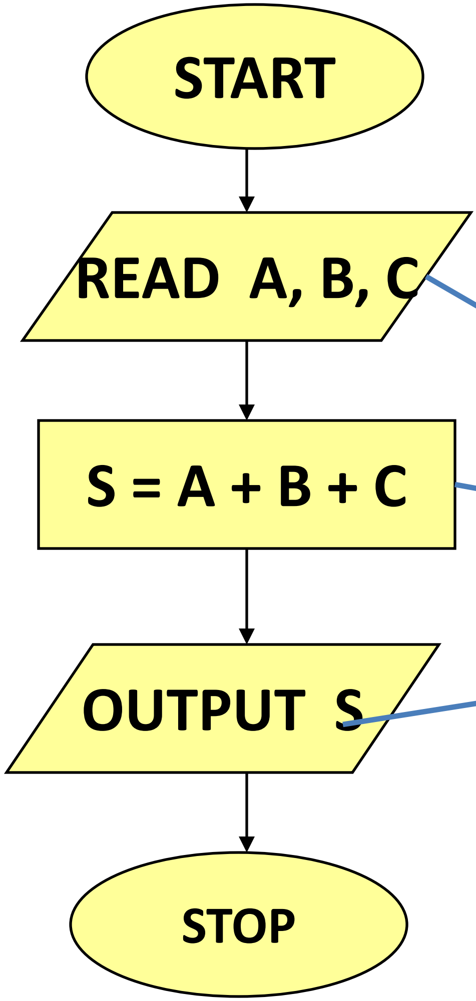
↓ ⇒ **Flow of control**

● ⇒ **Connector**

# Example 1: *Adding three numbers*



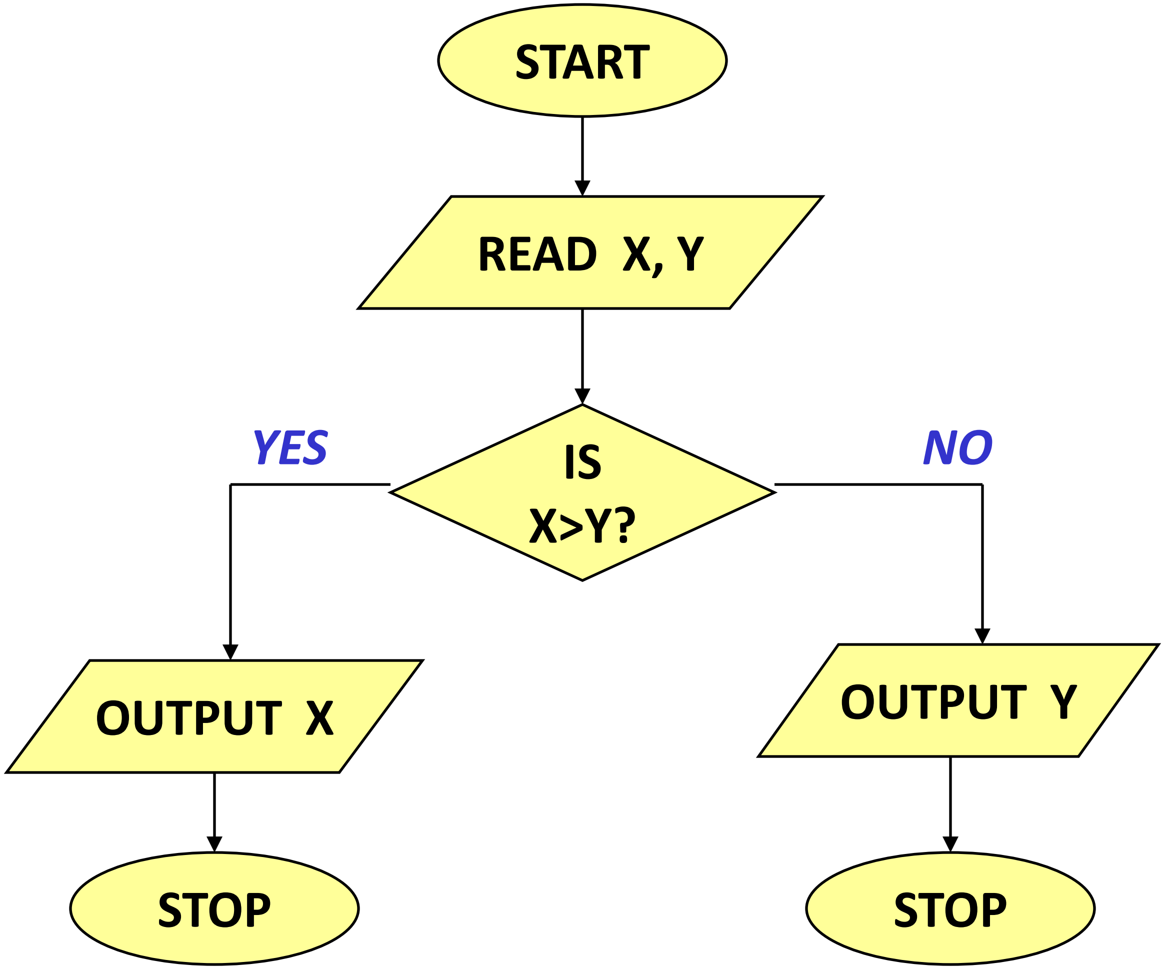
# Example 1: *Adding three numbers*



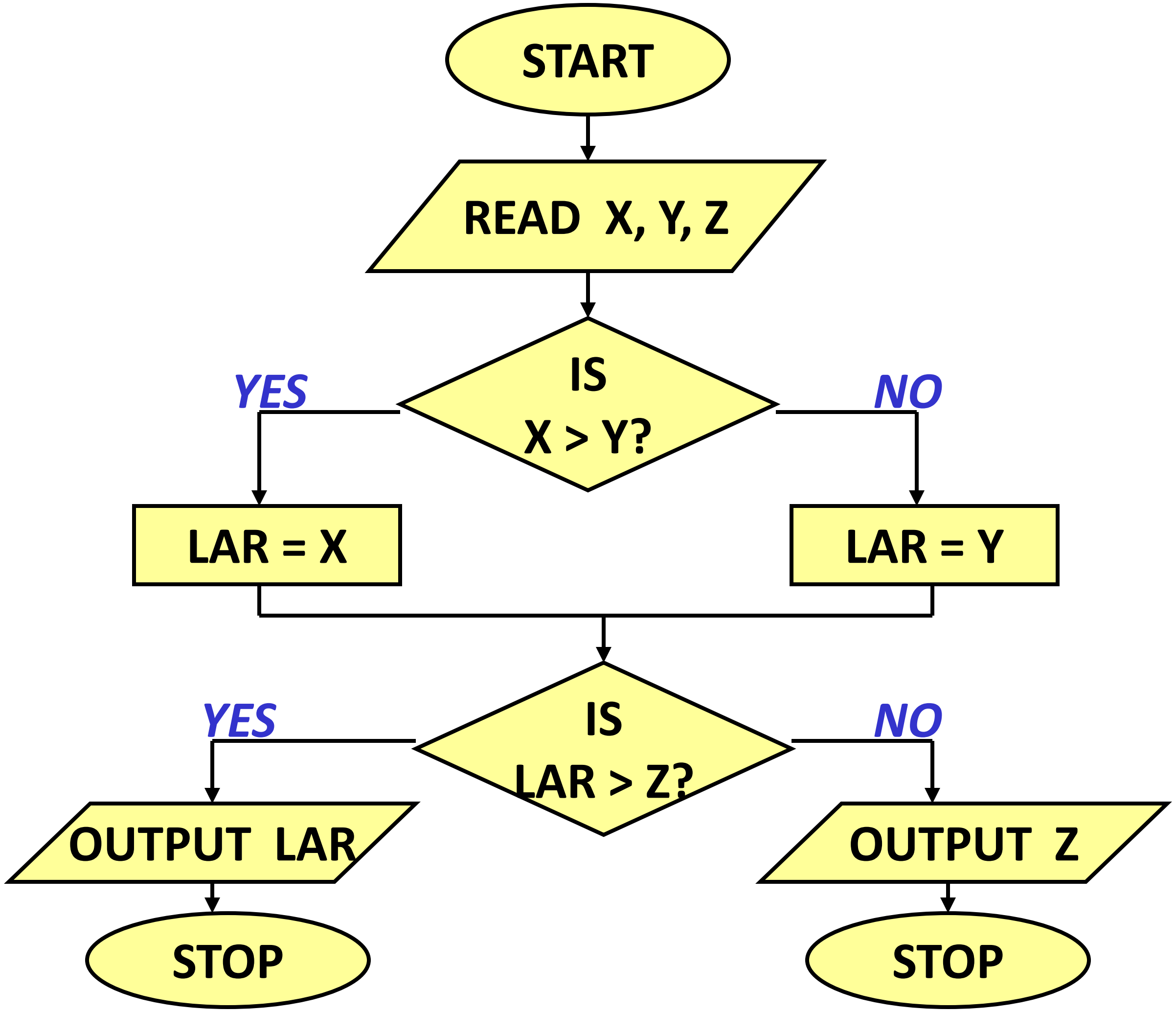
```
#include <stdio.h>
int main()
{
  int a, b, c, sum;

  scanf("%d%d%d",&a, &b, &c);

  sum = a + b + c;

  printf("%d",sum);

  return 0;
}
```

*Variable Declaration*

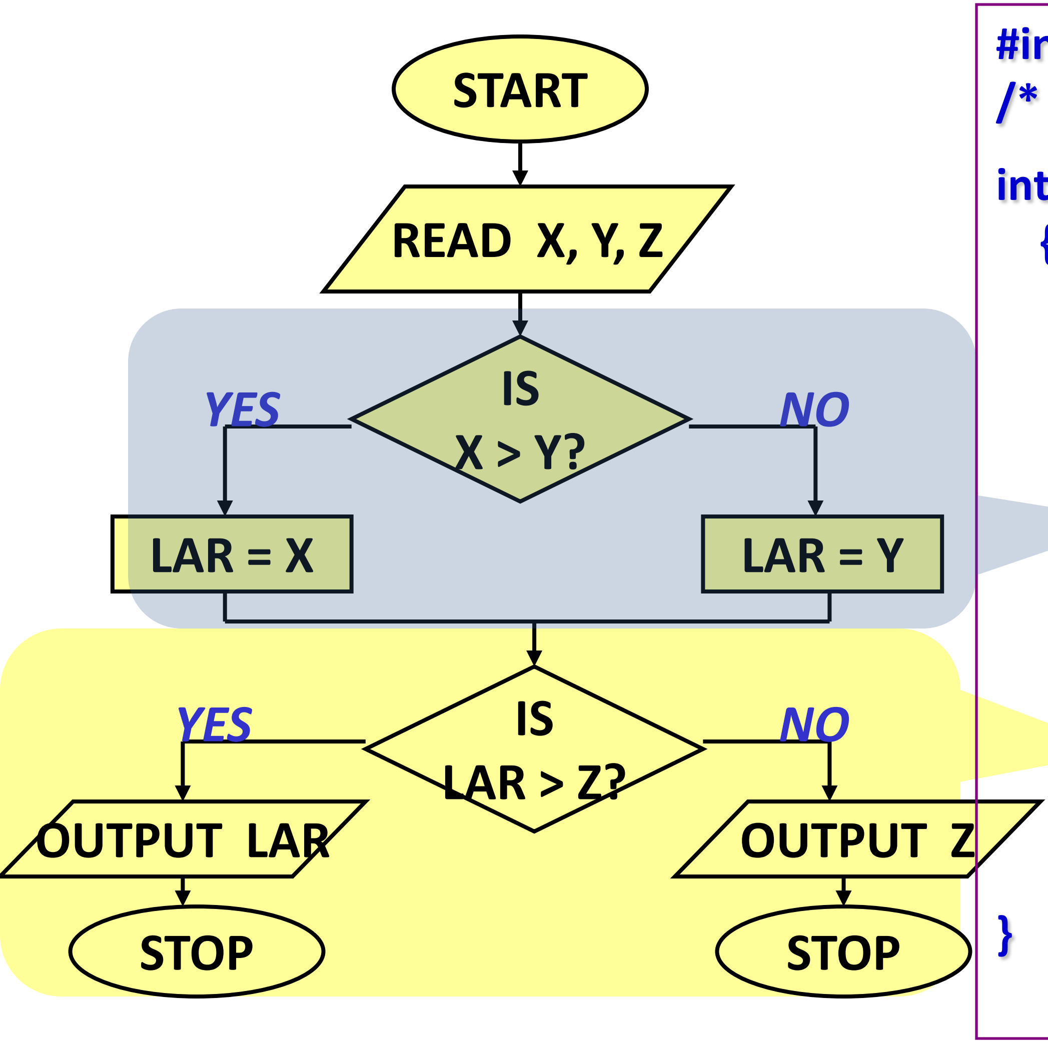
# Example 2: *Larger of two numbers*



# Example 3: *Largest of three numbers*

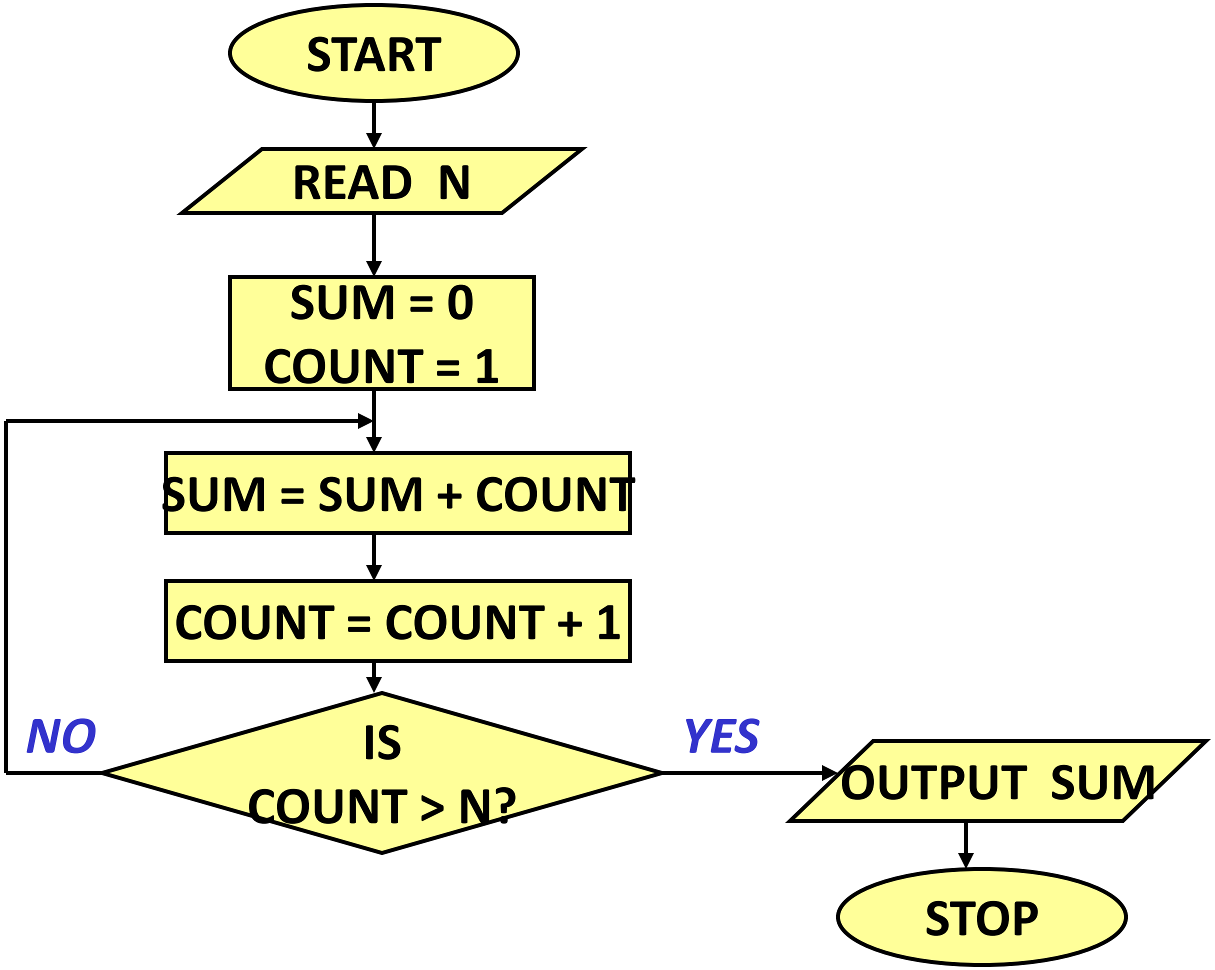


# Example 3: *Largest of three numbers*

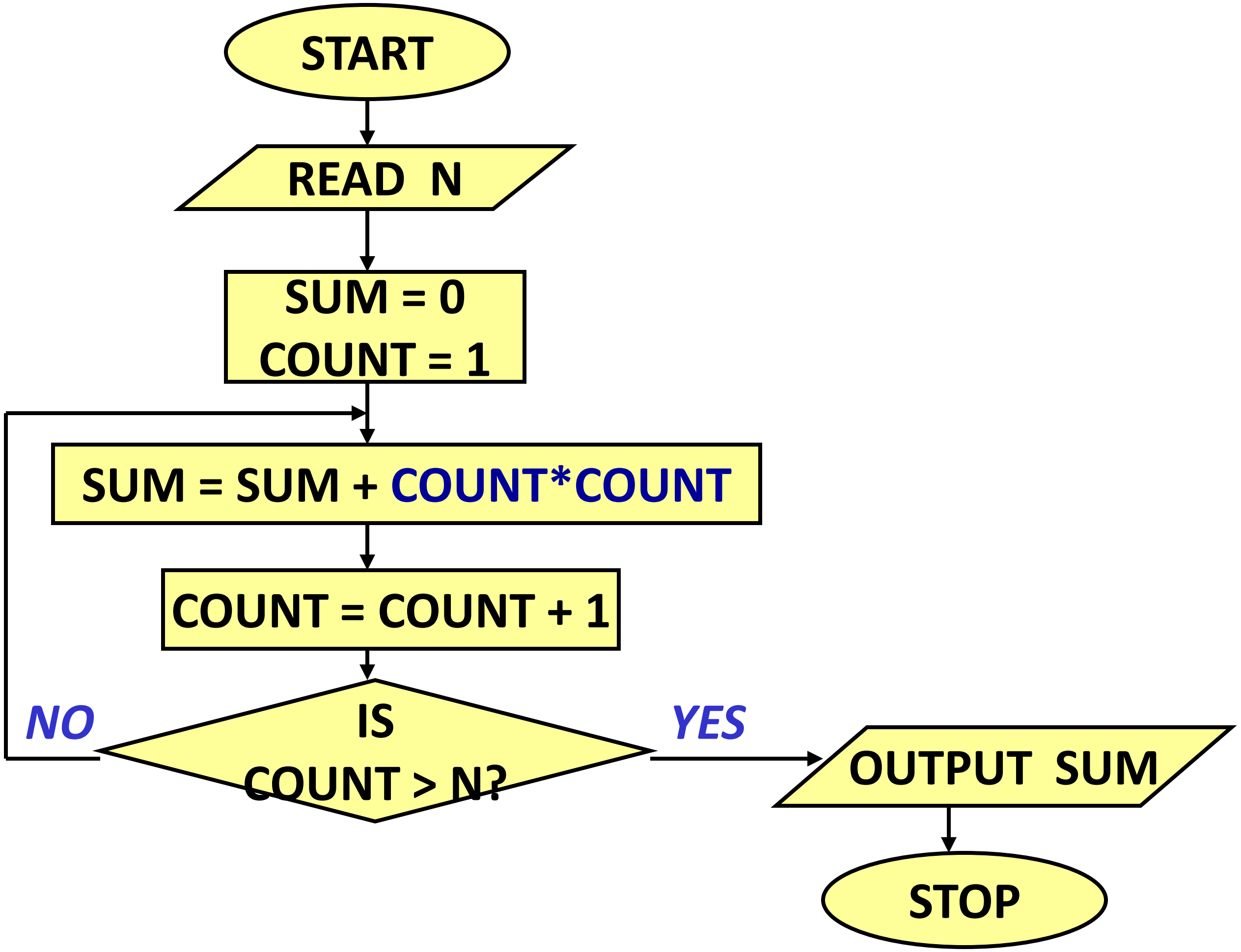


```
#include <stdio.h>
/* FIND THE LARGEST OF THREE NUMBERS */

int main()
   {
        int   a, b, c, lar;
        scanf (“%d %d %d”, &x, &y, &z);

        if  (x>y)
                lar = x;
        else lar = y;

        if (lar > z)
                printf(“Largest is %d”, lar);
        else printf(“Largest is %d”, z);
        return 0;
}
```

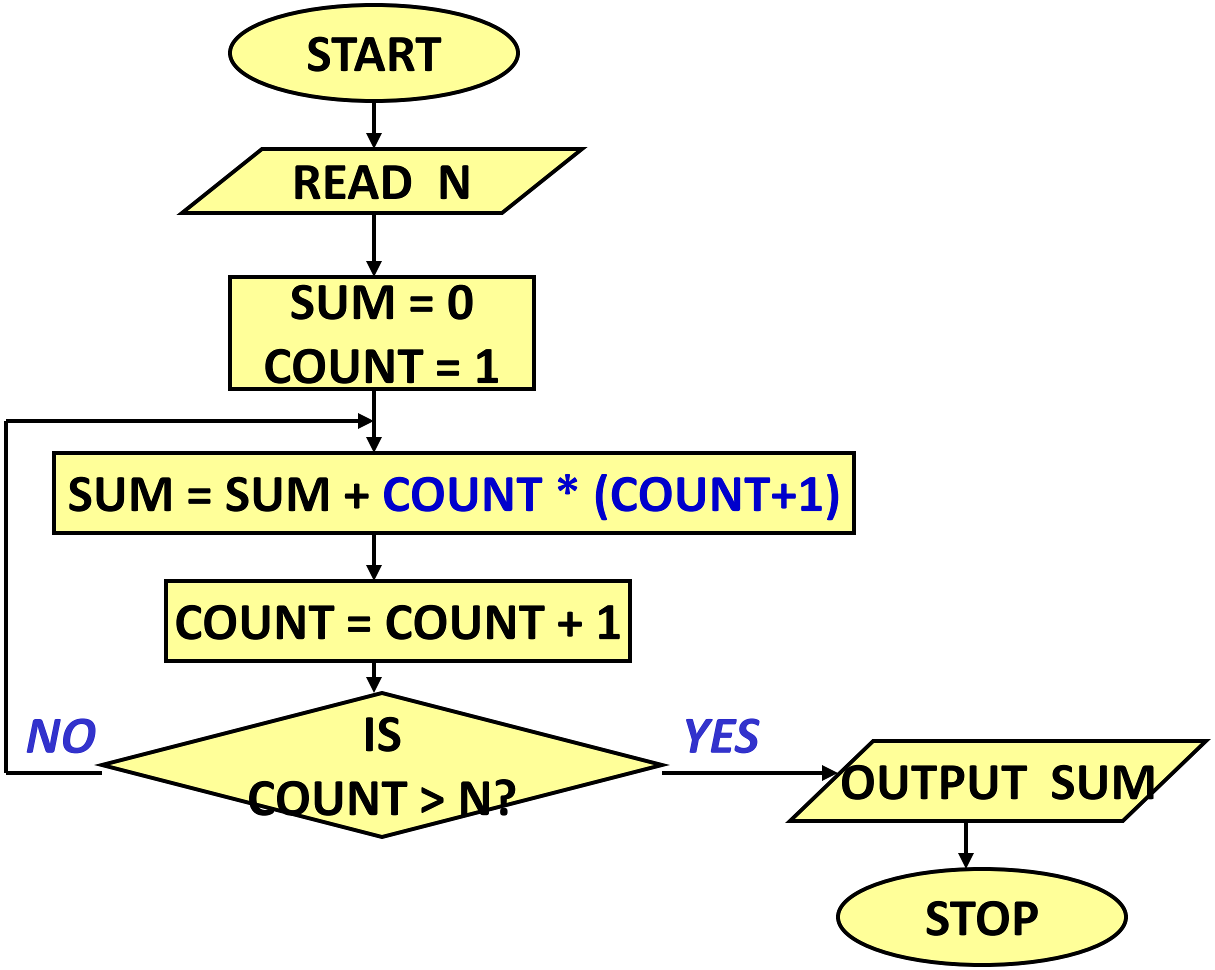
# Example 4: Sum of first N natural numbers



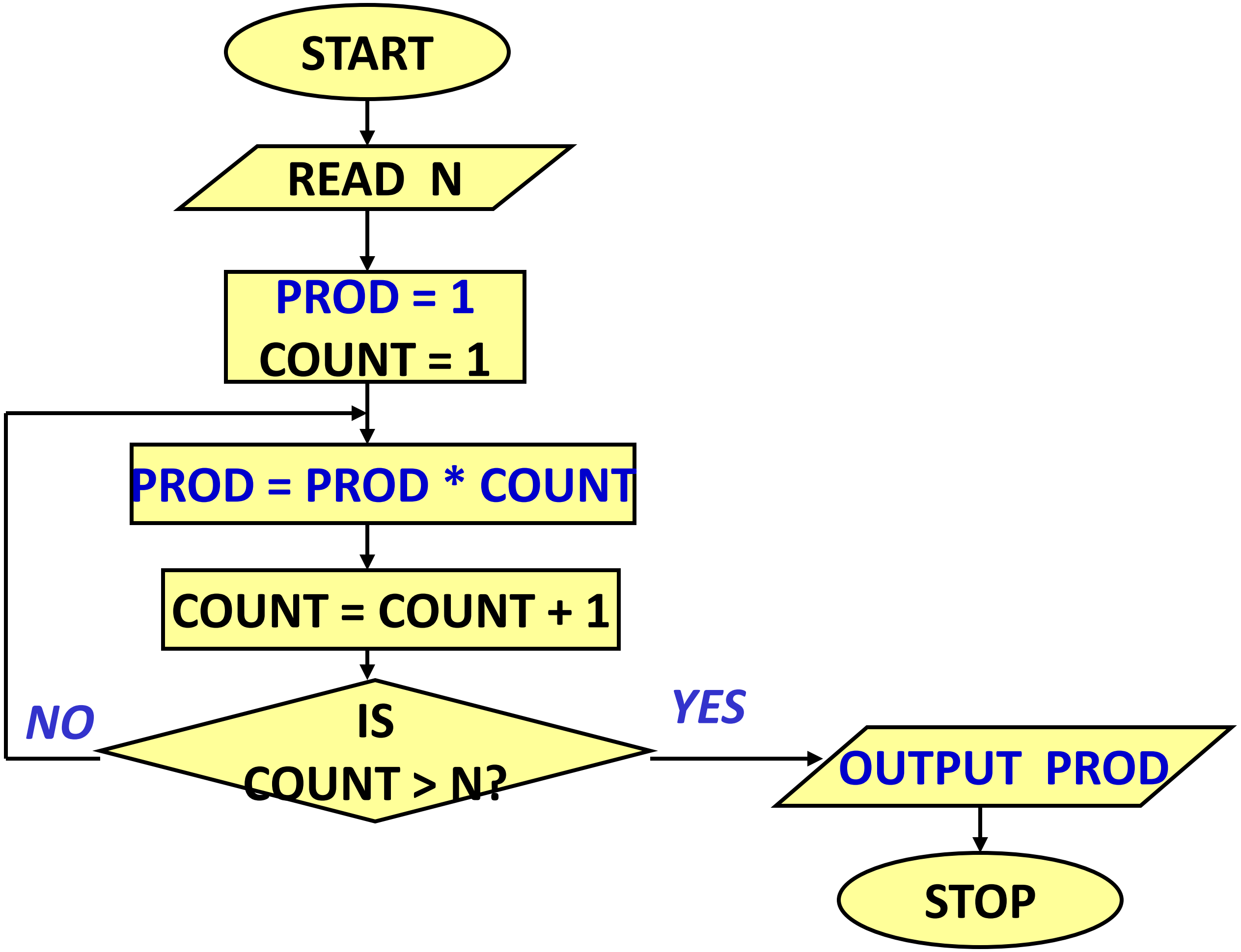
# Example 5: $SUM = 1^2 + 2^2 + 3^2 + N^2$



# Example 6: *SUM = 1.2 + 2.3 + 3.4 + to N terms*



# Example 7: *Computing Factorial*



START

READ N

PROD = 1

COUNT = 1

PROD = PROD * COUNT

COUNT = COUNT + 1

IS COUNT > N?

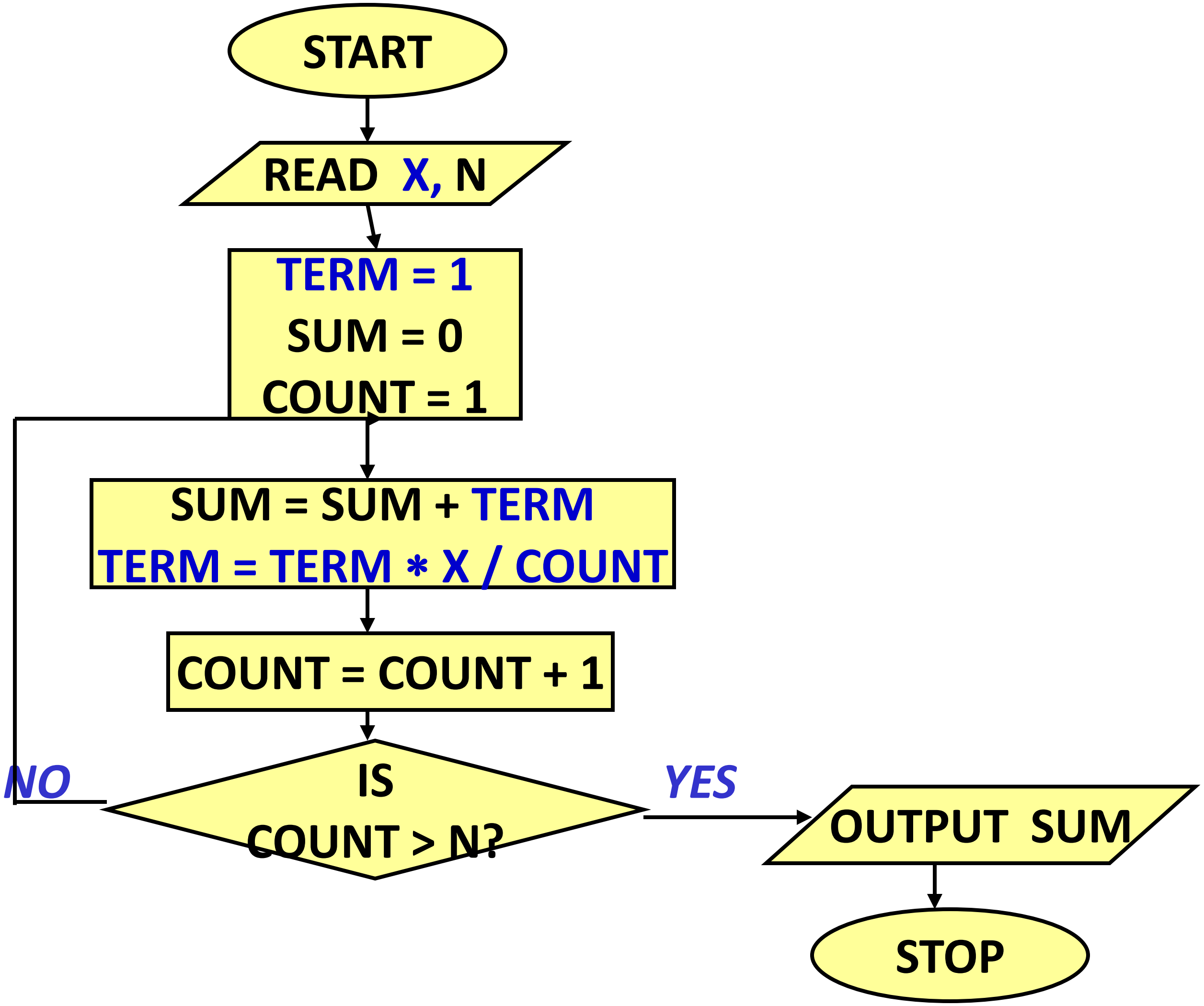
*NO*

*YES*

OUTPUT PROD

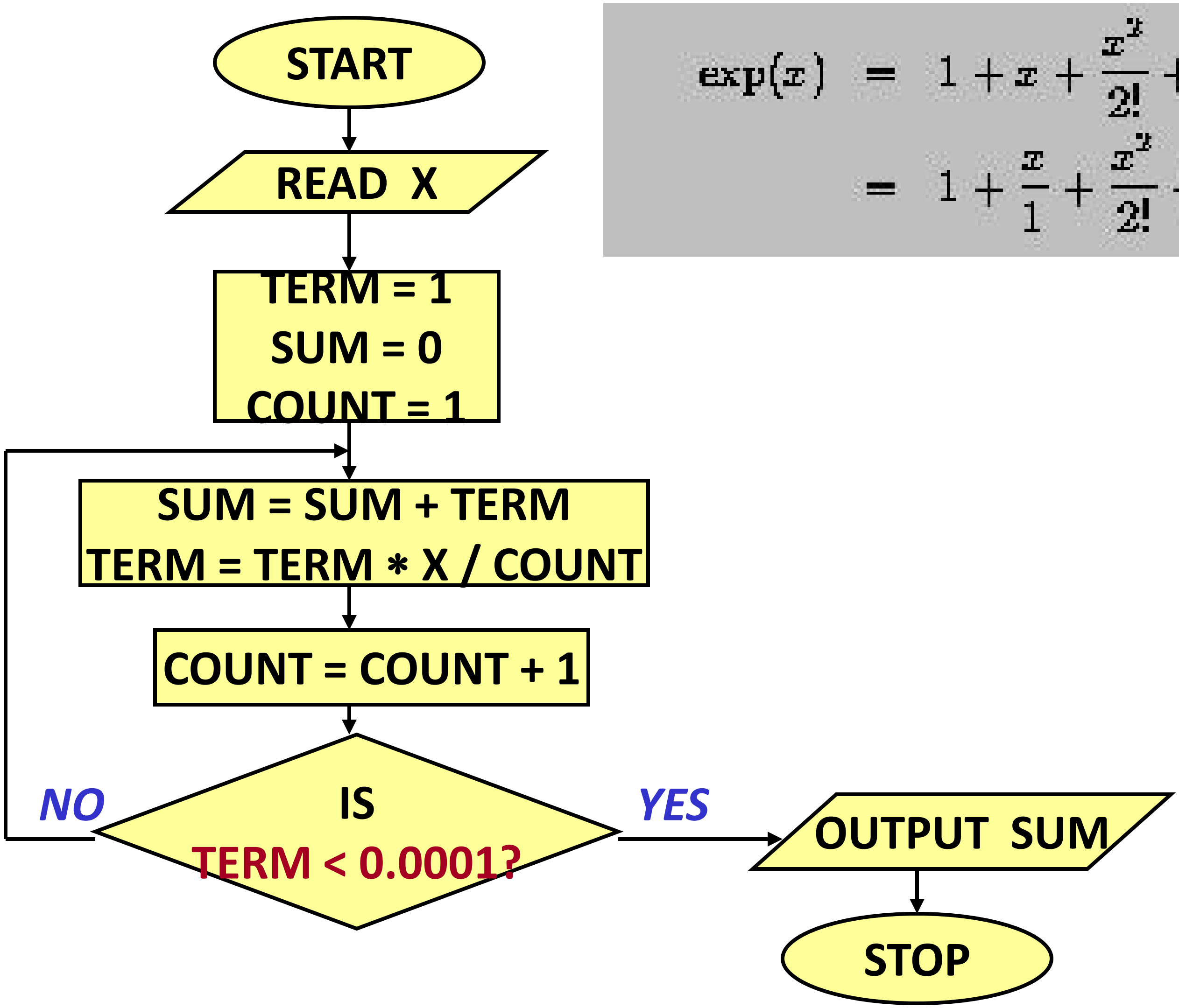
STOP

# Example 8: *Computing $e^x$ series up to N terms*



# Example 9: Computing e$^x$ series up to 4 decimal places

$$\begin{aligned}\exp(x) &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots \\ &= 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots\end{aligned}$$



START

READ X

TERM = 1

SUM = 0

COUNT = 1

SUM = SUM + TERM

TERM = TERM * X / COUNT

COUNT = COUNT + 1

IS TERM < 0.0001?

NO

YES

OUTPUT SUM

STOP

# Example 10: *Roots of a quadratic equation*

$$\mathbf{Ax^2 + Bx + C = 0}$$

# Example 11: *Grade computation*

MARKS ≥ 90 → Ex

89 ≥ MARKS ≥ 80 → A

79 ≥ MARKS ≥ 70 → B

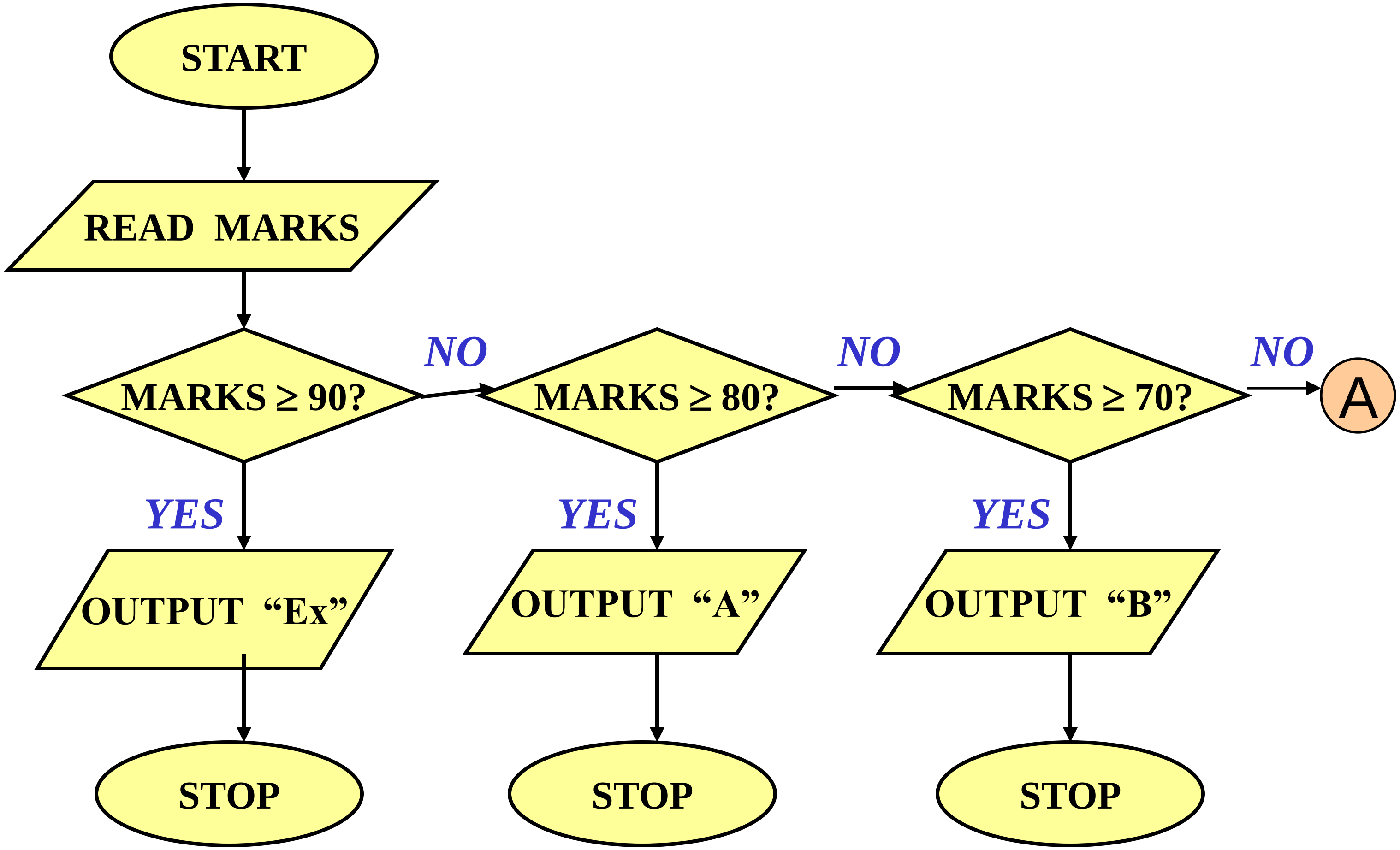
69 ≥ MARKS ≥ 60 → C

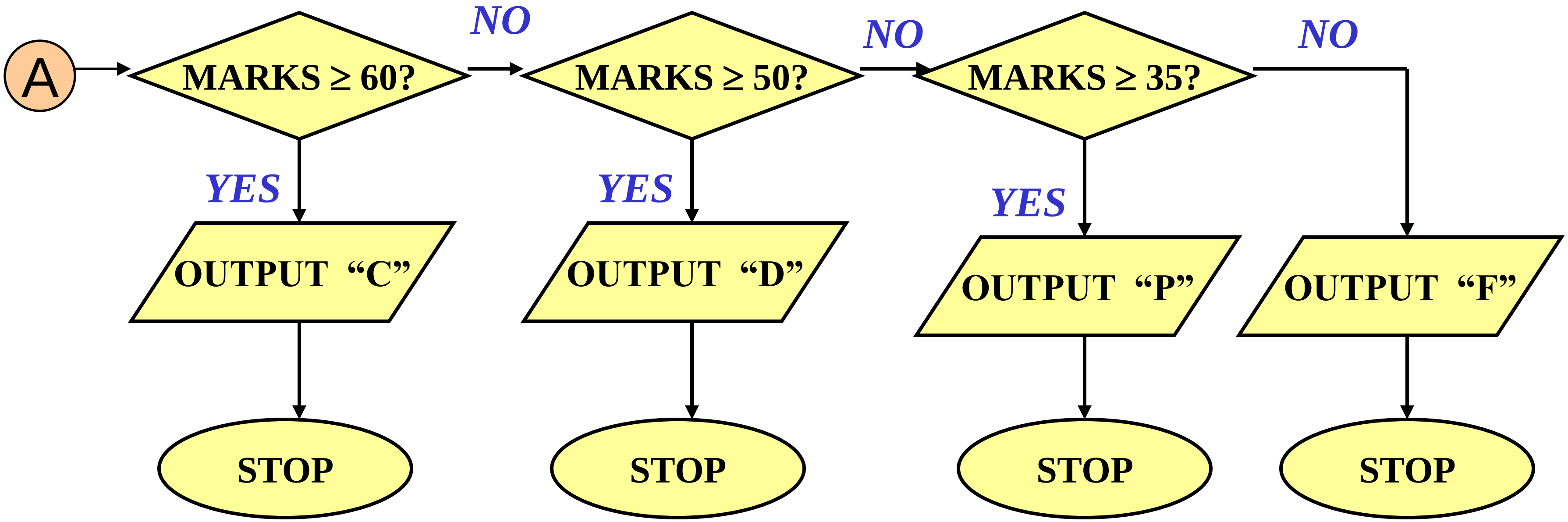
59 ≥ MARKS ≥ 50 → D

49 ≥ MARKS ≥ 35 → P

34 ≥ MARKS → F

# Grade Computation (contd.)




A
MARKS ≥ 60?
NO
MARKS ≥ 50?
NO
MARKS ≥ 35?
NO
YES
YES
YES
OUTPUT “C”
OUTPUT “D”
OUTPUT “P”
OUTPUT “F”
STOP
STOP
STOP
STOP

# Homework

## Design flowchart / algorithm

1. Take as input an integer **n** and a sequence of n numbers and prints their average.

2. Take a number as input and a base as input (both integers) and prints the digits of the number to the given base. For example, given 46 and 3, the output should be (from least significant to most significant) 1, 2, 0, 1.

# The C Character Set

- The C language alphabet:

  A,B,..Z     a, b, ..z

  0, 1,..9

  Certain special characters:

  ,  .  ;  %  \ | ~ # ? ( ) ‘ “ + ^ & * - _ + = [ ] { } < >

  blank tab newline

# Identifiers and Keywords

- Identifiers
  - Names given to various program elements (variables, constants, functions, etc.)
  - May consist of *letters*, *digits* and the *underscore* ('_') character, with no space between.
  - First character must be a letter.
  - An identifier can be arbitrary long.
    - Some C compilers recognize only the first few characters of the name (16 or 31).
  - Case sensitive
    - 'area', 'AREA' and 'Area' are all different.

# Contd.

- Keywords
  - Reserved words that have standard, predefined meanings in C.
  - Cannot be used as identifiers.
  - OK within comments.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **auto** | **break** | **case** | **char** | **const** | **continue** | **default** | **do** |
| **double** | **else** | **enum** | **extern** | **float** | **for** | **goto** | **if** |
| **int** | **long** | **register** | **return** | **short** | **signed** | **sizeof** | **static** |
| **struct** | **switch** | **typedef** | **union** | **unsigned** | **void** | **volatile** | **while** |

# Valid and Invalid Identifiers

X

abc

10abc

simple interest

simple_interest

Simple-interest

a123

LIST

stud_name

Employee_1

avg_empl_salary

“hello”

(area)

%rate

# Valid and Invalid Identifiers

√ X

√ abc

× ~~10abc~~

× ~~simple interest~~

√ simple_interest

× ~~Simple-interest~~

√ a123

√ LIST

√ stud_name

√ Employee_1

√ avg_empl_salary

× ~~“hello”~~

× ~~(area)~~

× ~~%rate~~

# Data Types in C

**int** :: integer quantity

Typically occupies 4 bytes (32 bits) in memory.

**char** :: single character

Typically occupies 1 byte (8 bits) in memory.

**float** :: floating-point number (a number with a decimal point)

Typically occupies 4 bytes (32 bits) in memory.

**double** :: double-precision floating-point number

# Contd.

- Some of the basic data types can be augmented by using certain data type qualifiers:
  - short
  - long
  - signed
  - unsigned
- Typical examples:
  - short int
  - long int
  - unsigned int

# Some Examples of Data Types

- **int**

  0, 25, -156, 12345, –99820

- **char**

  ‘a’, ‘A’, ‘*’, ‘/’, ‘ ’

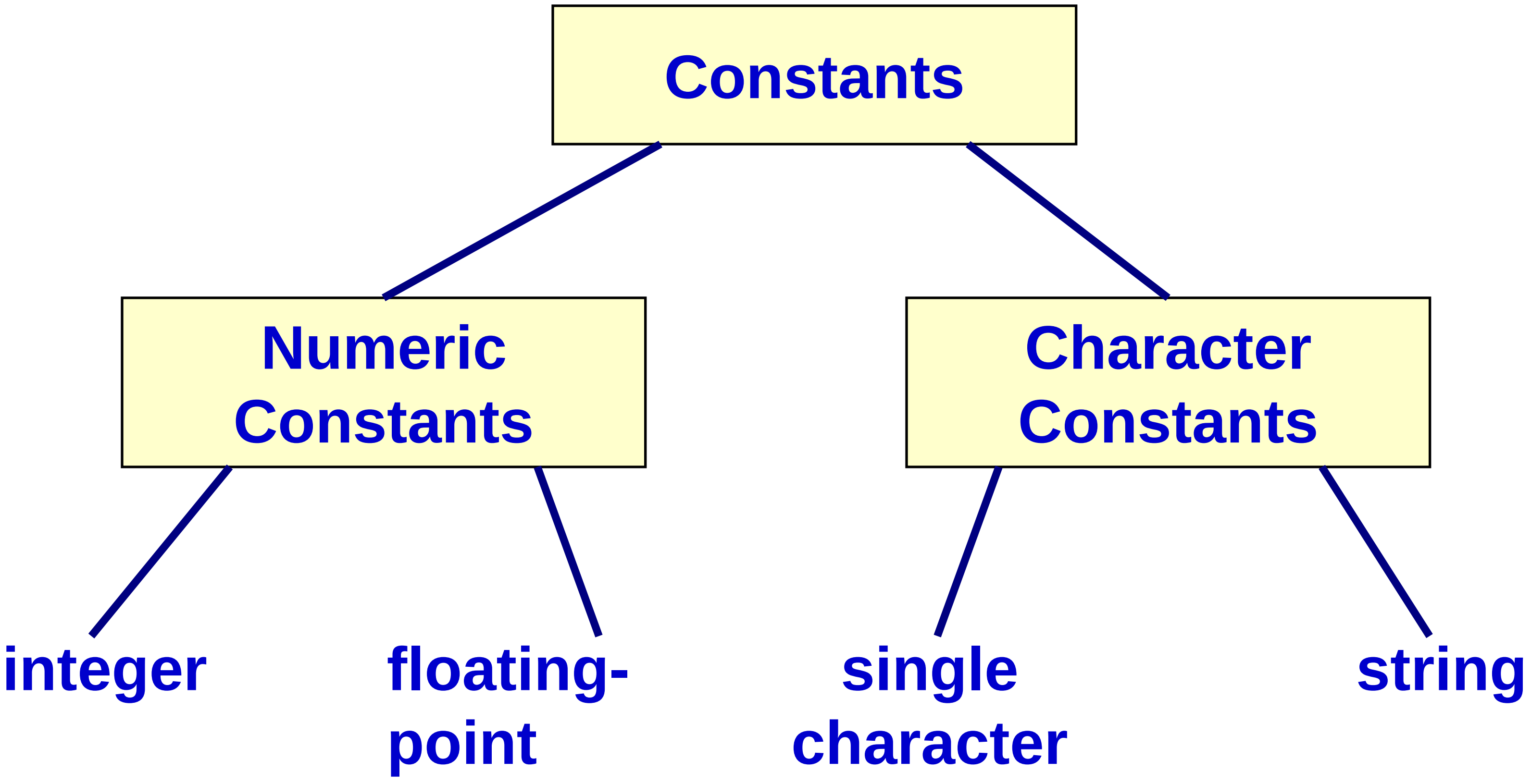
- **float**

  23.54, –0.00345, 25.0

  2.5E12, 1.234e-5

**E or e means “10 to the power of”**

# Constants


Constants
Numeric Constants
Character Constants
integer
floating-point
single character
string

# Integer Constants

- Consists of a sequence of digits, with possibly a plus or a minus sign before it.
  - – Embedded spaces, commas and non-digit characters are not permitted between digits.
- Maximum and minimum values (for 32-bit representations)

| | | |
|---|---|---|
| Maximum :: | 2 147 483 647 | (2^31 – 1) |
| Minimum :: | – 2 147 483 648 | (– 2^31) |

# Floating-point Constants

- Can contain fractional parts.
- Very large or very small numbers can be represented.

  23000000 can be represented as 2.3e7

- Two different notations:
  1. Decimal notation

     25.0, 0.0034, .84, -2.234

  2. Exponential (scientific) notation

     3.45e23, 0.123e-12, 123E2

**E or e means "10 to the power of"**

# Single Character Constants

- Contains a single character enclosed within a pair of single quote marks.
  - Examples :: ‘2’, ‘+’, ‘Z’
- Some special backslash characters

| | |
|---|---|
| ‘\n’ | new line |
| ‘\t’ | horizontal tab |
| ‘\’’ | single quote |
| ‘\”’ | double quote |
| ‘\\’ | backslash |
| ‘\0’ | null |

# String Constants

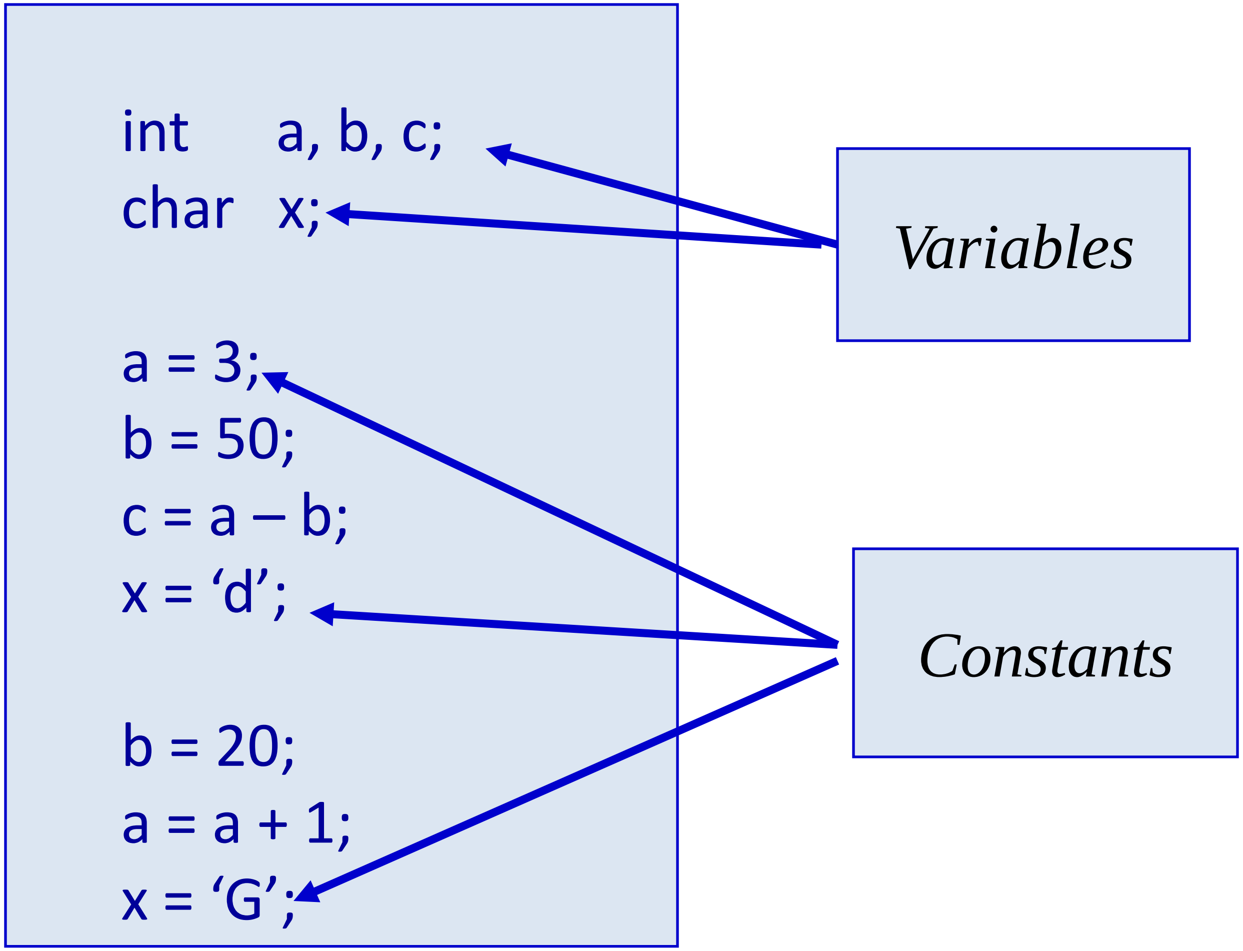
- Sequence of characters enclosed in double quotes.
  - The characters may be letters, numbers, special characters and blank spaces.
- Examples:

  “nice”, “Good Morning”, “3+6”, “3”, “C”
- Differences from character constants:
  - ‘C’ and “C” are not equivalent.
  - ‘C’ has an equivalent integer value while “C” does not.

# Variables

- It is a data name that can be used to store a data value.
- Unlike constants, a variable may take different values in memory during execution.
- Variable names follow the naming convention for identifiers.
  – Examples :: temp, speed, name2, current

# Example



```
int     a, b, c;
char    x;

a = 3;
b = 50;
c = a – b;
x = ‘d’;

b = 20;
a = a + 1;
x = ‘G’;
```

*Variables*

*Constants*

# Declaration of Variables

- There are two purposes:
  1. It tells the compiler what the variable name is.
  2. It specifies what type of data the variable will hold.
- General syntax:

```
data-type  variable-list;
```

- Examples:

```
int   velocity, distance;
int   a, b, c, d;
float  temp;
char  flag, option;
```

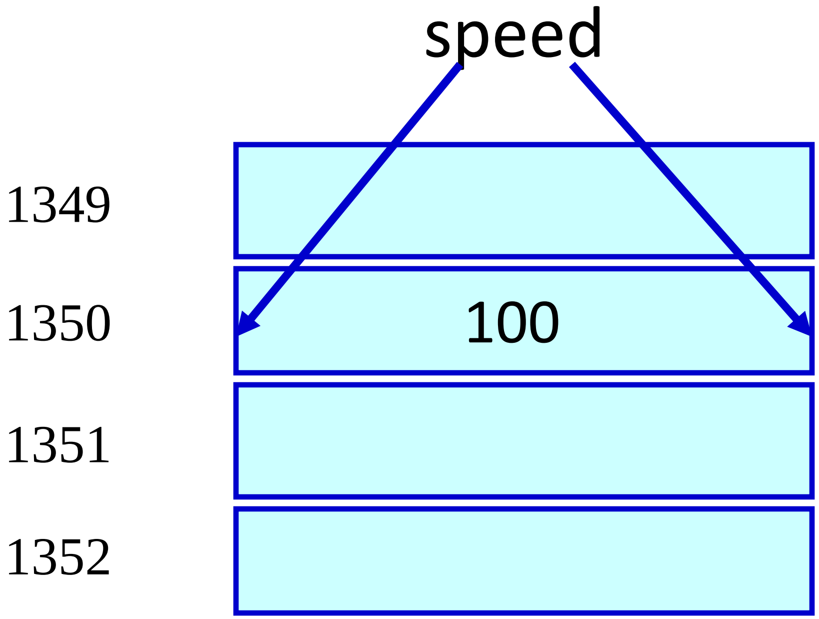
# A First Look at Pointers

- A variable is assigned a specific memory location.
  - For example, a variable speed is assigned memory location 1350.
  - Also assume that the memory location 1350 contains the data value 100.
  - When we use the name speed in an expression, it refers to the value 100 stored in the memory location.

```
distance = speed * time;
```

- Thus every variable has an *address* (in memory), and its *contents*.

# Adress and Content



```
int speed;
speed=100;
```

**speed** → 100

**&speed** → 1350

# Contd.

- In C terminology, in an expression

  speed refers to the contents of the memory location.

  &speed refers to the address of the memory location.

- Examples:

  ```
  printf ("%f %f %f", speed, time, distance);
  scanf ("%f %f", &speed, &time);
  ```

# An Example

```
#include <stdio.h>
int main()
{
        float  speed, time, distance;

        scanf (“%f %f”, &speed, &time);
        distance = speed * time;
        printf (“\n The distance traversed is: \n”, distance);
        return 0;
}
```

Address of speed

Content of speed

# Assignment Statement

- Used to assign values to variables, using the assignment operator (=).
- General syntax:

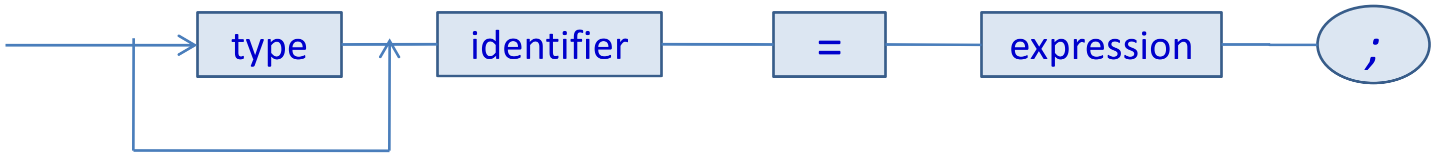
**variable_name = expression;**

**type variable_name = expression;**

- Examples:

```
int velocity = 20;
b = 15;  temp = 12.5;
A = A + 10;
v = u + f ∗ t;
s = u ∗ t + 0.5 ∗ f ∗ t ∗ t;
```

# **lvalue** and assignment operator



- Requires an lvalue as its left operand.
- l-value: represents an object stored in memory, which is neither a constant nor a result of computation.
- So a variable can be an lvalue, but neither any expressions nor any constant.

```
12 = i ;         // WRONG
i + j = 0 ;      // WRONG
–i = j ;          // WRONG
i++  = j ;       // WRONG
X+10 = Y*2;  // WRONG
```

# Assigning values to variables

- Lhs = rhs
- Lhs is a lvalue. Can be a variable name. Later we will consider arrays, pointers etc.
- Rhs is an expression compatible with the type of the lhs

```
 centigrade = 5*(fahrenheit – 32)/9;
```

- Assignment statement has value = rhs
- A value can be assigned to a variable at the time the variable is declared.

```
int   speed = 30;
char  flag = ‘y’;
```

# Contd.

- Several variables can be assigned the same value using multiple assignment operators.

```
a = b = c = 5;
flag1 = flag2 = ‘y’;
speed = flow = 0.0;
```

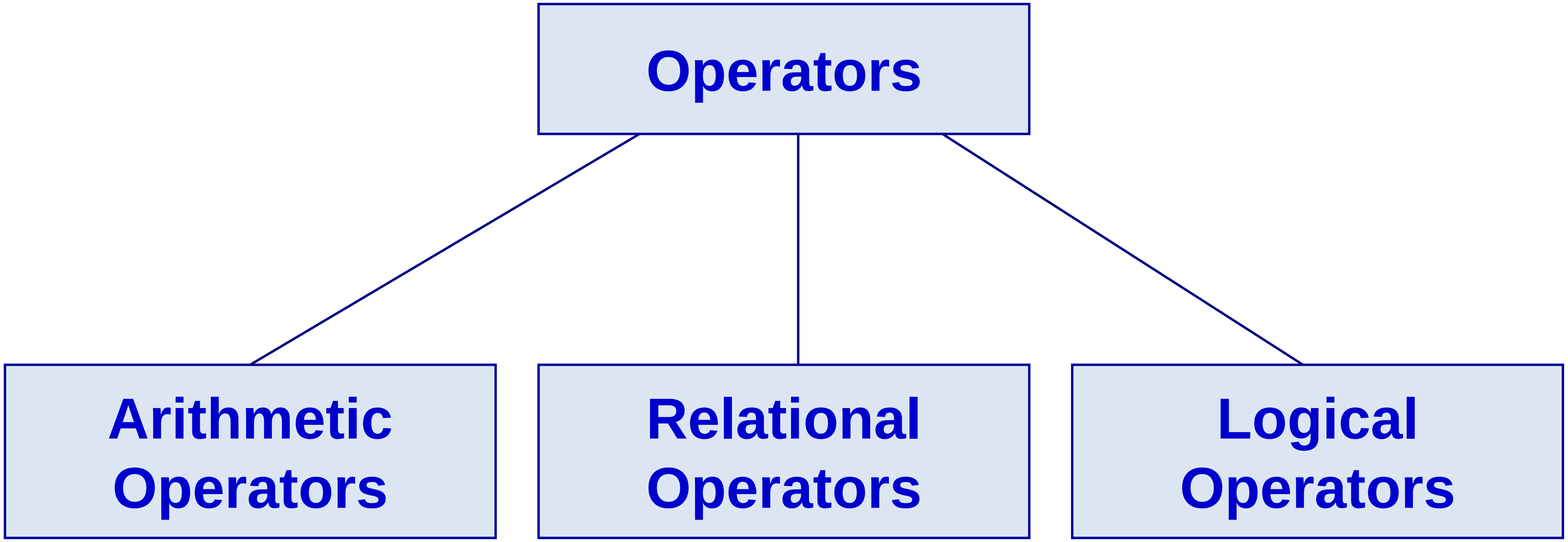
# Example: swapping two numbers

```
float x = 5, y = 11;
float  temporary ;
temporary = x;
x = y;
y = temporary;
```

*Can you swap without using a temporary variable?*

| x | y | temporary |
|---|---|---|
| 5 | 11 | |
| 5 | 11 | 5 |
| 11 | 11 | 5 |
| 11 | 5 | 5 |

# Operators in Expressions



Operators

Arithmetic Operators

Relational Operators

Logical Operators

# Arithmetic Operators

- Addition :: +
- Subtraction :: –
- Division :: /
- Multiplication :: *
- Modulus :: %

# Examples

```
distance = rate ∗ time ;
netIncome = income – tax ;
speed = distance / time ;
area = PI ∗ radius  ∗ radius;
y = a ∗ x ∗ x + b ∗ x + c;
quotient = dividend / divisor;
remain =dividend % divisor;
```

# Contd.

- Suppose x and y are two integer variables, whose values are 13 and 5 respectively.

| | |
|---|---|
| x + y | 18 |
| x – y | 8 |
| x * y | 65 |
| x / y | 2 |
| x % y | 3 |

# Operator Precedence

- In decreasing order of priority
  1. Parentheses :: ( )
  2. Unary minus :: –5
  3. Multiplication, Division, and Modulus
  4. Addition and Subtraction
- For operators of the *same priority*, evaluation is from *left to right* as they appear.
- Parenthesis may be used to change the precedence of operator evaluation.

# Examples: Arithmetic expressions

$$a + b \ast c - d / e \equiv a + (b \ast c) - (d / e)$$

$$a \ast - b + d \% e - f \equiv a \ast (- b) + (d \% e) - f$$

$$a - b + c + d \equiv (((a - b) + c) + d)$$

$$x \ast y \ast z \equiv ((x \ast y) \ast z)$$

$$a + b + c \ast d \ast e \equiv (a + b) + ((c \ast d) \ast e)$$

# Integer Arithmetic

- When the operands in an arithmetic expression are integers, the expression is called *integer expression*, and the operation is called *integer arithmetic*.
- Integer arithmetic always yields integer values.

# Real Arithmetic

- Arithmetic operations involving only real or floating-point operands.
- Since floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the final result.

  1.0 / 3.0 * 3.0 will have the value 0.99999 and not 1.0
- The modulus operator cannot be used with real operands.

# Mixed-mode Arithmetic

- When one of the operands is integer and the other is real, the expression is called a *mixed-mode* arithmetic expression.
- If either operand is of the real type, then only real arithmetic is performed, and the result is a real number.

  25 / 10 ➔ 2

  25 / 10.0 ➔ 2.5
- Some more issues will be considered later.

- Mixing types may result in precision loss, overflow, underflow and ability to process full range.

# Problem of value assignment

- Assignment operation

  variable= expression_value;

  or

  variable1 = variable2;

Data type of the RHS should be compatible with that of LHS.

If a floating point number is assigned to an integer variable, there will be truncation, may lead to loss.

# Type Casting

```
int a=10, b=4, c;
float x, y;

c = a / b;
x = a / b;
y = (float) a / b;
```

The value of c will be 2

The value of x will be 2.0

The value of y will be 2.5

# Type Casting

```
int x;
float r=3.0;

x= (int)(2*r);
```

Type casting of a floating point expression to an integer variable.

```
double perimeter;
float pi=3.14;
int r=3;

perimeter=2.0* (double) pi * (double) r;
```

Type casting to double

# Relational Operators

- Used to compare two quantities.

| | |
|---|---|
| **<** | **is less than** |
| **>** | **is greater than** |
| **<=** | **is less than or equal to** |
| **>=** | **is greater than or equal to** |
| **==** | **is equal to** |
| **!=** | **is not equal to** |

# Examples

| | |
|---|---|
| 10 > 20 | is false |
| 25 < 35.5 | is true |
| 12 > (7 + 5) | is false |

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared.

  a + b > c – d  is the same as  (a+b) > (c+d)

# Examples

- Sample code segment in C

```
if  (x > y)
   printf (“%d is larger\n”, x);
else
  printf (“%d is larger\n”, y);
```

# Logical Operators

- There are two logical operators in C (also called logical connectives).

  && ➔ Logical AND

  || ➔ Logical OR

  – They act upon operands that are themselves logical expressions.
  – The individual logical expressions get combined into more complex conditions that are true or false.

- Logical AND
  - Result is true if both the operands are true.
- Logical OR
  - Result is true if at least one of the operands are true.

| X | Y | X && Y | X \|\| Y |
|---|---|---|---|
| FALSE | FALSE | FALSE | FALSE |
| FALSE | TRUE | FALSE | TRUE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | TRUE | TRUE | TRUE |

# Input / Output

- printf
  - Performs output to the standard output device (typically defined to be the screen).

  - It requires a format string in which we can specify:
    - The text to be printed out.
    - Specifications on how to print the values.

```
printf ("The number is %d.\n", num) ;
```

    - The format specification %d causes the value listed after the format string to be embedded in the output as a decimal number in place of %d.
    - Output will appear as: **The number is 125.**

# Input

- **scanf**
  - Performs input from the standard input device, which is the keyboard by default.
  - It requires a format string and a list of variables into which the value received from the input device will be stored.
  - It is required to put an ampersand (&) before the names of the variables.

```
scanf ("%d", &size) ;
scanf ("%c", &nextchar) ;
scanf ("%f", &length) ;
scanf (“%d  %d”, &a, &b);
```