# Programming Constructs

P by Praveen Kumar

## What is a Variable?

A **variable** is a **named memory location** used to **store data** that can change during program execution.

👉 Think of a variable as a **container** that holds a value.

## Why Do We Need Variables?

- To store user input
- To store intermediate results
- To reuse values in a program
- To make programs flexible and dynamic

## Variable Representation (Conceptual)

```
age = 25
```

- `age` → variable name
- `25` → value stored in memory

## Rules for Variable Names

- Must start with a letter or underscore
- Can contain letters, digits, underscore
- Should not be a keyword
- Case-sensitive (`age` and `Age` are different)

# What is an Expression?

An **expression** is a **combination of variables, constants, and operators** that **produces a value**.

👉 Expressions are **evaluated**, not executed.

# Types of Expressions

## 1. Arithmetic Expressions

```
total = a + b
area = length * breadth
```

## 2. Relational (Comparison) Expressions

```
a > b
marks >= 50
```

## 3. Logical Expressions

```
(age > 18) AND (citizen = true)
```

## 4. Assignment Expressions

```
x = x + 1
```

# Example

```
sum = num1 + num2
```

- `num1 + num2` → expression
- Result is assigned to `sum`

## What is a Statement?

A **statement** is a **complete instruction** that tells the computer **to perform an action**.

👉 Statements are **executed**.

## Types of Statements

### 1. Assignment Statement

```
total = price * quantity
```

### 2. Input Statement

```
READ number
```

### 3. Output Statement

```
PRINT total
```

### 4. Conditional Statement

```
IF marks >= 50 THEN
  PRINT "Pass"
ENDIF
```

### 5. Loop Statement

```
FOR i = 1 TO 5
  PRINT i
ENDFOR
```

# Difference Between Variable, Expression, and Statement

| Aspect | Variable | Expression | Statement |
| --- | --- | --- | --- |
| Meaning | Stores data | Produces a value | Performs an action |
| Example | `x` | `x + 10` | `x = x + 10` |
| Execution | Stored | Evaluated | Executed |

# Real-World Analogy

| Programming | Real World |
| --- | --- |
| Variable | Container |
| Expression | Calculation |
| Statement | Instruction |

# What is Pseudocode?

**Pseudocode** is an informal, human-readable way of describing the **logic of a program**.
It looks like a mix of **plain English and programming constructs**, but it is **not written in any specific programming language**.

## Why Do We Use Pseudocode?

Pseudocode helps to:

- ✅ Understand the **problem logic clearly**
- ✅ Plan the solution **before writing code**
- ✅ Focus on **what to do**, not **how to code**
- ✅ Communicate ideas easily with others
- ✅ Reduce errors during actual implementation

Think of pseudocode as a **bridge between problem understanding and actual coding**.

## Key Characteristics of Pseudocode

- 📝 Simple English statements
- 🔤 No strict syntax rules
- 🧠 Language-independent
- 🔁 Uses common programming concepts
- 📖 Easy to read and modify

## Common Pseudocode Keywords

| Purpose | Keyword |
| --- | --- |
| Start program | `START`, `BEGIN` |
| End program | `END`, `STOP` |
| Input | `READ`, `INPUT` |
| Output | `PRINT`, `DISPLAY` |
| Decision | `IF`, `ELSE`, `ENDIF` |
| Loop | `FOR`, `WHILE`, `REPEAT` |
| Assignment | `SET`, `=` |

## Basic Structure of Pseudocode

```
START
  Read input
  Process the data
  Display output
END
```

## Example 1: Add Two Numbers

```
START
  READ num1
  READ num2
  SET sum = num1 + num2
  PRINT sum
END
```

## Example 2: Check Even or Odd

```
START
  READ number
  IF number MOD 2 = 0 THEN
     PRINT "Even Number"
  ELSE
     PRINT "Odd Number"
  ENDIF
END
```

## xample 3: Loop Example

```
START
  FOR i = 1 TO 5
     PRINT i
  ENDFOR
END
```

## Pseudocode vs Flowchart

| Pseudocode | Flowchart |
| --- | --- |
| Text-based | Diagram-based |
| Easy to write | Visual representation |
| Preferred by programmers | Preferred for presentations |

## Where Pseudocode is Commonly Used

- 📘 Algorithm design
- 🎓 Teaching beginners
- 🧪 Exam answers
- 💼 Technical interviews
- 🧠 Problem-solving discussions

# Efficient Flowcharting Techniques

## 1. Purpose of Efficient Flowcharting

Efficient flowcharting helps to:

- ✔️ Clearly understand program logic
- ✔️ Identify logical errors early
- ✔️ Improve communication among team members
- ✔️ Simplify complex processes
- ✔️ Serve as documentation for future reference

Standard Flowchart Symbols (Use Consistently)

| Symbol | Name | Function |
|---|---|---|
|  | Start/end | An oval represents a start or end point |
|  | Arrows | A line is a connector that shows relationships between the representative shapes |
|  | Input/Output | A parallelogram represents input or output |
|  | Process | A rectangle represents a process |
|  | Decision | A diamond indicates a decision |

| Symbol | Shape | Purpose |
| --- | --- | --- |
| Terminator | Oval | Start / End |
| Process | Rectangle | Calculation / Processing |
| Input / Output | Parallelogram | Read / Print |
| Decision | Diamond | Yes / No condition |
| Flow line | Arrow | Direction of flow |
| Connector | Circle | Connects flow |

# 3. Follow a Clear Flow Direction

## Technique:

- Flow should **start at the top and move downward**
- Avoid leftward or upward flows unless necessary

## Benefit:

- Enhances readability
- Reduces confusion

# 4. Use One Entry and One Exit

## Technique:

- Each flowchart should have:
  - **One START**
  - **One END**

## Benefit:

- Prevents ambiguous logic
- Ensures structured design

# 5. Keep Flowcharts Simple and Modular

## Technique:

- Break large flowcharts into **smaller sub-flowcharts**
- Use **predefined process (subroutine) symbols**

## Benefit:

- Easier debugging
- Better maintenance

# 6. Use Meaningful Descriptions

## Technique:

- Use **clear, concise text** inside symbols
- Prefer action verbs

❌ Bad:

```
Process Data
```

✅ Good:

```
Calculate Total Marks
```

# 7. Limit Decision Complexity

## Technique:

- Each decision should test **only one condition**
- Avoid combining multiple conditions in one diamond

❌ Bad:

```
marks > 50 AND attendance > 75
```

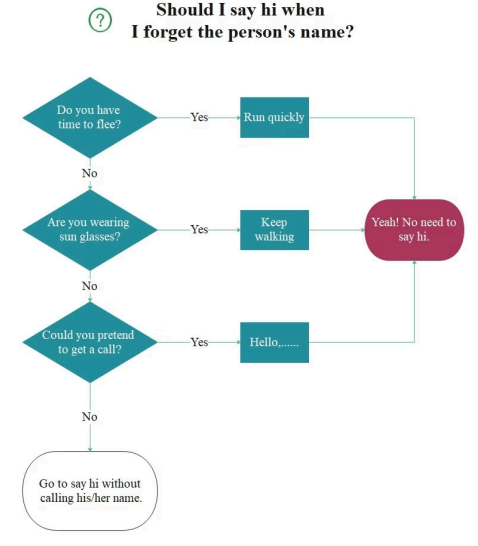
✅ Good:

- Split into two decisions

## Benefit:

- Easier testing
- Clear logic paths

# Clearly Label Decision Paths




## Technique:

- Label arrows as **YES / NO** or **TRUE / FALSE**

## Benefit:

- Eliminates ambiguity
- Helps beginners understand logic

# 9. Avoid Crossing Flow Lines

## Technique:

- Reorganize layout to avoid intersecting arrows
- Use connectors if unavoidable

## Benefit:

- Improves visual clarity

# 10. Use Connectors Effectively

## Types:

- **On-page connector** → small circle
- **Off-page connector** → pentagon

## Benefit:

- Keeps flowchart neat
- Prevents overcrowding

# 11. Maintain Consistent Spacing and Alignment

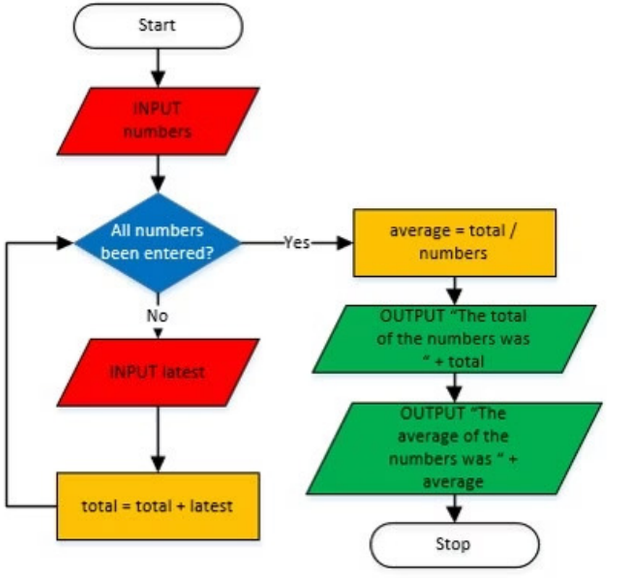
## Technique:

- Align symbols vertically
- Maintain equal spacing

## Benefit:

- Professional appearance
- Easier comprehension

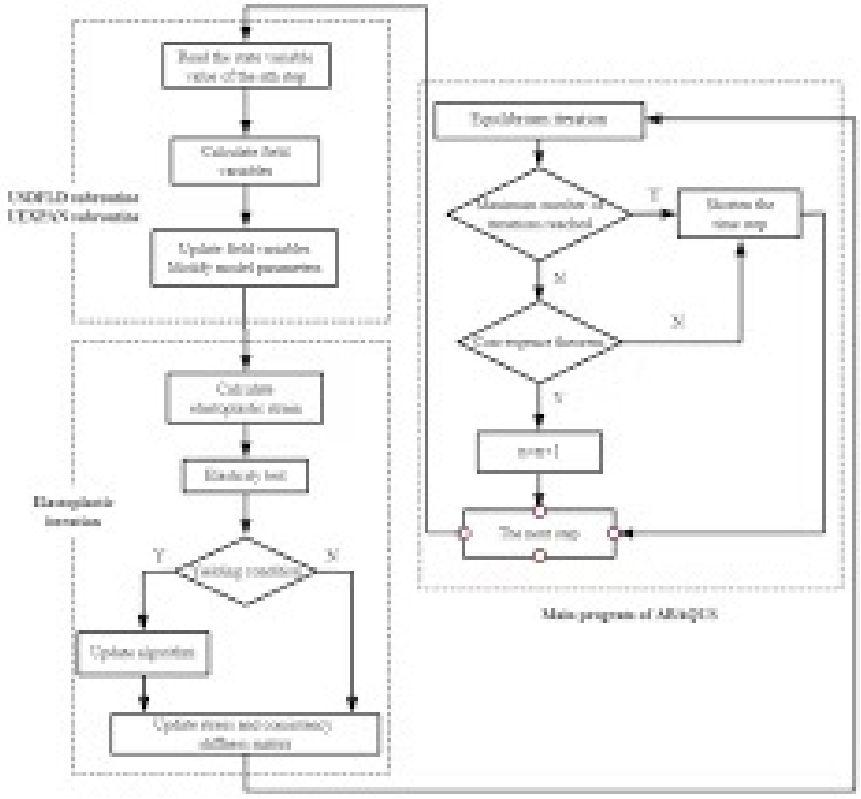
# 12. Show Loops Clearly



What is a Subroutine in a Flowchart?

A **subroutine** is a **group of instructions** that performs a specific task and can be **reused multiple times**.

👉 In flowcharts, a subroutine is represented using a **Predefined Process symbol**.



## Symbol:

- Shape: **Rectangle with double vertical sides**
- Meaning: Calls another set of steps (subroutine)

## 2. Why Use Subroutines in Flowcharts?

Using subroutines helps to:

- ✔️ Avoid repetition of logic
- ✔️ Reduce flowchart size
- ✔️ Improve readability
- ✔️ Simplify debugging
- ✔️ Promote modular thinking
- ✔️ Reflect real programming practices (functions/methods)

## 3. When Should You Use a Subroutine?

Use a subroutine when:

- The **same steps are repeated**
- A task is **logically independent**
- A process is **complex and lengthy**
- You want **clean and professional flowcharts**

# 4. How Subroutines Work in Flowcharts

## Flow:

1. Main flowchart reaches a **subroutine symbol**
2. Control transfers to the **subroutine flowchart**
3. Subroutine executes its steps
4. Control returns to the **next step in main flowchart**

# 5. Example Scenario

## Problem:

A program needs to:

- Read marks for **multiple students**
- Calculate **total and average**
- Display results

👉 The **calculation logic** repeats for every student.

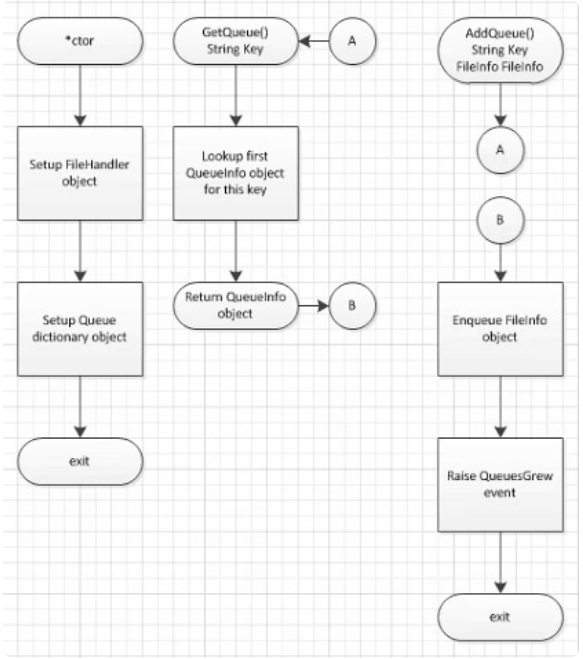
# 6. Flowchart WITHOUT Subroutine (Inefficient)

- Calculation steps repeated multiple times
- Flowchart becomes long and cluttered

❌ Not recommended

# 7. Flowchart WITH Subroutine (Efficient Design)

## Main Flowchart



# 8. Representing Subroutines Correctly

## Best Practices:

- Give **meaningful names** (e.g., `CalculateSalary`)
- Keep subroutines **single-purpose**
- Avoid input/output inside subroutine unless necessary
- Clearly indicate **return to calling point**

# 9. Passing Data to Subroutines

## Conceptually:

- Inputs → Passed to subroutine
- Outputs → Returned to main flow

Example:

```
Call Calculate_Average(marks)
Receive average
```