# Fortran Hawatri

Kia Hawatri

March 9, 2025

# Introduction to Fortran 77

Fortran, short for *Formula Translation*, is one of the oldest high-level programming languages, with its origins dating back to the 1950s. Developed by IBM for scientific and engineering applications, Fortran revolutionized the way numerical computations were performed, enabling researchers and engineers to write programs that were both efficient and portable. Fortran 77, released in 1978, is one of the most influential versions of the language, introducing structured programming features while retaining the simplicity and power that made Fortran a cornerstone of computational science.

## Why Fortran 77?

Fortran 77 represents a significant milestone in the evolution of programming languages. It introduced many features that are now considered standard in modern programming, such as structured control constructs (`IF-THEN-ELSE`, `DO` loops), character string handling, and improved input/output capabilities. Despite its age, Fortran 77 remains relevant today, particularly in legacy systems and fields such as computational physics, climate modeling, and engineering simulations. Its straightforward syntax and focus on numerical computation make it an excellent language for beginners and a powerful tool for experts.

## Who Is This Book For?

This book is designed for anyone interested in learning Fortran 77, whether you are a student, a researcher, or a professional in a technical field. No prior programming experience is required, as we will start from the basics and gradually build up to more advanced topics. For those already familiar with other programming languages, this book will help you quickly adapt to Fortran's unique features and conventions. By the end of this book, you will have a solid understanding of Fortran 77 and be able to write, debug, and optimize your own programs.

## What Will You Learn?

In this book, we will cover the following topics:

- The history and evolution of Fortran.
- Basic syntax and data types in Fortran 77.
- Control structures and loops.

- Arrays and subroutines.

- Input/output operations and file handling.

- Common pitfalls and best practices.

- Applications of Fortran 77 in scientific computing.

## How to Use This Book

Each chapter is designed to build on the previous one, with clear explanations, practical examples, and exercises to reinforce your understanding. Code snippets are provided throughout the text, and complete programs are available for download from the book's companion website. Whether you are reading this book cover-to-cover or using it as a reference, we encourage you to experiment with the examples and write your own programs to solidify your knowledge.

## A Legacy of Innovation

Fortran 77 may be a product of its time, but its influence is timeless. By learning Fortran 77, you are not only gaining a valuable skill but also connecting with a rich history of innovation in computing. As you progress through this book, you will discover why Fortran remains a trusted tool for solving some of the world's most complex problems. Welcome to the world of Fortran 77—let's begin this journey together.

# Chapter 1

# Your First Fortran 77 Program

## Writing "Hello, World!" in Fortran 77

Let's start with the classic first program. Create a file named `hello.f` and type the following:

```
C     FORTRAN 77 HELLO WORLD PROGRAM
      PROGRAM HELLOW
C     THIS IS A COMMENT LINE
      WRITE(*,*) 'HELLO WORLD'
      END
```

## Explanation of the Code

- Line 1: Comment line starting with 'C' in column 1

- Line 2: `PROGRAM HELLOW` declares the main program

- Line 3: Another comment line

- Line 4: `WRITE(*,*)` outputs text

- Line 5: `END` marks the program's conclusion

## Fortran 77 Coding Rules

### Fixed-Form Formatting

Fortran 77 uses **fixed-form source code** with strict column rules:

| Columns | Purpose |
|---------|---------|
| 1-5 | Statement labels, `FORMAT` identifiers |
| 6 | Continuation marker (any character except '0' or space) |
| 7-72 | Program statements |
| 73+ | Ignored (historical 80-column punch card limit) |

### Key Syntax Rules

- **Comments**: Start with 'C', '*', or '!' in column 1

- **Continuation**: Place a character in column 6 to continue long lines

- **Labels**: Numeric identifiers (1-99999) in columns 1-5

- **Statements**: Begin in column 7 or later

- **Case Insensitive**: WRITE, Write, and write are equivalent

## Spacing Requirements Explained

### Column Layout Example

```
123456789...
C Comment line
      PROGRAM TEST
      WRITE(*,*) 'THIS IS A
     * CONTINUED LINE'
      X = 5.0
      IF (X .GT. 0) THEN
          Y = X**2
      ENDIF
      END
```

- Line 1: Comment (C in column 1)

- Line 2: Program starts in column 7

- Line 3: Full statement in columns 7-72

- Line 4: Continuation character (*) in column 6

- Line 7: Code indentation (optional but recommended)

### Why These Rules Exist?

The column-based format dates back to punch card era programming:

- Columns 1-5: Used for card sequence numbers

- Column 6: Continuation indicator for multi-card statements

- Columns 73-80: Originally used for card identification numbers

## Common Pitfalls to Avoid

- Starting code in column 6 (reserved for continuation)
- Using lowercase letters (allowed but not traditional)
- Forgetting the continuation marker for long lines
- Writing past column 72 (code will be truncated)
- Mixing tabs and spaces (use spaces only)

## Best Practices

- Use uppercase letters for Fortran keywords
- Indent code blocks for readability (columns 7-72)
- Use comment headers for major sections
- Always include `IMPLICIT NONE` (more on this later)
- Test line length with a ruler in your editor

## Compiling Your First Program

Use a Fortran 77 compiler like `gfortran`:

```
gfortran -std=legacy hello.f -o hello
./hello
```

Output should be: `HELLO, WORLD!`

## 1.1 Commenting in Fortran 77

### The Art of Documentation

Comments are essential for writing maintainable code, especially in Fortran 77 where the fixed-format syntax can appear cryptic to modern programmers. Proper commenting helps explain complex algorithms, document assumptions, and make code accessible to future readers.

### Comment Syntax

Fortran 77 has strict rules for comments:

- Any line with `C`, `*`, or `!` in **column 1** becomes a comment
- Entire line is ignored by the compiler
- No inline comments (unlike modern languages)

- Blank lines are allowed but not considered comments

```
C THIS IS A CLASSIC FORTRAN COMMENT
* THIS VARIANT IS OFTEN USED FOR HEADERS
! SOME COMPILERS SUPPORT THIS (NON-STANDARD)
```

## Effective Commenting Techniques

### Basic Example

```
C      ==========================================
C      PROGRAM: FLUID_SIMULATION
C      PURPOSE: SOLVE NAVIER-STOKES EQUATIONS
C      AUTHOR:  J. DOE
C      DATE:    2023-08-20
C      ==========================================
       PROGRAM FLUID
C      DECLARE VARIABLES
       REAL U(100), V(100), P(100)
C      INITIALIZE ARRAYS
       DO 10 I = 1,100
         U(I) = 0.0
         V(I) = 0.0
10     CONTINUE
*      MAIN SIMULATION LOOP
       DO 20 T = 1,1000
C        UPDATE PRESSURE FIELD
         CALL CALC_PRESSURE(P,U,V)
20     CONTINUE
       END
```

## Commenting Best Practices

- **Header Blocks**: Use comments at the start of programs/subroutines to describe:

  - Program purpose
  - Input/Output specifications
  - Author and revision history
  - Special algorithms used

- **Section Dividers**:

  ```
  C      ---- INITIALIZATION PHASE ----
  ```

- **Explanatory Comments**:

```
C      APPLY COOLEY-TUKEY FFT ALGORITHM HERE
C      NOTE: ARRAY INDICES START AT 1 PER FORTRAN CONVENTION
```

- **Warnings**:

```
C      WARNING: DON'T CALL THIS SUBROUTINE RECURSIVELY
C      GLOBAL VARIABLE X MODIFIED IN SECTION 3.2
```

## Common Commenting Mistakes

- **Improper Alignment**:

```
C THIS COMMENT WILL CAUSE ERROR (C NOT IN COLUMN 1)
```

- **Redundant Comments**:

```
C      INCREMENT I
       I = I + 1  (BAD - OBVIOUS OPERATION)
```

- **Outdated Comments**:

```
C      MAX ARRAY SIZE 50 (ACTUAL SIZE IS 100 IN CODE)
```

## Advanced Commenting Strategies

### Commenting Large Blocks

```
C      ================================================
C      SUBROUTINE: MATRIX_SOLVER
C      PURPOSE:    SOLVE LINEAR SYSTEM AX=B
C      METHOD:     GAUSSIAN ELIMINATION WITH PIVOTING
C      ARGUMENTS:
C        A - COEFFICIENT MATRIX (N x N)
C        B - RIGHT-HAND SIDE VECTOR (N)
C        X - SOLUTION VECTOR (OUTPUT)
C        N - SYSTEM DIMENSION
C      ================================================
       SUBROUTINE MATRIX_SOLVER(A,B,X,N)
       DIMENSION A(N,N), B(N), X(N)
C      ... implementation ...
       END
```

**Temporary Code Exclusion**

```
C     DEBUGGING CODE - DISABLE FOR PRODUCTION
CC    WRITE(*,*) 'CURRENT VALUE:', X
C     CALL DEBUG_ROUTINE
```

## Historical Context

The column-based commenting system originated from:

- Punch card era physical constraints

- Need for quick visual identification of comments

- Limited screen space on early text terminals

## Modern Considerations

While maintaining Fortran 77 compatibility:

- Many modern editors support syntax highlighting

- Consider using lowercase for better readability:

```
c     Mixed-case comments often read better
c     Than all-uppercase text blocks
```

- Use version control instead of comment-based revision tracking

## 1.2   Variables in Fortran 77

**Variable Types**

Fortran 77 supports these fundamental data types:

| Type | Description | Example Values |
|---|---|---|
| INTEGER | Whole numbers | -3, 0, 42 |
| REAL | Single-precision floating point | 3.14, -0.001 |
| DOUBLE PRECISION | Double-precision floating point | 1.23456D+08 |
| CHARACTER | Text/String | 'Hello', 'A' |
| LOGICAL | Boolean values | .TRUE., .FALSE. |
| COMPLEX | Complex numbers | (1.0, -2.5) |

## Declaration Syntax

Variables must be declared at the start of the program/subroutine:

```
PROGRAM VARIABLES
INTEGER COUNT, INDEX
REAL TEMP, PRESSURE
CHARACTER*20 NAME
LOGICAL FLAG
DOUBLE PRECISION PI
COMPLEX WAVE
```

## Naming Rules

- Maximum 6 characters (truncated if longer)

- Must start with a letter (A-Z)

- Subsequent characters: letters/digits (0-9)

- Case insensitive: `Var` = `VAR` = `var`

- Avoid reserved words: `PROGRAM`, `END`, etc.

## Type-Specific Examples

### INTEGER

```
PROGRAM INT_EX
INTEGER AGE, YEAR
WRITE(*,*) 'ENTER BIRTH YEAR:'
READ(*,*) YEAR
AGE = 2023 - YEAR
WRITE(*,*) 'AGE:', AGE
STOP
END
```

### REAL

```
PROGRAM REAL_EX
REAL TEMP_C, TEMP_F
WRITE(*,*) 'ENTER FAHRENHEIT TEMP:'
READ(*,*) TEMP_F
TEMP_C = (TEMP_F - 32.0) * 5.0/9.0
WRITE(*,*) 'CELSIUS:', TEMP_C
STOP
END
```

**DOUBLE PRECISION**

```
PROGRAM DOUBLE_EX
DOUBLE PRECISION PI
PI = 4.0D0 * ATAN(1.0D0)
WRITE(*,*) 'PI =', PI
STOP
END
```

**CHARACTER**

```
PROGRAM CHAR_EX
CHARACTER*15 CITY
WRITE(*,*) 'ENTER YOUR CITY:'
READ(*,*) CITY
WRITE(*,*) 'CITY:', CITY
STOP
END
```

**LOGICAL**

```
PROGRAM LOG_EX
LOGICAL FLAG
FLAG = .TRUE.
IF (FLAG) THEN
    WRITE(*,*) 'CONDITION IS TRUE'
ENDIF
STOP
END
```

**COMPLEX**

```
PROGRAM COMPLEX_EX
COMPLEX Z
Z = (3.0, 4.0)  ! 3 + 4i
WRITE(*,*) 'MAGNITUDE:', ABS(Z)
STOP
END
```

## Type Conversion

Convert between types explicitly:

```
REAL X
INTEGER N
X = 3.14
N = INT(X)      ! N becomes 3
X = REAL(N)     ! X becomes 3.0
```

## Common Mistakes

- **Implicit Typing**: Variables starting with I-N are integers by default

```
K = 2.5  ! Becomes INTEGER 2 (no error!)
```

- **Solution**: Always declare `IMPLICIT NONE` first

```
PROGRAM SAFE
IMPLICIT NONE
```

- **Truncation**:

```
CHARACTER*5 NAME = 'LONDON'  ! Becomes 'LONDO'
```

- **Precision Loss**:

```
REAL PI = 3.1415926535  ! Stored as 3.141593
```

## Best Practices

- Always use `IMPLICIT NONE` to force declarations

- Choose meaningful names: `VOLTAGE` vs `V`

- Use `DOUBLE PRECISION` for scientific calculations

- Initialize variables before use

- Comment on variable purposes in complex programs

## Storage Considerations

| Type | Typical Size |
|------|--------------|
| INTEGER | 4 bytes |
| REAL | 4 bytes |
| DOUBLE PRECISION | 8 bytes |
| CHARACTER*n | n bytes |
| LOGICAL | 4 bytes (usually) |
| COMPLEX | 8 bytes ($2\times4$-byte reals) |

## 1.3 User Input and Variable Handling

### Basic Input-Process-Output Workflow

Fortran 77 programs typically follow this pattern:

1. Prompt user with WRITE(*,*)

2. Read input with READ(*,*)

3. Process data

4. Display results with WRITE(*,*)

### Single Variable Example

```
C     PROGRAM: AGE_CHECKER
C     PURPOSE: DEMONSTRATE SINGLE VARIABLE INPUT
      PROGRAM AGE_CHECK
      INTEGER AGE
C     DISPLAY PROMPT
      WRITE(*,*) 'ENTER YOUR AGE:'
C     READ INTEGER INPUT
      READ(*,*) AGE
C     DISPLAY RESULT
      WRITE(*,*) 'IN 10 YEARS YOU WILL BE:', AGE + 10
      STOP
      END
```

### Multiple Variables Example

```
C     PROGRAM: RECTANGLE_AREA
C     INPUT: LENGTH AND WIDTH
C     OUTPUT: CALCULATED AREA
      PROGRAM RECT_AREA
      REAL LENGTH, WIDTH, AREA
C     GET DIMENSIONS
      WRITE(*,*) 'ENTER LENGTH AND WIDTH (SEPARATE BY SPACE):'
      READ(*,*) LENGTH, WIDTH
C     CALCULATE AND DISPLAY
      AREA = LENGTH * WIDTH
      WRITE(*,*) 'AREA OF RECTANGLE:', AREA
      STOP
      END
```

### Type-Specific Input Handling

#### Character Input

```
C     PROGRAM: GREETER
```

```
C       DEMONSTRATES STRING HANDLING
        PROGRAM GREETER
        CHARACTER*20 NAME
C       GET USER NAME
        WRITE(*,*) 'ENTER YOUR NAME:'
        READ(*,*) NAME
C       DISPLAY GREETING
        WRITE(*,*) 'HELLO, ', TRIM(NAME), '! WELCOME!'
        STOP
        END
```

## Logical Input

```
C       PROGRAM: LOGIC_TEST
C       SHOWS BOOLEAN INPUT HANDLING
        PROGRAM LOGTEST
        LOGICAL FLAG
C       GET TRUE/FALSE INPUT
        WRITE(*,*) 'ENTER .TRUE. OR .FALSE.:'
        READ(*,*) FLAG
C       DISPLAY NEGATION
        WRITE(*,*) 'NEGATED VALUE:', .NOT.FLAG
        STOP
        END
```

## Input Validation

```
C       PROGRAM: TEMP_CONVERTER
C       WITH BASIC ERROR CHECKING
        PROGRAM TEMPCONV
        REAL FAHREN
C       INPUT LOOP
10      WRITE(*,*) 'ENTER TEMPERATURE (-200 TO 200 F):'
        READ(*,*) FAHREN
        IF (FAHREN .LT. -200 .OR. FAHREN .GT. 200) THEN
            WRITE(*,*) 'INVALID INPUT! TRY AGAIN.'
            GOTO 10
        ENDIF
C       CONVERT TO CELSIUS
        CELSIUS = (FAHREN - 32.0) * 5.0/9.0
        WRITE(*,*) 'CELSIUS TEMPERATURE:', CELSIUS
        STOP
        END
```

## Troubleshooting Input Issues

| Issue | Solution |
|---|---|
| User enters text for numeric input | Program crashes - add error handling (see Ch. 7) |
| Multiple values without spaces | Use comma/space separation: `10,20` not `10 20` |
| String longer than declaration | Truncated to variable length |
| Mixing data types | Ensure `READ` matches variable types |

## Best Practices

- Always include clear prompts before `READ` statements

- Use descriptive variable names

- Initialize variables before use

- Add comments explaining non-obvious input requirements

- Test with boundary values and invalid inputs

- Use `TRIM()` for character variables in output

## Complete Example with Comments

```
C      PROGRAM: EMPLOYEE_RECORD
C      PURPOSE: DEMONSTRATE MIXED DATA TYPE INPUT
       PROGRAM EMP_REC
       CHARACTER*15 NAME
       INTEGER AGE
       REAL SALARY
       LOGICAL FULLTIME

C      GET EMPLOYEE DETAILS
       WRITE(*,*) 'ENTER EMPLOYEE NAME:'
       READ(*,*) NAME
       WRITE(*,*) 'ENTER AGE (YEARS):'
       READ(*,*) AGE
       WRITE(*,*) 'ENTER ANNUAL SALARY:'
       READ(*,*) SALARY
       WRITE(*,*) 'FULL-TIME? (.TRUE./.FALSE.):'
       READ(*,*) FULLTIME

C      DISPLAY SUMMARY
       WRITE(*,*) 'EMPLOYEE DETAILS:'
       WRITE(*,*) 'NAME:    ', TRIM(NAME)
       WRITE(*,*) 'AGE:     ', AGE
```

```
      WRITE(*,*) 'SALARY:  $', SALARY
      WRITE(*,*) 'FULL-TIME: ', FULLTIME

      STOP
      END
```

## Notes on Input Formatting

- Use free-format `READ(*,*)` for simple programs

- Numeric input accepts:

  - Integers: `42`, `-15`
  - Reals: `3.14`, `.5`, `6.02E23`

- Logical input requires `.TRUE.` or `.FALSE.`

- Character input stops at first whitespace (use `READ` with format for spaces)

## Compilation & Testing Tip

```
# Compile with strict Fortran 77 checking
gfortran -std=legacy -Wall input_example.f -o demo
```

# 1.4 Arithmetic Operations in Fortran 77

## Fundamental Arithmetic Operators

Fortran 77 supports standard mathematical operations with this precedence:

| Operator | Operation | Example |
|----------|----------------|---------|
| `**` | Exponentiation | `X**2` |
| `*` | Multiplication | `A * B` |
| `/` | Division | `Y / Z` |
| `+` | Addition | `C + D` |
| `-` | Subtraction | `M - N` |

## Basic Operation Examples

### Simple Calculations

```
C     PROGRAM: BASIC_MATH
C     DEMONSTRATES FUNDAMENTAL OPERATIONS
      PROGRAM CALC
      REAL X, Y, RESULT

      X = 10.0
      Y = 3.0
```

```
      RESULT = X + Y
      WRITE(*,*) 'SUM:       ', RESULT

      RESULT = X - Y
      WRITE(*,*) 'DIFFERENCE:', RESULT

      RESULT = X * Y
      WRITE(*,*) 'PRODUCT:  ', RESULT

      RESULT = X / Y
      WRITE(*,*) 'QUOTIENT: ', RESULT

      RESULT = X**2 + Y**3
      WRITE(*,*) 'X² + Y³:  ', RESULT

      STOP
      END
```

## Operator Precedence

Operations follow PEMDAS rules (Parentheses, Exponents, Multiplication/Division, Addition/-Subtraction):

```
C     PROGRAM: PRECEDENCE
C     SHOWS ORDER OF OPERATIONS
      PROGRAM ORDER
      REAL A, B, C, RESULT

      A = 2.0
      B = 3.0
      C = 4.0

C     EQUIVALENT TO: (A + B) * C
      RESULT = A + B * C
      WRITE(*,*) 'WITHOUT PARENTHESES:', RESULT

C     EXPLICIT ORDERING
      RESULT = (A + B) * C
      WRITE(*,*) 'WITH PARENTHESES:   ', RESULT

      STOP
      END
```

## Mixed-Type Operations

Fortran automatically converts types during operations:

```
C      PROGRAM: TYPE_MIX
C      DEMONSTRATES INTEGER/REAL INTERACTIONS
       PROGRAM TYPEMIX
       INTEGER I
       REAL R
       DOUBLE PRECISION D

       I = 5
       R = 2.5
       D = 1.0D0

C      INTEGER + REAL = REAL
       WRITE(*,*) '5 + 2.5 =', I + R

C      REAL / INTEGER = REAL
       WRITE(*,*) '2.5 / 2 =', R / 2

C      DOUBLE PRECISION OPERATION
       D = D / 3.0D0
       WRITE(*,*) '1/3 (DP):', D

       STOP
       END
```

## Common Mathematical Functions

Fortran 77 provides intrinsic functions:

```
C      PROGRAM: MATH_FUNCS
C      SHOWS BUILT-IN MATHEMATICAL FUNCTIONS
       PROGRAM MFUNCS
       REAL X, Y, ANGLE

       X = 16.0
       Y = 2.5
       ANGLE = 45.0

C      SQUARE ROOT
       WRITE(*,*) 'SQRT(16):    ', SQRT(X)

C      EXPONENTIAL
       WRITE(*,*) 'EXP(2.5):    ', EXP(Y)

C      NATURAL LOG
       WRITE(*,*) 'LOG(2.5):    ', LOG(Y)

C      TRIG FUNCTIONS (IN RADIANS)
```

```
      WRITE(*,*) 'SIN(45°):    ', SIN(ANGLE * 3.14159 / 180.0)

C     ABSOLUTE VALUE
      WRITE(*,*) 'ABS(-2.5):   ', ABS(-Y)

C     MODULO OPERATION
      WRITE(*,*) 'MOD(17,5):   ', MOD(17, 5)

      STOP
      END
```

## Complete Example: Quadratic Equation

```
C     PROGRAM: QUADRATIC_SOLVER
C     SOLVES AX² + BX + C = 0
      PROGRAM QUAD
      REAL A, B, C, DISC, X1, X2

C     GET COEFFICIENTS
      WRITE(*,*) 'ENTER A, B, C (SEPARATED BY SPACES):'
      READ(*,*) A, B, C

C     CALCULATE DISCRIMINANT
      DISC = B**2 - 4.0*A*C

C     HANDLE COMPLEX ROOTS
      IF (DISC .LT. 0.0) THEN
          WRITE(*,*) 'COMPLEX ROOTS!'
          STOP
      ENDIF

C     CALCULATE ROOTS
      X1 = (-B + SQRT(DISC)) / (2.0*A)
      X2 = (-B - SQRT(DISC)) / (2.0*A)

      WRITE(*,*) 'ROOTS ARE:', X1, 'AND', X2
      STOP
      END
```

## Common Arithmetic Pitfalls

| Issue | Solution |
|---|---|
| Integer division: `5/2 = 2` | Use real numbers: `5.0/2.0 = 2.5` |
| Overflow with large exponents | Use `DOUBLE PRECISION` variables |
| Division by zero | Add validation checks before division |
| Mixing precedence | Use parentheses for clarity |

### Best Practices

- Use parentheses for complex expressions

- Avoid integer division when fractional results are needed

- Use `DOUBLE PRECISION` for sensitive calculations

- Check for division by zero and negative roots

- Use meaningful variable names (`VOLUME` vs `V`)

### Troubleshooting Table

| Error Message | Meaning |
| --- | --- |
| `Arithmetic overflow` | Result exceeds variable type capacity |
| `Divided by zero` | Attempted division with zero denominator |
| `Type mismatch` | Mixed incompatible types without conversion |

### Compilation Note

```
# Enable all warnings for arithmetic checks
gfortran -std=legacy -Wall -Wextra math_example.f -o demo
```

## 1.5 Type Conversion in Fortran 77

### Implicit vs. Explicit Conversion

Fortran 77 allows both implicit (automatic) and explicit (programmer-controlled) type conversion. While convenient, implicit conversion can lead to subtle bugs, making explicit conversion the safer approach.

### Implicit Type Conversion

- **Mixed-Type Operations**: Fortran automatically promotes types in expressions

  ```
  INTEGER I = 5
  REAL R = 2.5
  RESULT = I + R  ! I is converted to REAL (5.0) first
  ```

- **Assignment Conversion**: Right-hand side converted to left-hand side type

  ```
  REAL X
  X = 3   ! Integer 3 converted to REAL 3.0
  ```

- **Default Typing**: Variables starting with I-N are INTEGER by default

  ```
  K = 2.7  ! K is INTEGER → becomes 2 (truncation occurs)
  ```

## Explicit Type Conversion Functions

Fortran provides intrinsic functions for controlled conversion:

| Function | Purpose |
|----------|---------|
| INT(X) | Convert to INTEGER (truncates) |
| REAL(X) | Convert to single-precision REAL |
| DBLE(X) | Convert to DOUBLE PRECISION |
| CMPLX(X,Y) | Create COMPLEX number $(X + Yi)$ |
| ICHAR(C) | Convert character to ASCII code |
| CHAR(I) | Convert ASCII code to character |

## Code Examples

**Integer to Real**

```
      PROGRAM INT2REAL
      INTEGER COUNT
      REAL AVERAGE
      COUNT = 7
C     EXPLICIT CONVERSION TO AVOID INTEGER DIVISION
      AVERAGE = REAL(COUNT) / 2.0
      WRITE(*,*) 'AVERAGE:', AVERAGE  ! Output: 3.5
      STOP
      END
```

**Real to Integer**

```
      PROGRAM REAL2INT
      REAL TEMP = 98.6
      INTEGER ITEMP
C     TRUNCATE DECIMAL PART
      ITEMP = INT(TEMP)
      WRITE(*,*) 'INTEGER TEMP:', ITEMP  ! Output: 98
      STOP
      END
```

**Double Precision Conversion**

```
      PROGRAM DBLE_CONV
      REAL PI_SINGLE = 3.14159
      DOUBLE PRECISION PI_DOUBLE
C     PRESERVE PRECISION
      PI_DOUBLE = DBLE(PI_SINGLE)
      WRITE(*,*) 'DOUBLE PI:', PI_DOUBLE
      STOP
      END
```

## Character Conversions

```
      PROGRAM CHAR_CONV
      CHARACTER C
      INTEGER ASCII
C     CHARACTER TO ASCII
      C = 'A'
      ASCII = ICHAR(C)
      WRITE(*,*) 'ASCII CODE:', ASCII  ! Output: 65

C     ASCII TO CHARACTER
      C = CHAR(66)
      WRITE(*,*) 'CHARACTER:', C  ! Output: B
      STOP
      END
```

## Common Pitfalls

| Issue | Solution |
|---|---|
| `REAL(5/2) = 2.0` (integer division first) | Use `REAL(5)/2.0 = 2.5` |
| `INT(3.999) = 3` (truncation) | Use `NINT()` for rounding |
| Implicit real→integer conversion | Always use `INT()` explicitly |
| Precision loss in real→double | Use `DBLE()` on literals: `DBLE(0.1D0)` |

## Best Practices

- Always use `IMPLICIT NONE` to disable automatic typing

- Perform explicit conversions for clarity

- Use `NINT()` instead of `INT()` for rounding

- Avoid mixing types in complex expressions

- Comment non-obvious conversions

## Advanced Conversion: Complex Numbers

```
      PROGRAM COMPLEX_CONV
      COMPLEX Z
      REAL X, Y
      X = 3.0
      Y = 4.0
C     CREATE COMPLEX FROM REALS
      Z = CMPLX(X, Y)
```

```
      WRITE(*,*) 'COMPLEX:', Z  ! Output: (3.0,4.0)
      STOP
      END
```

## Type Conversion Rules

| Conversion | Behavior |
|---|---|
| `REAL → INTEGER` | Truncates decimal (no rounding) |
| `INTEGER → REAL` | Exact conversion |
| `REAL → DOUBLE` | Preserves precision |
| `DOUBLE → REAL` | Truncates to single precision |
| `CHARACTER → INTEGER` | ASCII code conversion |

## Why Explicit Conversion Matters

```
C DANGEROUS IMPLICIT CONVERSION EXAMPLE
      PROGRAM DANGER
      IMPLICIT NONE
      REAL A = 5.0
      INTEGER B = 2
      WRITE(*,*) A/B  ! = 2.5 (GOOD)

      REAL C = 5
      INTEGER D = 2
      WRITE(*,*) C/D  ! = 2.5 (STILL GOOD? DEPENDS ON COMPILER!)
      STOP
      END
```

## Final Recommendations

- Use `REAL()` when mixing integers and reals

- Prefer `DBLE()` for high-precision calculations

- Always validate ranges before narrowing conversions

- Test conversions at boundary values

## 1.6   Exercises

### Problem 1: Basic Program Structure

Write a Fortran 77 program that:

- Prints "MY FIRST FORTRAN PROGRAM"

- Includes proper comments

- Follows fixed-format rules

*Sample Output:*

```
MY FIRST FORTRAN PROGRAM
```

## Problem 2: Variable Declaration

Create a program that:

- Declares an integer (AGE = 25)
- Declares a real number (PI = 3.14159)
- Declares a character (INITIAL = 'A')
- Prints all variables with labels

## Problem 3: User Input Handling

Write a program that:

- Asks for user's name and birth year
- Calculates approximate age
- Prints formatted message

*Sample Input/Output:*

```
ENTER YOUR NAME: JOHN
ENTER BIRTH YEAR: 1998
HELLO JOHN, YOU ARE ABOUT 25 YEARS OLD.
```

## Problem 4: Arithmetic Operations

Create a program to calculate kinetic energy:

$$KE = \frac{1}{2}mv^2$$

Where:

- Mass (m) = 10.5 kg
- Velocity (v) = 5.2 m/s
- Print result with description

## Problem 5: Mixed-Type Calculation

Write a program that:

- Declares integer HOURS = 8
- Declares real RATE = 12.50
- Calculates total pay (HOURS * RATE)
- Explain why result is real

## Problem 6: Explicit Type Conversion

Create a program that:

- Takes a real number input (e.g., 7.89)

- Converts to integer using INT()

- Converts to nearest integer using NINT()

- Prints both results

## Problem 7: Temperature Conversion

Write a program that:

- Reads Celsius temperature

- Converts to Fahrenheit using:

$$F = \frac{9}{5}C + 32$$

- Prints both temperatures

## Problem 8: Geometric Calculations

Develop a program to calculate:

- Circle circumference $C = 2\pi r$

- Sphere volume $V = \frac{4}{3}\pi r^3$

- Use radius = 5.0

- Print both results

## Problem 9: Character Manipulation

Create a program that:

- Takes a character input

- Prints its ASCII code

- Takes an integer input (65-90)

- Prints corresponding character

## Problem 10: Precision Demonstration

Write a program that:

- Calculates $\frac{1}{3}$ as REAL

- Calculates $\frac{1}{3}$ as DOUBLE PRECISION

- Prints both results

- Explain the difference

## Challenge Problem: Unit Converter

Create an interactive program that:

- Asks user for length in kilometers

- Converts to miles (1 km = 0.621371 miles)

- Prints both values

- Uses proper type conversions

*Bonus: Add error checking for negative inputs*

# 1.7 Exercise Answers

## Problem 1: Basic Program Structure

```
C     PROBLEM 1 SOLUTION
C     PURPOSE: DEMONSTRATE BASIC PROGRAM STRUCTURE
      PROGRAM FIRST
C     OUTPUT MESSAGE
      WRITE(*,*) 'MY FIRST FORTRAN PROGRAM'
      STOP
      END
```

**Explanation:**

- Comments start with 'C' in column 1

- Program statement begins in column 7

- WRITE statement uses list-directed output

- STOP terminates execution, END concludes program

## Problem 2: Variable Declaration

```
C     PROBLEM 2 SOLUTION
      PROGRAM VARDEC
      INTEGER AGE
      REAL PI
      CHARACTER INITIAL
C     INITIALIZE VALUES
      AGE = 25
      PI = 3.14159
      INITIAL = 'A'
C     OUTPUT RESULTS
      WRITE(*,*) 'AGE:    ', AGE
      WRITE(*,*) 'PI:     ', PI
      WRITE(*,*) 'INITIAL:', INITIAL
      STOP
      END
```

**Key Points:**

- Variables declared before executable statements

- Different data types require specific declarations

- Character literals enclosed in single quotes

## Problem 3: User Input Handling

```
C     PROBLEM 3 SOLUTION
      PROGRAM AGE_CALC
      CHARACTER*20 NAME
      INTEGER B_YEAR, AGE
C     GET INPUT
      WRITE(*,*) 'ENTER YOUR NAME:'
      READ(*,*) NAME
      WRITE(*,*) 'ENTER BIRTH YEAR:'
      READ(*,*) B_YEAR
C     CALCULATE AGE
      AGE = 2023 - B_YEAR
C     OUTPUT RESULTS
      WRITE(*,*) 'HELLO ', TRIM(NAME), ', YOU ARE ABOUT ', AGE, ' YEARS OLD.'
      STOP
      END
```

**Notes:**

- CHARACTER*20 reserves 20 characters for the name

- TRIM() removes trailing spaces from the name

- Input order must match variable types

## Problem 4: Arithmetic Operations

```
C     PROBLEM 4 SOLUTION
      PROGRAM KINETIC
      REAL MASS, VEL, KE
C     INITIALIZE VALUES
      MASS = 10.5
      VEL = 5.2
C     CALCULATE KINETIC ENERGY
      KE = 0.5 * MASS * VEL**2
C     OUTPUT RESULT
      WRITE(*,*) 'KINETIC ENERGY:', KE, 'JOULES'
      STOP
      END
```

**Formula Implementation:**

$$KE = \frac{1}{2} \times 10.5 \times (5.2)^2$$

- Exponentiation operator ** used for velocity squared

- Operator precedence handled correctly

## Problem 5: Mixed-Type Calculation

```
C     PROBLEM 5 SOLUTION
      PROGRAM PAYCALC
      INTEGER HOURS
      REAL RATE, TOTAL
C     INITIALIZE VALUES
      HOURS = 8
      RATE = 12.50
C     CALCULATE PAY
      TOTAL = HOURS * RATE
C     OUTPUT RESULT
      WRITE(*,*) 'TOTAL PAY: $', TOTAL
      STOP
      END
```

**Type Conversion:**

- Integer HOURS promoted to real during multiplication

- Result TOTAL is real (100.0 instead of 100)

- Explicit conversion not needed but recommended

## Problem 6: Explicit Type Conversion

```
C     PROBLEM 6 SOLUTION
      PROGRAM CONVERT
      REAL NUM
      INTEGER ITRUNC, IROUND
C     GET INPUT
      WRITE(*,*) 'ENTER A REAL NUMBER:'
      READ(*,*) NUM
C     CONVERT
      ITRUNC = INT(NUM)
      IROUND = NINT(NUM)
C     OUTPUT RESULTS
      WRITE(*,*) 'TRUNCATED:', ITRUNC
      WRITE(*,*) 'ROUNDED:  ', IROUND
      STOP
      END
```

### Differences:

- INT(7.89) $\rightarrow$ 7 (truncation)

- NINT(7.89) $\rightarrow$ 8 (rounding)

- Always use NINT() for proper rounding

## Problem 7: Temperature Conversion

```
C     PROBLEM 7 SOLUTION
      PROGRAM TEMPCONV
      REAL CELS, FAHR
C     GET INPUT
      WRITE(*,*) 'ENTER TEMPERATURE IN CELSIUS:'
      READ(*,*) CELS
C     CONVERT
      FAHR = (9.0/5.0)*CELS + 32.0
C     OUTPUT RESULTS
      WRITE(*,*) CELS, 'C =', FAHR, 'F'
      STOP
      END
```

### Formula Notes:

- Use 9.0/5.0 instead of 9/5 to force real division

- Operator precedence handled with parentheses

## Problem 8: Geometric Calculations

```
C     PROBLEM 8 SOLUTION
      PROGRAM GEOMETRY
      REAL R, CIRCUM, VOLUME
      PARAMETER (PI = 3.14159)
C     INITIALIZE RADIUS
      R = 5.0
C     CALCULATIONS
      CIRCUM = 2 * PI * R
      VOLUME = (4.0/3.0) * PI * R**3
C     OUTPUT
      WRITE(*,*) 'CIRCUMFERENCE:', CIRCUM
      WRITE(*,*) 'VOLUME:       ', VOLUME
      STOP
      END
```

**Important:**

- PARAMETER for constant PI

- Use parentheses for fractional coefficients

- R**3 calculates radius cubed

## Problem 9: Character Manipulation

```
C     PROBLEM 9 SOLUTION
      PROGRAM CHAR_CONVERT
      CHARACTER C
      INTEGER ASCII, CODE
C     CHARACTER TO ASCII
      WRITE(*,*) 'ENTER A CHARACTER:'
      READ(*,*) C
      ASCII = ICHAR(C)
      WRITE(*,*) 'ASCII CODE:', ASCII
C     ASCII TO CHARACTER
      WRITE(*,*) 'ENTER ASCII CODE (65-90):'
      READ(*,*) CODE
      C = CHAR(CODE)
      WRITE(*,*) 'CHARACTER:', C
      STOP
      END
```

**Notes:**

- ICHAR returns ASCII value

- CHAR converts ASCII code to character

- Limited to single characters per input

## Problem 10: Precision Demonstration

```
C     PROBLEM 10 SOLUTION
      PROGRAM PRECISION
      REAL R
      DOUBLE PRECISION D
C     CALCULATIONS
      R = 1.0/3.0
      D = 1.0D0/3.0D0
C     OUTPUT
      WRITE(*,*) 'SINGLE PRECISION:', R
      WRITE(*,*) 'DOUBLE PRECISION:', D
      STOP
      END
```

### Output:

```
SINGLE PRECISION:  0.3333333
DOUBLE PRECISION:  0.3333333333333333
```

### Explanation:

- REAL provides  7 significant digits

- DOUBLE PRECISION provides  15 digits

- Use D0 suffix for double-precision literals

## Challenge Problem: Unit Converter

```
C     CHALLENGE PROBLEM SOLUTION
      PROGRAM UNIT_CONV
      REAL KM, MILES
C     INPUT LOOP
10    WRITE(*,*) 'ENTER KILOMETERS (>=0):'
      READ(*,*) KM
      IF (KM .LT. 0.0) THEN
          WRITE(*,*) 'ERROR: NEGATIVE VALUE!'
          GOTO 10
      ENDIF
C     CONVERSION
      MILES = KM * 0.621371
C     OUTPUT
      WRITE(*,*) KM, 'KM =', MILES, 'MILES'
      STOP
      END
```

### Features:

- Input validation with GOTO loop

- Real-to-real conversion maintains precision

- Clear error messaging

- Conversion factor from exact definition

# Chapter 2

# Conditional Statement in FORTRAN77

## Types of Conditional Statements

Fortran 77 provides three main conditional constructs:

| Type | Description |
|------|-------------|
| Logical IF | Single-line conditional execution |
| Block IF | Multi-line IF-THEN-ENDIF structure |
| ELSE IF | Multiple alternative conditions |
| Nested IF | IF statements within other IF blocks |
| Arithmetic IF | Three-way branching (legacy) |

## Relational Operators

| Operator | Meaning | Example |
|----------|---------|---------|
| .EQ. | Equal to | A .EQ. B |
| .NE. | Not equal to | X .NE. Y |
| .GT. | Greater than | N .GT. 0 |
| .GE. | Greater than or equal | AGE .GE. 18 |
| .LT. | Less than | TEMP .LT. 32.0 |
| .LE. | Less than or equal | COUNT .LE. 10 |

## 1. Logical IF (Single-Line)

Executes one statement if condition is true:

```
C     PROGRAM: SINGLE_LINE_IF
C     CHECKS IF NUMBER IS POSITIVE
      PROGRAM LOGIF
      REAL NUM
      WRITE(*,*) 'ENTER A NUMBER:'
```

```
      READ(*,*) NUM
      IF (NUM .GT. 0.0) WRITE(*,*) 'POSITIVE NUMBER'
      STOP
      END
```

## 2. Block IF Structure

Executes multiple statements when condition is true:

```
C     PROGRAM: TEMPERATURE_CHECK
C     DEMONSTRATES BLOCK IF
      PROGRAM BLKIF
      REAL TEMP
      WRITE(*,*) 'ENTER TEMPERATURE (°C):'
      READ(*,*) TEMP

      IF (TEMP .LT. 0.0) THEN
          WRITE(*,*) 'WARNING: BELOW FREEZING!'
          WRITE(*,*) 'TAKE WINTER PRECAUTIONS'
      ENDIF

      STOP
      END
```

## 3. IF-ELSE Structure

Handles alternative conditions:

```
C     PROGRAM: GRADE_EVALUATOR
C     DEMONSTRATES IF-ELSE
      PROGRAM IFELSE
      INTEGER SCORE
      WRITE(*,*) 'ENTER TEST SCORE (0-100):'
      READ(*,*) SCORE

      IF (SCORE .GE. 60) THEN
          WRITE(*,*) 'PASSING GRADE'
      ELSE
          WRITE(*,*) 'FAILING GRADE'
      ENDIF

      STOP
      END
```

## 4. ELSE IF Ladder

Handles multiple conditions:

```
C       PROGRAM: TAX_BRACKET
C       DEMONSTRATES ELSE IF
        PROGRAM TAXCALC
        REAL INCOME
        WRITE(*,*) 'ENTER ANNUAL INCOME:'
        READ(*,*) INCOME

        IF (INCOME .LE. 50000.0) THEN
            WRITE(*,*) 'TAX BRACKET: 10%'
        ELSE IF (INCOME .LE. 100000.0) THEN
            WRITE(*,*) 'TAX BRACKET: 20%'
        ELSE IF (INCOME .LE. 250000.0) THEN
            WRITE(*,*) 'TAX BRACKET: 30%'
        ELSE
            WRITE(*,*) 'TAX BRACKET: 40%'
        ENDIF

        STOP
        END
```

## 5. Nested IF Statements

IF blocks within other IF blocks:

```
C       PROGRAM: LOGIN_SYSTEM
C       DEMONSTRATES NESTED IF
        PROGRAM LOGIN
        CHARACTER*10 USER
        INTEGER PASS
        LOGICAL ADMIN

        WRITE(*,*) 'ENTER USERNAME:'
        READ(*,*) USER
        WRITE(*,*) 'ENTER PASSWORD:'
        READ(*,*) PASS

        IF (USER .EQ. 'ADMIN') THEN
            IF (PASS .EQ. 12345) THEN
                ADMIN = .TRUE.
                WRITE(*,*) 'ADMIN ACCESS GRANTED'
            ELSE
                WRITE(*,*) 'INCORRECT PASSWORD'
            ENDIF
        ELSE
            WRITE(*,*) 'GUEST ACCESS ONLY'
        ENDIF
```

```
      STOP
      END
```

## 6. Arithmetic IF (Legacy)

Three-way branching based on expression sign:

```
C     PROGRAM: SIGN_CHECK
C     DEMONSTRATES ARITHMETIC IF (HISTORICAL)
      PROGRAM ARIF
      INTEGER NUM
      WRITE(*,*) 'ENTER AN INTEGER:'
      READ(*,*) NUM

      IF (NUM) 10, 20, 30
10    WRITE(*,*) 'NEGATIVE NUMBER'
      GOTO 40
20    WRITE(*,*) 'ZERO'
      GOTO 40
30    WRITE(*,*) 'POSITIVE NUMBER'
40    STOP
      END
```

## Compound Conditions

Combine conditions with logical operators:

| Operator | Meaning |
|----------|---------|
| .AND.    | Both conditions true |
| .OR.     | Either condition true |
| .NOT.    | Inverts condition |

```
C     PROGRAM: WEATHER_CHECK
C     DEMONSTRATES COMPOUND CONDITIONS
      PROGRAM WEATHER
      REAL TEMP
      LOGICAL RAINING

      WRITE(*,*) 'ENTER TEMPERATURE (°C):'
      READ(*,*) TEMP
      WRITE(*,*) 'IS IT RAINING? (.TRUE./.FALSE.):'
      READ(*,*) RAINING

      IF (TEMP .GT. 25.0 .AND. .NOT. RAINING) THEN
          WRITE(*,*) 'GOOD DAY FOR BEACH'
      ELSE IF (TEMP .LT. 5.0 .OR. RAINING) THEN
          WRITE(*,*) 'STAY INDOORS'
      ELSE
```

```
      WRITE(*,*) 'NORMAL DAY'
   ENDIF

   STOP
   END
```

## Common Pitfalls

| Error | Solution |
|---|---|
| Missing `ENDIF` | Always match IF with ENDIF |
| Using = instead of `.EQ.` | Fortran uses `.EQ.` for equality |
| No space around operators | `.LT.` not `.LT.` (depends on compiler) |
| Uninitialized variables | Always initialize variables before use |

## Best Practices

- Use indentation for nested conditionals

- Always include `ELSE` blocks for error handling

- Use parentheses for complex logical expressions

- Avoid arithmetic IF in new code

- Comment complex conditions

- Test boundary conditions thoroughly

## Performance Tips

- Order conditions from most to least likely

- Use `ELSE IF` instead of multiple IFs when mutually exclusive

- Avoid deep nesting (max 3-4 levels)

- Use logical operators instead of nested IFs when possible

# 2.1 Spacing in Nested Conditional Statements

## Fixed-Format Column Rules

Fortran 77 requires strict column adherence for nested conditionals:

| Columns | Purpose |
|---|---|
| 1-5 | Optional statement labels |
| 6 | Continuation character (if needed) |
| 7-72 | Executable code and conditions |
| 73+ | Ignored (legacy punch card limit) |

## Indentation Guidelines

- **Base Level**: Start at column 7 for first IF

- **Nested Level**: Add 3 spaces per nesting level

- **Alignment**: Match THEN/ELSE/ENDIF with their IF level

- **Continuation**: Use column 6 for multi-line conditions

## Properly Formatted Example

```
C     PROGRAM: NESTED_GRADE_SYSTEM
      PROGRAM NESTED
      INTEGER SCORE
      CHARACTER*1 GRADE
      WRITE(*,*) 'ENTER EXAM SCORE (0-100):'
      READ(*,*) SCORE

C     Level 1 IF (column 7)
      IF (SCORE .GE. 90) THEN
C        Level 2 code (column 10)
         GRADE = 'A'
         IF (SCORE .EQ. 100) THEN      ! Level 2 IF (column 10)
C           Level 3 code (column 13)
            WRITE(*,*) 'PERFECT SCORE!'
         END IF                        ! Level 2 END IF
      ELSE IF (SCORE .GE. 80) THEN     ! Level 1 ELSE IF
         GRADE = 'B'
         IF (SCORE .GE. 85) THEN       ! Level 2 IF
            WRITE(*,*) 'NEARLY AN A!'
         END IF
      ELSE
         GRADE = 'F'
      END IF

      WRITE(*,*) 'YOUR GRADE: ', GRADE
      STOP
      END
```

## Column Breakdown

```
Columns: 1   5 6 7   72
         |   | | |   |
         v   v v v   v
      IF (X .GT. 0) THEN        <- Level 1 (start at 7)
         IF (Y .LT. 10) THEN    <- Level 2 (+3 spaces)
            Z = X + Y           <- Level 3 (+6 spaces)
```

```
      END IF                <- Level 2 alignment
   END IF                   <- Level 1 alignment
```

## Common Spacing Errors

| Error | Solution |
|---|---|
| Code starts in column 6 | Reserved for continuation markers |
| Uneven ELSE/END IF alignment | Use same indentation as opening IF |
| Overlapping columns (past 72) | Break lines with continuation marker |
| Mixed tabs and spaces | Use spaces only for consistency |

## Best Practices

- Use 3-space indentation per nesting level

- Align related keywords vertically:

```
      IF (...) THEN
         ...
      ELSE IF (...) THEN
         ...
      ELSE
         ...
      END IF
```

- Limit nesting depth to 3-4 levels maximum

- Use comments to mark closing END IFs:

```
      END IF  ! CLOSES TEMPERATURE CHECK
```

- Prefer this:

```
      IF (A .GT. B) THEN
         ...
      END IF
```

Over this:

```
      IF(A.GT.B)THEN
      ...
      ENDIF
```

## Multi-Line Condition Example

```
C     PROGRAM: COMPLEX_CONDITION
      PROGRAM COMPLEX
      REAL X, Y
      LOGICAL FLAG
      WRITE(*,*) 'ENTER X, Y:'
      READ(*,*) X, Y

C     Continuation marker (* in column 6)
      IF (X .GT. 100.0 .AND.
     *    Y .LT. 50.0 .OR.
     *    FLAG) THEN
          WRITE(*,*) 'CONDITION MET'
      END IF

      STOP
      END
```

## Historical Context

The strict column rules originate from:

- 80-column punch card limitations

- Physical card layout requirements

- Early compiler design constraints

## Modern Editor Tips

- Set tab stops at 6, 9, 12, etc.

- Enable column guides at 6 and 72

- Use syntax highlighting for:

    - IF/THEN/ELSE keywords
    - Continuation markers
    - Comment lines

- Configure auto-indent for nested blocks

**Troubleshooting Table**

| Compiler Error | Spacing Fix |
|---|---|
| `Unclassifiable statement` | Check code starts in column 7+ |
| `Unterminated IF block` | Align END IF with opening IF |
| `Invalid character in column 6` | Remove unintended characters |
| `Label field ignored` | Move code from columns 1-5 to 7+ |

## 2.2   Conditional Statement Examples

### Example 1: Simple Logical IF

```
C     CHECKS IF NUMBER IS POSITIVE
      PROGRAM POSCHK
      REAL NUM
      WRITE(*,*) 'ENTER A NUMBER:'
      READ(*,*) NUM
C     SINGLE-LINE CONDITIONAL
      IF (NUM .GT. 0.0) WRITE(*,*) 'POSITIVE NUMBER'
      STOP
      END
```

**Explanation:** - Uses logical IF for single-condition check - Executes WRITE only if NUM ¿ 0 - No action for negative/zero values

### Example 2: Block IF Structure

```
C     TEMPERATURE STATUS CHECKER
      PROGRAM TEMPSTAT
      REAL TEMP
      WRITE(*,*) 'ENTER TEMPERATURE (°C):'
      READ(*,*) TEMP

      IF (TEMP .LT. 0.0) THEN
          WRITE(*,*) 'FREEZING TEMPERATURE!'
      ELSE IF (TEMP .GT. 35.0) THEN
          WRITE(*,*) 'HEAT WARNING!'
      ELSE
          WRITE(*,*) 'NORMAL TEMPERATURE'
      ENDIF

      STOP
      END
```

**Features:** - Uses IF-ELSE IF-ELSE structure - Checks multiple temperature ranges - Default case for normal temperatures

## Example 3: Even/Odd Checker

```
C     DETERMINES IF NUMBER IS EVEN OR ODD
      PROGRAM EVENODD
      INTEGER NUM
      WRITE(*,*) 'ENTER AN INTEGER:'
      READ(*,*) NUM

      IF (MOD(NUM,2) .EQ. 0) THEN
          WRITE(*,*) 'EVEN NUMBER'
      ELSE
          WRITE(*,*) 'ODD NUMBER'
      ENDIF

      STOP
      END
```

**Key Points:** - Uses MOD intrinsic function - Compares remainder with .EQ. operator - Demonstrates simple IF-ELSE structure

## Example 4: Grade Calculator

```
C     CONVERTS SCORE TO LETTER GRADE
      PROGRAM GRADE
      INTEGER SCORE
      WRITE(*,*) 'ENTER EXAM SCORE (0-100):'
      READ(*,*) SCORE

      IF (SCORE .GE. 90) THEN
          WRITE(*,*) 'GRADE: A'
      ELSE IF (SCORE .GE. 80) THEN
          WRITE(*,*) 'GRADE: B'
      ELSE IF (SCORE .GE. 70) THEN
          WRITE(*,*) 'GRADE: C'
      ELSE IF (SCORE .GE. 60) THEN
          WRITE(*,*) 'GRADE: D'
      ELSE
          WRITE(*,*) 'GRADE: F'
      ENDIF

      STOP
      END
```

**Notes:** - Sequential ELSE IF structure - Conditions checked from highest to lowest - No overlap between grade ranges

## Example 5: Login System

```
C     SIMPLE USER AUTHENTICATION
      PROGRAM LOGIN
      CHARACTER*10 USER
      INTEGER PASS
      WRITE(*,*) 'ENTER USERNAME:'
      READ(*,*) USER
      WRITE(*,*) 'ENTER PASSWORD:'
      READ(*,*) PASS

      IF (USER .EQ. 'ADMIN') THEN
          IF (PASS .EQ. 12345) THEN
              WRITE(*,*) 'ACCESS GRANTED'
          ELSE
              WRITE(*,*) 'WRONG PASSWORD'
          ENDIF
      ELSE
          WRITE(*,*) 'INVALID USER'
      ENDIF

      STOP
      END
```

**Features:** - Nested IF statements - Outer check for username - Inner check for password - Multiple ELSE conditions

## Example 6: Voting Eligibility

```
C     CHECKS VOTING ELIGIBILITY
      PROGRAM VOTE
      INTEGER AGE
      LOGICAL CITIZEN
      WRITE(*,*) 'ENTER AGE:'
      READ(*,*) AGE
      WRITE(*,*) 'CITIZEN? (.TRUE./.FALSE.):'
      READ(*,*) CITIZEN

      IF (AGE .GE. 18 .AND. CITIZEN) THEN
          WRITE(*,*) 'ELIGIBLE TO VOTE'
      ELSE
          WRITE(*,*) 'NOT ELIGIBLE'
      ENDIF

      STOP
      END
```

**Explanation:** - Uses .AND. logical operator - Combines multiple conditions - Requires both conditions to be true

## Example 7: Arithmetic IF (Legacy)

```
C     NUMBER SIGN CHECK (HISTORICAL)
      PROGRAM ARIF
      INTEGER NUM
      WRITE(*,*) 'ENTER INTEGER:'
      READ(*,*) NUM

      IF (NUM) 10, 20, 30
10    WRITE(*,*) 'NEGATIVE'
      GOTO 40
20    WRITE(*,*) 'ZERO'
      GOTO 40
30    WRITE(*,*) 'POSITIVE'
40    STOP
      END
```

**Notes:** - Uses legacy arithmetic IF - Branches based on negative/zero/positive - Requires statement labels - Not recommended for new code

## Example 8: Division Validation

```
C     SAFE DIVISION PROGRAM
      PROGRAM DIVIDE
      REAL A, B, RESULT
      WRITE(*,*) 'ENTER TWO NUMBERS:'
      READ(*,*) A, B

      IF (B .EQ. 0.0) THEN
          WRITE(*,*) 'ERROR: DIVISION BY ZERO'
      ELSE
          RESULT = A / B
          WRITE(*,*) 'RESULT:', RESULT
      ENDIF

      STOP
      END
```

**Key Points:** - Prevents division by zero - Uses .EQ. for float comparison - Error handling before operation

## Example 9: Range Checker

```
C     NUMBER RANGE VALIDATION
```

```
      PROGRAM RANGE
      INTEGER NUM
      WRITE(*,*) 'ENTER NUMBER (1-100):'
      READ(*,*) NUM

      IF (NUM .LT. 1) THEN
          WRITE(*,*) 'TOO SMALL'
      ELSE IF (NUM .GT. 100) THEN
          WRITE(*,*) 'TOO LARGE'
      ELSE
          WRITE(*,*) 'VALID NUMBER'
      ENDIF

      STOP
      END
```

**Features:** - Validates input range - Separate checks for lower/upper bounds - Else case for valid numbers

## Example 10: Simple Calculator

```
C     MENU-DRIVEN CALCULATOR
      PROGRAM CALC
      REAL A, B
      INTEGER CHOICE
      WRITE(*,*) 'ENTER TWO NUMBERS:'
      READ(*,*) A, B
      WRITE(*,*) '1:ADD 2:SUB 3:MUL 4:DIV'
      READ(*,*) CHOICE

      IF (CHOICE .EQ. 1) THEN
          WRITE(*,*) 'SUM:', A+B
      ELSE IF (CHOICE .EQ. 2) THEN
          WRITE(*,*) 'DIFF:', A-B
      ELSE IF (CHOICE .EQ. 3) THEN
          WRITE(*,*) 'PRODUCT:', A*B
      ELSE IF (CHOICE .EQ. 4) THEN
          IF (B .NE. 0.0) THEN
              WRITE(*,*) 'QUOTIENT:', A/B
          ELSE
              WRITE(*,*) 'CANNOT DIVIDE BY ZERO'
          ENDIF
      ELSE
          WRITE(*,*) 'INVALID CHOICE'
      ENDIF

      STOP
```

```
      END
```

**Explanation:** - Nested IF in division case - Menu-driven interface - Multiple conditional checks - Error handling for invalid menu choices

## General Notes

- All examples use Fortran 77 fixed-format
- Column 6+ for code, column 1 for comments
- Use .EQ. instead of == for comparisons
- ELSE IF must be on same line as ELSE
- Indentation improves readability

# 2.3   Exercises: Conditional Statements

### Problem 1: Basic If-Else

Write a program that:

- Reads an integer
- Prints "POSITIVE" if ¿0, "NEGATIVE" if ¡0, "ZERO" otherwise

### Problem 2: Grade Calculator

Create a program that:

- Takes a score (0-100) as input
- Uses ELSE IF to assign grades: - A (90-100), B (80-89), C (70-79), D (60-69), F (¡60)

### Problem 3: Voting Eligibility

Write a program that:

- Checks if a user can vote
- Input: Age and citizenship status (logical)
- Output eligibility using .AND. operator

### Problem 4: Login System

Create a program with:

- Nested IF statements
- Checks username (text) and password (number)
- Grants access only if both match predefined values

## Problem 5: Leap Year Checker

Write a program that:

- Determines if a year is a leap year

- Conditions: Divisible by 4 but not 100, unless also by 400

- Use compound logical operators

## Problem 6: Temperature Advisor

Create a program that:

- Reads temperature

- Advises: - "HOT" (¿35°C), "COLD" (¡10°C), "MODERATE" otherwise

- Use ELSE IF structure

## Problem 7: Division Validator

Write a program that:

- Takes two numbers

- Divides them only if denominator is not equality 0

- Prints error message for zero denominator

## Problem 8: Vowel Checker

Create a program that:

- Reads a single character

- Uses nested IF to check if it's a vowel (A/E/I/O/U)

- Case insensitive (.EQ. with uppercase and lowercase)

## Problem 9: Simple Calculator

Write a menu-driven program that:

- Takes two numbers and operation choice (1-4)

- Performs +, -, *, / based on user selection

- Handles invalid menu choices

## Problem 10: Number Range Check

Create a program that:

- Checks if number is between 1-100

- Prints "VALID" or "INVALID"

- Adds specific messages for "TOO LOW" (¡1) and "TOO HIGH" (¿100)

## Challenge Problem: ATM Simulator

Write a program that:

- Checks PIN (4-digit number)

- Checks account balance before withdrawal

- Outputs: - "INVALID PIN" if wrong - "INSUFFICIENT FUNDS" if balance ¡ requested amount - "SUCCESS" otherwise

## 2.4  Exercise Answers: Conditional Statements

### Problem 1: Basic If-Else

```
C     DETERMINES NUMBER SIGN
      PROGRAM POSNEG
      INTEGER NUM
      WRITE(*,*) 'ENTER AN INTEGER:'
      READ(*,*) NUM

      IF (NUM .GT. 0) THEN
          WRITE(*,*) 'POSITIVE'
      ELSE IF (NUM .LT. 0) THEN
          WRITE(*,*) 'NEGATIVE'
      ELSE
          WRITE(*,*) 'ZERO'
      END IF
      STOP
      END
```

**Explanation:** - Uses IF-ELSE IF-ELSE structure - Checks ¿0 first, then ¡0, default to zero - .GT. and .LT. relational operators

### Problem 2: Grade Calculator

```
C     ASSIGNS LETTER GRADES
      PROGRAM GRADE
      INTEGER SCORE
```

```
      WRITE(*,*) 'ENTER SCORE (0-100):'
      READ(*,*) SCORE

      IF (SCORE .GE. 90) THEN
          WRITE(*,*) 'GRADE: A'
      ELSE IF (SCORE .GE. 80) THEN
          WRITE(*,*) 'GRADE: B'
      ELSE IF (SCORE .GE. 70) THEN
          WRITE(*,*) 'GRADE: C'
      ELSE IF (SCORE .GE. 60) THEN
          WRITE(*,*) 'GRADE: D'
      ELSE
          WRITE(*,*) 'GRADE: F'
      END IF
      STOP
      END
```

**Key Points:** - ELSE IF ladder structure - Descending order of conditions - Inclusive lower bounds

## Problem 3: Voting Eligibility

```
C     CHECKS VOTING RIGHTS
      PROGRAM VOTE
      INTEGER AGE
      LOGICAL CITIZEN
      WRITE(*,*) 'ENTER AGE:'
      READ(*,*) AGE
      WRITE(*,*) 'CITIZEN? (.TRUE./.FALSE.):'
      READ(*,*) CITIZEN

      IF (AGE .GE. 18 .AND. CITIZEN) THEN
          WRITE(*,*) 'ELIGIBLE TO VOTE'
      ELSE
          WRITE(*,*) 'NOT ELIGIBLE'
      END IF
      STOP
      END
```

**Features:** - Uses .AND. logical operator - Combines numeric and logical input - Single condition check

## Problem 4: Login System

```
C     SIMPLE AUTHENTICATION
      PROGRAM LOGIN
      CHARACTER*10 USER
      INTEGER PASS
```

```fortran
      WRITE(*,*) 'ENTER USERNAME:'
      READ(*,*) USER
      WRITE(*,*) 'ENTER PASSWORD:'
      READ(*,*) PASS

      IF (USER .EQ. 'ADMIN') THEN
          IF (PASS .EQ. 1234) THEN
              WRITE(*,*) 'ACCESS GRANTED'
          ELSE
              WRITE(*,*) 'WRONG PASSWORD'
          END IF
      ELSE
          WRITE(*,*) 'INVALID USER'
      END IF
      STOP
      END
```

**Explanation:** - Nested IF structure - Outer check for username - Inner check for password - Character comparison with .EQ.

## Problem 5: Leap Year Checker

```fortran
C     DETERMINES LEAP YEARS
      PROGRAM LEAP
      INTEGER YEAR
      LOGICAL COND1, COND2, COND3
      WRITE(*,*) 'ENTER YEAR:'
      READ(*,*) YEAR

      COND1 = MOD(YEAR,4) .EQ. 0
      COND2 = MOD(YEAR,100) .NE. 0
      COND3 = MOD(YEAR,400) .EQ. 0

      IF ((COND1 .AND. COND2) .OR. COND3) THEN
          WRITE(*,*) 'LEAP YEAR'
      ELSE
          WRITE(*,*) 'NOT A LEAP YEAR'
      END IF
      STOP
      END
```

**Logic:** - Uses MOD for divisibility checks - Combines conditions with .AND./.OR. - Follows Gregorian calendar rules

## Problem 6: Temperature Advisor

```fortran
C     WEATHER ADVISORY SYSTEM
```

```
      PROGRAM TEMPADV
      REAL TEMP
      WRITE(*,*) 'ENTER TEMPERATURE (°C):'
      READ(*,*) TEMP

      IF (TEMP .GT. 35.0) THEN
          WRITE(*,*) 'HOT'
      ELSE IF (TEMP .LT. 10.0) THEN
          WRITE(*,*) 'COLD'
      ELSE
          WRITE(*,*) 'MODERATE'
      END IF
      STOP
      END
```

**Structure:** - Three-way ELSE IF - Floating point comparisons - Explicit temperature thresholds

## Problem 7: Division Validator

```
C     SAFE DIVISION PROGRAM
      PROGRAM DIVIDE
      REAL A, B
      WRITE(*,*) 'ENTER TWO NUMBERS:'
      READ(*,*) A, B

      IF (B .EQ. 0.0) THEN
          WRITE(*,*) 'ERROR: DIVISION BY ZERO'
      ELSE
          WRITE(*,*) 'RESULT:', A/B
      END IF
      STOP
      END
```

**Safety:** - Checks denominator before division - Uses .EQ. for float comparison - Prevents runtime errors

## Problem 8: Vowel Checker

```
C     VOWEL IDENTIFICATION
      PROGRAM VOWEL
      CHARACTER C
      WRITE(*,*) 'ENTER A LETTER:'
      READ(*,*) C

      IF (C .EQ. 'A' .OR. C .EQ. 'E' .OR.
     *    C .EQ. 'I' .OR. C .EQ. 'O' .OR.
     *    C .EQ. 'U' .OR. C .EQ. 'a' .OR.
```

```
*     C .EQ. 'e' .OR. C .EQ. 'i' .OR.
*     C .EQ. 'o' .OR. C .EQ. 'u') THEN
          WRITE(*,*) 'VOWEL'
     ELSE
          WRITE(*,*) 'NOT A VOWEL'
     END IF
     STOP
     END
```

**Features:** - Multi-line condition with continuation (* in column 6) - Checks both uppercase and lowercase - Uses .OR. for multiple possibilities

## Problem 9: Simple Calculator

```
C     MENU-DRIVEN CALCULATOR
      PROGRAM CALC
      REAL A, B
      INTEGER CHOICE
      WRITE(*,*) 'ENTER TWO NUMBERS:'
      READ(*,*) A, B
      WRITE(*,*) '1:ADD 2:SUB 3:MUL 4:DIV'
      READ(*,*) CHOICE

      IF (CHOICE .EQ. 1) THEN
          WRITE(*,*) 'SUM:', A+B
      ELSE IF (CHOICE .EQ. 2) THEN
          WRITE(*,*) 'DIFFERENCE:', A-B
      ELSE IF (CHOICE .EQ. 3) THEN
          WRITE(*,*) 'PRODUCT:', A*B
      ELSE IF (CHOICE .EQ. 4) THEN
          IF (B .NE. 0.0) THEN
              WRITE(*,*) 'QUOTIENT:', A/B
          ELSE
              WRITE(*,*) 'DIVISION BY ZERO!'
          END IF
      ELSE
          WRITE(*,*) 'INVALID CHOICE'
      END IF
      STOP
      END
```

**Structure:** - Nested IF for division check - ELSE IF ladder for menu options - ELSE clause for invalid input

## Problem 10: Number Range Check

```
C     RANGE VALIDATION
```

```
      PROGRAM RANGE
      INTEGER NUM
      WRITE(*,*) 'ENTER NUMBER (1-100):'
      READ(*,*) NUM

      IF (NUM .LT. 1) THEN
          WRITE(*,*) 'TOO LOW'
      ELSE IF (NUM .GT. 100) THEN
          WRITE(*,*) 'TOO HIGH'
      ELSE
          WRITE(*,*) 'VALID'
      END IF
      STOP
      END
```

**Logic:** - Checks lower bound first - Then upper bound - Else validates number

## Challenge Problem: ATM Simulator

```
C     ATM TRANSACTION SYSTEM
      PROGRAM ATM
      INTEGER PIN, CORRECT_PIN
      REAL BALANCE, AMOUNT
      PARAMETER (CORRECT_PIN = 5678)
      BALANCE = 2500.0

      WRITE(*,*) 'ENTER PIN:'
      READ(*,*) PIN
      WRITE(*,*) 'ENTER WITHDRAWAL AMOUNT:'
      READ(*,*) AMOUNT

      IF (PIN .NE. CORRECT_PIN) THEN
          WRITE(*,*) 'INVALID PIN'
      ELSE IF (AMOUNT .GT. BALANCE) THEN
          WRITE(*,*) 'INSUFFICIENT FUNDS'
      ELSE
          WRITE(*,*) 'SUCCESS'
      END IF
      STOP
      END
```

**Security:** - PIN validation first - Balance check second - PARAMETER for secure PIN storage

# Chapter 3

# LOOPS & LOOPS IN' FORTRAN77

## 3.1 Loops in Fortran 77

### Types of Loops

Fortran 77 provides three main looping constructs:

| Type | Description |
|---|---|
| DO Loop | Fixed iteration count |
| DO-WHILE | Conditional looping |
| Arithmetic IF (legacy) | GOTO-based iteration |

### 1. DO Loop (Fixed Iterations)

```
C     SIMPLE DO LOOP EXAMPLE
      PROGRAM DO_LOOP
      INTEGER I
C     LOOP FROM 1 TO 5 (STEP 1)
      DO 10 I = 1, 5
          WRITE(*,*) 'ITERATION:', I
10    CONTINUE
      STOP
      END
```

**Key Features:**

- `DO 10 I = 1, 5` - Label 10 marks loop end

- `CONTINUE` - Loop termination marker

- Default step size = 1

- Loop variable (I) automatically increments

## DO Loop with Step

```
C     LOOP WITH STEP VALUE
      PROGRAM DO_STEP
      INTEGER N
C     COUNTDOWN FROM 10 TO 0, STEP -2
      DO 20 N = 10, 0, -2
          WRITE(*,*) 'COUNT:', N
20    CONTINUE
      STOP
      END
```

**Explanation:** - Step value (-2) specified after range - Loop variable decreases by 2 each iteration - Loop ends when N ¡ 0

## 2. DO-WHILE Loop (Conditional)

```
C     CONDITIONAL LOOP EXAMPLE
      PROGRAM DOWHILE
      REAL TEMP
      TEMP = 100.0
C     LOOP WHILE TEMPERATURE > 32.0
30    IF (TEMP .GT. 32.0) THEN
          WRITE(*,*) 'CURRENT TEMP:', TEMP
          TEMP = TEMP - 10.0
          GOTO 30
      END IF
      STOP
      END
```

**Structure:** - Label 30 marks loop start - Condition checked before each iteration - GOTO creates loopback - Variable modification inside loop

## 3. Nested DO Loops

```
C     MULTIPLICATION TABLE GENERATOR
      PROGRAM NESTED
      INTEGER I, J
C     OUTER LOOP (ROWS)
      DO 40 I = 1, 5
C        INNER LOOP (COLUMNS)
         DO 50 J = 1, 5
             WRITE(*,*) I, 'X', J, '=', I*J
50       CONTINUE
40    CONTINUE
      STOP
      END
```

**Features:** - Outer loop (I) controls rows - Inner loop (J) controls columns - Unique labels for each loop (40, 50) - Proper indentation for readability

## 4. Loop Control Statements

Fortran 77 has limited control flow:

| Statement | Purpose |
|-----------|---------|
| GOTO | Jump to label |
| EXIT | Terminate loop (non-standard) |
| CYCLE | Skip iteration (non-standard) |

```
C     LOOP EXIT EXAMPLE
      PROGRAM LOOPEXIT
      INTEGER COUNT
      COUNT = 1
60    IF (COUNT .LE. 10) THEN
          IF (COUNT .EQ. 5) GOTO 70
          WRITE(*,*) COUNT
          COUNT = COUNT + 1
          GOTO 60
      END IF
70    STOP
      END
```

**Explanation:** - Exits loop when COUNT reaches 5 - Uses GOTO to jump out of loop - Limited to label-based control

## 5. Legacy Arithmetic IF Loop

```
C     HISTORICAL APPROACH (NOT RECOMMENDED)
      PROGRAM ARIF
      INTEGER N
      N = 5
80    WRITE(*,*) N
      N = N - 1
      IF (N) 90, 90, 80
90    STOP
      END
```

**Behavior:** - IF (N) 90, 90, 80 branches to: - 90 if N ¡ 0 - 90 if N = 0 - 80 if N ¿ 0 - Creates countdown from 5 to 0

## Loop Variable Rules

- Loop variable must be INTEGER

- Modification inside loop is allowed but discouraged

- Value persists after loop exit

- Zero-trip loops possible:

```
DO 100 I = 5, 1  ! Never executes
```

## Common Loop Patterns

### Summation

```
C     SUM FIRST 10 NATURAL NUMBERS
      PROGRAM SUMMATION
      INTEGER I, SUM
      SUM = 0
      DO 110 I = 1, 10
          SUM = SUM + I
110   CONTINUE
      WRITE(*,*) 'TOTAL:', SUM
      STOP
      END
```

### Input Validation

```
C     REPEAT UNTIL VALID INPUT
      PROGRAM VALIDATE
      REAL X
120   WRITE(*,*) 'ENTER POSITIVE NUMBER:'
      READ(*,*) X
      IF (X .LE. 0.0) GOTO 120
      WRITE(*,*) 'THANK YOU'
      STOP
      END
```

## Best Practices

- Use DO loops for known iterations

- Prefer DO-WHILE for condition-based loops

- Avoid modifying loop variables

- Use unique labels for nested loops

- Indent loop bodies consistently

- Comment complex loop logic

## Common Errors

| Error | Solution |
|---|---|
| Missing CONTINUE | Ensure every DO has matching label |
| Infinite loop | Verify exit condition changes |
| Label mismatch | Check GOTO targets |
| Real loop variables | Use INTEGER for counters |

## Performance Considerations

- Place loop-invariant code outside

- Minimize I/O inside loops

- Avoid complex conditions in DO-WHILE

- Use INTEGER for counters

- Prefer DO loops over GOTO when possible

# 3.2 Loop Examples in Fortran 77

## 1. DO Loops (Fixed Iterations)

### Example 1: Basic Number Sequence

```
C     PRINT NUMBERS 1 TO 5
      PROGRAM DO1
      INTEGER I
C     START LOOP AT 1, END AT 5, STEP 1
      DO 10 I = 1, 5
          WRITE(*,*) 'NUMBER:', I
10    CONTINUE
      STOP
      END
```

**Explanation:** - Loop variable I starts at 1, increments by 1 - Executes exactly 5 times - CONTINUE marks loop end (label 10)

### Example 2: Step Value in Reverse

```
C     COUNTDOWN FROM 10 TO 0
      PROGRAM DO2
      INTEGER COUNT
C     STEP BY -2 (DECREMENT)
      DO 20 COUNT = 10, 0, -2
          WRITE(*,*) 'COUNTDOWN:', COUNT
20    CONTINUE
      STOP
      END
```

**Features:** - Negative step value (-2) - Loop ends when COUNT ¡ 0 - Output: 10, 8, 6, 4, 2, 0

**Example 3: Nested Multiplication Table**

```
C     5x5 MULTIPLICATION TABLE
      PROGRAM DO3
      INTEGER I, J
C     OUTER LOOP FOR ROWS
      DO 30 I = 1, 5
C         INNER LOOP FOR COLUMNS
          DO 40 J = 1, 5
              WRITE(*,*) I, 'x', J, '=', I*J
40        CONTINUE
30    CONTINUE
      STOP
      END
```

**Key Points:** - Outer loop (I) runs 5 times - Inner loop (J) completes fully for each I - Unique labels (30, 40) for each loop

## 2. DO-WHILE Loops (Conditional)

**Example 1: Temperature Monitor**

```
C     COOLING SIMULATION
      PROGRAM WHILE1
      REAL TEMP
      TEMP = 100.0
50    IF (TEMP .GT. 32.0) THEN
          WRITE(*,*) 'Current Temp:', TEMP
          TEMP = TEMP - 10.0
          GOTO 50
      END IF
      STOP
      END
```

**Explanation:** - Loop continues while TEMP ¿ 32.0 - GOTO 50 creates loopback - TEMP decreases by 10 each iteration

**Example 2: Sum Until Threshold**

```
C     SUM NUMBERS UNTIL TOTAL > 100
      PROGRAM WHILE2
      INTEGER NUM, TOTAL
      TOTAL = 0
60    IF (TOTAL .LE. 100) THEN
          WRITE(*,*) 'Enter number:'
          READ(*,*) NUM
```

```
      TOTAL = TOTAL + NUM
      GOTO 60
   END IF
   WRITE(*,*) 'Final total:', TOTAL
   STOP
   END
```

**Features:** - Loop until TOTAL exceeds 100 - User input inside loop - Condition checked before each iteration

**Example 3: Input Validation**

```
C     VALIDATE POSITIVE INPUT
      PROGRAM WHILE3
      REAL X
70    WRITE(*,*) 'Enter positive value:'
      READ(*,*) X
      IF (X .LE. 0.0) THEN
          WRITE(*,*) 'Invalid! Try again'
          GOTO 70
      END IF
      WRITE(*,*) 'Accepted:', X
      STOP
      END
```

**Key Points:** - Forces valid input using GOTO - Loop continues until X ¿ 0 - No separate loop variable needed

# 3. Arithmetic IF Loops (Legacy)

## Example 1: Simple Countdown

```
C     COUNTDOWN USING ARITHMETIC IF
      PROGRAM ARIF1
      INTEGER N
      N = 5
80    WRITE(*,*) N
      N = N - 1
C     IF(N) neg,zero,pos labels
      IF (N) 90, 90, 80
90    STOP
      END
```

**Explanation:** - IF (N) branches to 90 if N  0 - Branches to 80 if N ¿ 0 - Output: 5 4 3 2 1 0

## Example 2: Sum Positive Numbers

```
C     SUM INPUT UNTIL NEGATIVE
```

```
      PROGRAM ARIF2
      INTEGER NUM, SUM
      SUM = 0
100   WRITE(*,*) 'Enter number (negative to stop):'
      READ(*,*) NUM
C     BRANCH BASED ON NUM SIGN
      IF (NUM) 110, 120, 120
110   WRITE(*,*) 'Total:', SUM
      STOP
120   SUM = SUM + NUM
      GOTO 100
      END
```

**Features:** - 110: Negative number exit - 120: Zero/positive accumulation - Three-way branching

**Example 3: Password Attempts**

```
C     LIMITED PASSWORD ATTEMPTS
      PROGRAM ARIF3
      INTEGER TRIES, PASS
      TRIES = 3
      PASS = 1234
130   WRITE(*,*) 'Enter password (', TRIES, 'left):'
      READ(*,*) INPUT
      IF (INPUT .NE. PASS) THEN
          TRIES = TRIES - 1
          IF (TRIES) 140, 140, 130
      ELSE
          WRITE(*,*) 'Access granted'
          STOP
      END IF
140   WRITE(*,*) 'Account locked'
      STOP
      END
```

**Key Points:** - Gives 3 password attempts - Uses Arithmetic IF for attempt counting - Combines modern IF-THEN with legacy branching

## 3.3   Spacing for Loops and Nested Loops

### Fixed-Format Column Rules

Fortran 77 requires strict adherence to column-based formatting:

| Columns | Purpose |
|---------|---------|
| 1-5 | Statement labels (optional) |
| 6 | Continuation character |
| 7-72 | Executable code |
| 73-80 | Ignored (historical) |

## Basic Loop Structure

```
C     BASIC DO LOOP
      PROGRAM LOOP1
      INTEGER I
C     DO statement starts at column 7
      DO 10 I = 1, 5
         WRITE(*,*) I  ! Body indented 3 spaces
10    CONTINUE          ! Label 10 in columns 1-5
      STOP
      END
```

## Nested Loop Spacing

```
C     NESTED LOOPS
      PROGRAM NESTED
      INTEGER I, J
C     Outer loop
      DO 20 I = 1, 3
C        Inner loop (indented 3 spaces)
         DO 30 J = 1, 2
            WRITE(*,*) I, J  ! Double indentation
30       CONTINUE           ! Inner label
20    CONTINUE              ! Outer label
      STOP
      END
```

## Key Spacing Rules

- **DO Statement**: Start at column 7

- **Labels**: Place in columns 1-5

- **Body**: Indent 3-6 spaces per nesting level

- **CONTINUE**: Align with corresponding DO

## Proper Column Layout

```
Columns: 1    5 6 7    72
         |    | | |    |
         v    v v v    v
```

```
      DO 40 I = 1, 3      <- Outer loop (col 7)
         DO 50 J = 1, 2 <- Inner loop (+3 spaces)
            ...           <- Body (+6 spaces)
50       CONTINUE         <- Inner label (col 1-5)
40    CONTINUE            <- Outer label
```

## Common Mistakes

| Error | Solution |
|-------|----------|
| Code starts in column 6 | Shift to column 7+ |
| Missing CONTINUE label | Ensure every DO has matching label |
| Overlapping labels | Use unique numbers (10, 20, 30, etc.) |
| Body not indented | Add 3-6 spaces per nesting level |

## Best Practices

- **Indentation**: Use 3 spaces per nesting level

- **Labels**: Increment by 10s (10, 20, 30) for flexibility

- **Comments**: Describe loop purpose

- **Deep Nesting**: Avoid beyond 3 levels

- **Variable Names**: Use meaningful names (ROW/COL vs I/J)

## Advanced Example: Triple Nested Loop

```
C     3D MATRIX INITIALIZATION
      PROGRAM TRIPLE
      INTEGER X, Y, Z
C     Outer loop
      DO 100 X = 1, 2
C        Middle loop
         DO 200 Y = 1, 3
C           Inner loop
            DO 300 Z = 1, 2
               WRITE(*,*) X, Y, Z
300         CONTINUE
200      CONTINUE
100   CONTINUE
      STOP
      END
```

## Legacy Approach (Arithmetic IF)

```
C     NOT RECOMMENDED - HISTORICAL USE
      PROGRAM LEGACY
```

```
      INTEGER K
      K = 1
400   WRITE(*,*) K
      K = K + 1
      IF (K - 5) 400, 400, 500
500   STOP
      END
```

## Performance Tips

- Place WRITE/READ outside loops when possible

- Prefer DO loops over GOTO for readability

- Initialize variables before loops

- Avoid modifying loop counters

## 3.4 Exercises: Loops in Fortran 77

### Problem 1: Basic DO Loop

Write a program that:

- Uses a DO loop to print numbers 1 through 10

- Follows fixed-format column rules

- Uses a CONTINUE statement

### Problem 2: Step Value Practice

Create a program that:

- Prints even numbers between 2 and 20

- Uses a DO loop with step value 2

- Labels loop termination properly

### Problem 3: Nested Loop Grid

Write a program that:

- Uses nested DO loops to print all (i,j) pairs for a 3x3 grid

- Outer loop for i-values (1-3)

- Inner loop for j-values (1-3)

## Problem 4: Conditional Summation

Create a program that:

- Uses a DO-WHILE structure (IF-GOTO)

- Accumulates numbers until total exceeds 100

- Shows intermediate sums

## Problem 5: Input Validation

Write a program that:

- Repeatedly asks for positive number input

- Uses a DO-WHILE loop with .LE. operator

- Exits only when valid input received

## Problem 6: Pattern Printing

Create a program that:

- Uses nested loops to print:

  ```
  *
  **
  ***
  ```

- Each level adds one more asterisk

## Problem 7: Factorial Calculator

Write a program that:

- Calculates factorial of user-input number

- Uses a DO loop for multiplication

- Handles 0! = 1 case

## Problem 8: Early Exit Loop

Create a program that:

- Reads numbers until negative entered

- Uses GOTO to exit loop early

- Accumulates positive numbers

## Problem 9: Legacy Countdown

Write a program that:

- Uses arithmetic IF loop structure

- Counts down from 5 to 1

- Prints "LIFTOFF!" at end

## Problem 10: Login System

Create a program that:

- Gives 3 password attempts

- Uses loop with attempt counter

- Shows remaining attempts

- Uses fixed-format spacing

## Challenge Problem: Prime Checker

Write a program that:

- Checks if input number is prime

- Uses nested loops and MOD function

- Optimizes loop range for efficiency

# 3.5 Exercise Answers: Loops in Fortran 77

## Problem 1: Basic DO Loop

```
C     PRINTS NUMBERS 1 TO 10
      PROGRAM DO_LOOP
      INTEGER I
C     LOOP FROM 1 TO 10
      DO 10 I = 1, 10
          WRITE(*,*) I
10    CONTINUE
      STOP
      END
```

**Explanation:** - Loop variable `I` runs from 1 to 10 - `CONTINUE` at label 10 marks loop end - Implicit increment of 1

## Problem 2: Step Value Practice

```
C     PRINTS EVEN NUMBERS 2-20
      PROGRAM EVENS
      INTEGER N
C     STEP BY 2
      DO 20 N = 2, 20, 2
          WRITE(*,*) N
20    CONTINUE
      STOP
      END
```

**Features:** - Step value 2 specified - Loop ends at 20 (inclusive) - Output: 2, 4, 6,..., 20

## Problem 3: Nested Loop Grid

```
C     PRINTS 3x3 GRID COORDINATES
      PROGRAM GRID
      INTEGER I, J
C     OUTER LOOP (ROWS)
      DO 30 I = 1, 3
C         INNER LOOP (COLUMNS)
          DO 40 J = 1, 3
              WRITE(*,*) '(', I, ',', J, ')'
40        CONTINUE
30    CONTINUE
      STOP
      END
```

**Output:**

```
(1,1)
(1,2)
...
(3,3)
```

## Problem 4: Conditional Summation

```
C     SUMS NUMBERS UNTIL >100
      PROGRAM SUM100
      INTEGER NUM, TOTAL
      TOTAL = 0
50    IF (TOTAL .LE. 100) THEN
          WRITE(*,*) 'Current total:', TOTAL
          WRITE(*,*) 'Enter number:'
          READ(*,*) NUM
          TOTAL = TOTAL + NUM
          GOTO 50
```

```
      END IF
      WRITE(*,*) 'Final total:', TOTAL
      STOP
      END
```

**Logic:** - Loop continues while total  100 - User input inside loop - `GOTO 50` creates repetition

## Problem 5: Input Validation

```
C     ENSURES POSITIVE INPUT
      PROGRAM VALIDATE
      REAL X
60    WRITE(*,*) 'Enter positive number:'
      READ(*,*) X
      IF (X .LE. 0.0) THEN
          WRITE(*,*) 'Invalid input!'
          GOTO 60
      END IF
      WRITE(*,*) 'Accepted:', X
      STOP
      END
```

**Key Points:** - Forces valid input using `GOTO` - Loop continues until X ¿ 0 - No explicit loop counter needed

## Problem 6: Pattern Printing

```
C     PRINTS RIGHT TRIANGLE PATTERN
      PROGRAM PATTERN
      INTEGER I, J
      DO 70 I = 1, 3
          DO 80 J = 1, I
              WRITE(*,*) '*'
80        CONTINUE
          WRITE(*,*) ' '  ! Newline
70    CONTINUE
      STOP
      END
```

**Output:**

```
*
**
***
```

## Problem 7: Factorial Calculator

```
C      CALCULATES N!
       PROGRAM FACT
       INTEGER N, I, FACTORIAL
       FACTORIAL = 1
       WRITE(*,*) 'Enter number:'
       READ(*,*) N
C      HANDLE 0! = 1 CASE
       IF (N .EQ. 0) GOTO 90
       DO 100 I = 1, N
           FACTORIAL = FACTORIAL * I
100    CONTINUE
90     WRITE(*,*) N, '! =', FACTORIAL
       STOP
       END
```

**Note:** - Special case for 0! handled - Loop multiplies sequentially

## Problem 8: Early Exit Loop

```
C      SUMS POSITIVE NUMBERS
       PROGRAM SUM_POS
       INTEGER NUM, TOTAL
       TOTAL = 0
110    WRITE(*,*) 'Enter number (negative to stop):'
       READ(*,*) NUM
       IF (NUM .LT. 0) GOTO 120
       TOTAL = TOTAL + NUM
       GOTO 110
120    WRITE(*,*) 'Total:', TOTAL
       STOP
       END
```

**Explanation:** - GOTO 120 exits on negative input - Accumulates positive numbers - Infinite loop until exit condition

## Problem 9: Legacy Countdown

```
C      COUNTDOWN USING ARITHMETIC IF
       PROGRAM LIFTOFF
       INTEGER K
       K = 5
130    WRITE(*,*) K
       K = K - 1
       IF (K) 140, 140, 130
140    WRITE(*,*) 'LIFTOFF!'
```

```
      STOP
      END
```

**Output:**

```
5
4
3
2
1
LIFTOFF!
```

## Problem 10: Login System

```
C     PASSWORD ATTEMPT SYSTEM
      PROGRAM LOGIN
      INTEGER TRIES, PASS
      TRIES = 3
      PASS = 1234
150   WRITE(*,*) 'Attempts left:', TRIES
      WRITE(*,*) 'Enter password:'
      READ(*,*) INPUT
      IF (INPUT .EQ. PASS) THEN
          WRITE(*,*) 'Access granted!'
          STOP
      END IF
      TRIES = TRIES - 1
      IF (TRIES .GT. 0) GOTO 150
      WRITE(*,*) 'Account locked!'
      STOP
      END
```

**Features:** - 3 attempt counter - `GOTO` for loop control - Checks password match

### 3.5.1  Challenge Problem: Prime Checker

```
C     CHECKS PRIME NUMBERS
      PROGRAM PRIME
      INTEGER N, I
      LOGICAL ISPRIME
      ISPRIME = .TRUE.
      WRITE(*,*) 'Enter number:'
      READ(*,*) N
C     CHECK DIVISORS UP TO SQRT(N)
      DO 160 I = 2, INT(SQRT(REAL(N)))
          IF (MOD(N, I) .EQ. 0) THEN
              ISPRIME = .FALSE.
```

```
            EXIT
         END IF
160    CONTINUE
      IF (ISPRIME) THEN
         WRITE(*,*) N, 'is prime'
      ELSE
         WRITE(*,*) N, 'is not prime'
      END IF
      STOP
      END
```

**Optimization:** - Loops only up to square root of n - Uses `EXIT` for early termination - `MOD` checks divisibility

# Chapter 4

# Arrays in Fortran 77

## Introduction to Arrays

Arrays allow storage and manipulation of multiple values of the same type. They are essential for handling datasets, matrices, and structured data. Fortran 77 supports static arrays with fixed sizes determined at compile time.

## Declaring Arrays

### One-Dimensional Arrays

```
C     DECLARING 1D ARRAYS
      PROGRAM ARRAY_DECLARE
      INTEGER NUMBERS(5)      ! 5-element integer array
      REAL    TEMPS(0:10)     ! 11 elements (0-10)
      LOGICAL FLAGS(3)        ! 3-element logical array
      CHARACTER*10 NAMES(4)   ! 4 strings of 10 chars each

      NUMBERS(1) = 10         ! Access first element
      TEMPS(0) = 23.5         ! Index starts at 0
      STOP
      END
```

### Multi-Dimensional Arrays

```
C     2D ARRAY DECLARATION
      PROGRAM MATRIX_DECLARE
      REAL GRID(3,3)          ! 3x3 matrix
      INTEGER CUBE(2,2,2)     ! 2x2x2 3D array

      GRID(2,1) = 4.7         ! Row 2, Column 1
      STOP
```

```
      END
```

# Initializing Arrays

## DATA Statement

```
C     COMPILE-TIME INITIALIZATION
      PROGRAM DATA_INIT
      INTEGER MARKS(5)
      DATA MARKS /85, 90, 78, 92, 88/

      REAL MATRIX(2,2)
      DATA MATRIX /1.0, 2.0, 3.0, 4.0/ ! Column-wise filling
      STOP
      END
```

## Runtime Initialization

```
C     LOOP INITIALIZATION
      PROGRAM LOOP_INIT
      REAL SQUARES(10)
      INTEGER I

      DO 10 I = 1, 10
          SQUARES(I) = I**2
10    CONTINUE
      STOP
      END
```

# Accessing Array Elements

```
C     MATRIX SUMMATION EXAMPLE
      PROGRAM MAT_SUM
      REAL A(3,3), TOTAL
      INTEGER I, J

C     Initialize matrix
      DO 20 I = 1, 3
          DO 30 J = 1, 3
              A(I,J) = I + J
30        CONTINUE
20    CONTINUE

C     Calculate sum
      TOTAL = 0.0
```

```
      DO 40 I = 1, 3
          DO 50 J = 1, 3
              TOTAL = TOTAL + A(I,J)
50        CONTINUE
40    CONTINUE
      WRITE(*,*) 'Total sum:', TOTAL
      STOP
      END
```

# 4.1 Passing Arrays to Subprograms

## 4.1.1 Main Program

```
      PROGRAM MAIN
      INTEGER ARR(5)
      DATA ARR /1,2,3,4,5/
      CALL PRINT_ARRAY(ARR, 5)
      STOP
      END
```

## 4.1.2 Subroutine

```
C     ADJUSTABLE ARRAY IN SUBROUTINE
      SUBROUTINE PRINT_ARRAY(A, N)
      INTEGER N, A(N)
      INTEGER I

      DO 60 I = 1, N
          WRITE(*,*) 'Element', I, '=', A(I)
60    CONTINUE
      RETURN
      END
```

# 4.2 Array Operations

## Element-wise Operations

```
C     VECTOR ADDITION
      PROGRAM VEC_ADD
      REAL V1(5), V2(5), RESULT(5)
      INTEGER I

C     Initialize vectors
      DO 70 I = 1, 5
          V1(I) = I
          V2(I) = I*2
```

```
70    CONTINUE

C     Perform addition
      DO 80 I = 1, 5
          RESULT(I) = V1(I) + V2(I)
80    CONTINUE
      STOP
      END
```

## 4.3   Common Pitfalls

- **Out-of-Bounds Access:**

```
INTEGER ARR(5)
ARR(6) = 10  ! Undefined behavior
```

- **Column-Major Order:**

```
REAL MAT(100,100)
! More efficient:
DO 100 J = 1, 100   ! Columns outer loop
    DO 200 I = 1, 100
        MAT(I,J) = ...
200 CONTINUE
100 CONTINUE
```

- **Size Mismatch:**

```
CALL SUB(ARR(5)) when SUB expects ARR(10)
```

## 4.4   Best Practices

- Use PARAMETER for array sizes:

```
INTEGER, PARAMETER :: SIZE = 100
REAL DATA(SIZE)
```

- Initialize arrays explicitly

- Comment array dimensions and purposes

- Prefer column-wise iteration for matrices

# Advanced Example: Matrix Multiplication

```
C     MATRIX MULTIPLICATION
      PROGRAM MAT_MUL
      REAL A(2,2), B(2,2), C(2,2)
      INTEGER I, J, K

C     Initialize matrices
      DATA A /1.0, 2.0, 3.0, 4.0/
      DATA B /5.0, 6.0, 7.0, 8.0/

C     Perform multiplication
      DO 300 I = 1, 2
          DO 400 J = 1, 2
              C(I,J) = 0.0
              DO 500 K = 1, 2
                  C(I,J) = C(I,J) + A(I,K)*B(K,J)
500           CONTINUE
400       CONTINUE
300   CONTINUE

C     Print result
      WRITE(*,*) 'Product matrix:'
      DO 600 I = 1, 2
          WRITE(*,*) C(I,1), C(I,2)
600   CONTINUE
      STOP
      END
```