Fortran Hawatri

Kia Hawatri

March 18, 2025

Contents

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 (2×4-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
```

```
      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

```
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

```
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
```

4.5 Array and Matrix Declaration & Access in Fortran 77

1. Fundamental Array Types

Fortran 77 supports several array declaration styles, each with specific use cases:

Explicit-Shape Arrays

- **Purpose**: Fixed-size arrays with compile-time dimensions
- **Syntax**:

  ```
  DATA_TYPE NAME(LOWER:UPPER, ...)
  ```

- **Key Features**: - Dimensions specified with explicit bounds - Most common array type - Memory allocated at program start

```
C 1D: 5 elements (indices 1-5)
```

```
INTEGER SCORES(5)

C 2D: 3x4 matrix (rows 1-3, cols 1-4)
REAL TEMPERATURES(3,4)

C Custom bounds: indices 0-10 (11 elements)
CHARACTER*20 NAMES(0:10)
```

Adjustable Arrays

- **Purpose**: Pass array sections to subprograms
- **Syntax**:

  ```
  DATA_TYPE NAME(*)
  ```

- **Key Features**: - Used in subprogram parameter lists - Size determined by calling program - Requires explicit interface in some cases

```
SUBROUTINE PROCESS(VECTOR, N)
INTEGER N, VECTOR(N)  ! Adjustable size
...
END
```

Assumed-Size Arrays

- **Purpose**: Handle arrays of unknown size
- **Syntax**:

  ```
  DATA_TYPE NAME(*)
  ```

- **Key Features**: - Last dimension can be asterisk - Limited to subprogram parameters - Avoid for complex operations

```
SUBROUTINE PRINT_ARRAY(ARR, SIZE)
REAL ARR(*)  ! Assumed-size array
...
END
```

2. Matrix Declaration Techniques

Row vs Column Major Order

Fortran uses **column-major** storage:

- Elements stored column-wise in memory
- Critical for performance optimization
- Affects loop nesting order

```
REAL MATRIX(3,3)  ! Stored as:
! (1,1), (2,1), (3,1), (1,2), (2,2), ...
```

Multi-Dimensional Arrays

- Created by specifying multiple dimensions
- Maximum 7 dimensions (per standard)
- Higher dimensions less common

```
C 3D: 2x3x4 array
INTEGER CUBE(2,3,4)

C 4D: Time-varying 3D data
REAL SPACETIME(10,10,10,100)
```

3. Array Access Methods

Element Access

- Use `( )` with comma-separated indices
- Indices must be within declared bounds
- No automatic bounds checking

```
REAL GRID(5,5)
GRID(2,3) = 4.5  ! Single element
```

Section Access (Subarrays)

- Access contiguous array portions
- Limited to *start:end:step* syntax
- Fortran 77 requires explicit loops

```
INTEGER ARR(10), SUB(5)
DO 10 I = 1,5
    SUB(I) = ARR(I+2)  ! Elements 3-7
10 CONTINUE
```

4. Special Array Cases

Zero-Based Arrays

- Not required to start at 1
- Useful for mathematical indices

```
REAL WAVE(-100:100)  ! 201 elements
WAVE(-100) = 0.0     ! First element
```

Character Arrays

- Arrays of fixed-length strings
- Different from character arrays in C

```
CHARACTER*15 NAMES(50)  ! 50 names, 15 chars each
NAMES(1)(1:5) = 'John '  ! Access substring
```

5. Array Usage in Subprograms

Passing Full Arrays

- Pass array name without indices
- Actual and dummy arrays must match rank

```
CALL PRINT_MATRIX(MATRIX)  ! Main program

SUBROUTINE PRINT_MATRIX(ARR)
REAL ARR(3,3)  ! Must match dimensions
...
END
```

Common Pitfalls

- **Dimension Mismatch**:

  ```
  REAL A(5)
  CALL SUB(A(2))  ! Passing single element
  ```

- **Assumed-Size Limitations**:

  ```
  SUBROUTINE BAD(ARR)
  REAL ARR(*)
  PRINT *, SIZE(ARR)  ! Undefined!
  END
  ```

6. Best Practices

- **Use PARAMETER Constants**:

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

- **Initialize Explicitly**:

  ```
  REAL VECTOR(5)
  DATA VECTOR /5*0.0/  ! Initialize to zero
  ```

- **Column-Major Optimization**:

```
DO 20 J = 1, COLS    ! Outer loop columns
    DO 30 I = 1, ROWS
        MATRIX(I,J) = ...
30  CONTINUE
20 CONTINUE
```

7. Comprehensive Example

```
C MATRIX-VECTOR MULTIPLICATION
      PROGRAM MATVEC
      REAL MAT(3,3), VEC(3), RESULT(3)
      INTEGER I,J

C Initialize matrix (column-wise)
      DATA MAT /1.0, 4.0, 7.0,   ! First column
     *          2.0, 5.0, 8.0,   ! Second column
     *          3.0, 6.0, 9.0/    ! Third column

C Initialize vector
      DATA VEC /1.0, 2.0, 3.0/

C Perform multiplication
      DO 40 I = 1, 3
          RESULT(I) = 0.0
          DO 50 J = 1, 3
              RESULT(I) = RESULT(I) + MAT(I,J)*VEC(J)
50        CONTINUE
40    CONTINUE

      WRITE(*,*) 'Result:', RESULT
      STOP
      END
```

Key Observations: 1. Matrix initialized column-wise via DATA 2. Nested loops follow column-major order 3. Explicit element-by-element calculation 4. RESULT array stores final values

4.6 Exercises: Loops in Fortran 77

Basic Loop Structures

1. **Simple DO Loop**: Write a program to print numbers from 1 to a user-input integer N using a DO loop. *Input*: 5 — *Output*: 1 2 3 4 5

2. **Step Value Practice**: Modify the above program to print even numbers between 2 and N using a step value of 2. *Input*: 10 — *Output*: 2 4 6 8 10

3. **Summation Loop**: Calculate the sum of the first N natural numbers using a DO loop. *Input*: 5 — *Output*: 15

4. **Factorial Calculator**: Compute $N!$ (factorial) using a DO loop. Handle $N = 0$ as a special case. *Input*: 4 — *Output*: 24

Conditional Loops & Control Flow

5. **Input Validation**: Use a DO-WHILE loop to repeatedly ask for a positive integer until valid input is received.

6. **Early Exit**: Read numbers until a negative value is entered. Print the sum of positive numbers using a loop with a conditional 'GOTO' exit. *Input*: 3 5 -1 — *Output*: 8

7. **Prime Checker**: Check if a number is prime using a loop. Terminate early if a divisor is found. *Input*: 7 — *Output*: "Prime"

8. **Password Attempts**: Implement 3 login attempts using a loop. Exit early if the correct password is entered.

Nested Loops

9. **Multiplication Table**: Print a $N \times N$ multiplication table using nested loops. *Input*: 3 — *Output*: 1 2 3 2 4 6 3 6 9

10. **Pattern Printing**: Use nested loops to print:

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

11. **Matrix Initialization**: Initialize a 3×3 matrix with values $A(i, j) = i + j$ using nested loops.

Array Operations with Loops

12. **Array Sum & Average**: Read 5 numbers into an array. Compute their sum and average using a loop.

13. **Maximum Element**: Find the largest value in a 1D array of 10 elements using a loop.

14. **Array Reversal**: Reverse the elements of a 1D array using loops. *Input*: [1, 2, 3] — *Output*: [3, 2, 1]

15. **Matrix Transpose**: Transpose a 3×3 matrix using nested loops. *Input*: 1 2 3 4 5 6 7 8 9 *Output*: 1 4 7 2 5 8 3 6 9

Challenge Problem

Bubble Sort: Sort a 1D array of 10 integers in ascending order using nested loops.

4.7 Exercise Answers: Loops in Fortran 77

1. Simple DO Loop

```
C     PRINTS NUMBERS 1 TO N
```

```
      PROGRAM COUNT
      INTEGER N, I
      WRITE(*,*) 'ENTER N:'
      READ(*,*) N
      DO 10 I = 1, N
          WRITE(*,*) I
10    CONTINUE
      STOP
      END
```

Explanation: - Loop variable `I` runs from 1 to user-input `N` - `CONTINUE` at label 10 marks loop end - Implicit increment of 1 per iteration

2. Even Numbers with Step Value

```
C     PRINTS EVEN NUMBERS UPTO N
      PROGRAM EVENS
      INTEGER N, I
      WRITE(*,*) 'ENTER N:'
      READ(*,*) N
      DO 20 I = 2, N, 2
          WRITE(*,*) I
20    CONTINUE
      STOP
      END
```

Key Features: - Step value 2 creates even sequence - Handles any even/odd `N` correctly - Loop bounds inclusive

3. Sum of First N Natural Numbers

```
C     CALCULATES SUM(1-N)
      PROGRAM SUMN
      INTEGER N, I, TOTAL
      TOTAL = 0
      WRITE(*,*) 'ENTER N:'
      READ(*,*) N
      DO 30 I = 1, N
          TOTAL = TOTAL + I
30    CONTINUE
      WRITE(*,*) 'SUM:', TOTAL
      STOP
      END
```

Logic: - Initializes `TOTAL` to 0 - Accumulates sum in loop - Works for N 0

4. Factorial Calculator

```
C     COMPUTES N!
      PROGRAM FACT
      INTEGER N, I, RESULT
      RESULT = 1
      WRITE(*,*) 'ENTER N:'
      READ(*,*) N
      IF (N .EQ. 0) GOTO 40
      DO 50 I = 1, N
          RESULT = RESULT * I
50    CONTINUE
40    WRITE(*,*) N, '! =', RESULT
      STOP
      END
```

Special Case: - Handles 0! = 1 via `GOTO` - Loop multiplies sequentially - Result initialized to 1

5. Input Validation

```
C     ENSURES POSITIVE INPUT
      PROGRAM VALID
      INTEGER NUM
60    WRITE(*,*) 'ENTER POSITIVE NUMBER:'
      READ(*,*) NUM
      IF (NUM .LE. 0) GOTO 60
      WRITE(*,*) 'VALID INPUT:', NUM
      STOP
      END
```

Features: - Infinite loop until valid input - `GOTO` creates repetition - Strict positive check

6. Early Exit Summation

```
C     SUMS POSITIVE NUMBERS
      PROGRAM SUM_POS
      INTEGER NUM, TOTAL
      TOTAL = 0
70    WRITE(*,*) 'ENTER NUMBER:'
      READ(*,*) NUM
      IF (NUM .LT. 0) GOTO 80
      TOTAL = TOTAL + NUM
      GOTO 70
80    WRITE(*,*) 'TOTAL:', TOTAL
      STOP
      END
```

Control Flow: - Loop exits on negative input - Accumulates in `TOTAL` - `GOTO` creates loop structure

7. Prime Number Check

```
C     CHECKS PRIME STATUS
      PROGRAM PRIME
      INTEGER N, I
      LOGICAL ISPRIME
      ISPRIME = .TRUE.
      WRITE(*,*) 'ENTER NUMBER:'
      READ(*,*) N
      DO 90 I = 2, INT(SQRT(REAL(N)))
          IF (MOD(N, I) .EQ. 0) THEN
              ISPRIME = .FALSE.
              GOTO 100
          END IF
90    CONTINUE
100   IF (ISPRIME) THEN
          WRITE(*,*) 'PRIME'
      ELSE
          WRITE(*,*) 'NOT PRIME'
      END IF
      STOP
      END
```

Optimization: - Loops up to n - Early exit using `GOTO` - Handles 1 and 2 correctly

8. Password Attempt System

```
C     PASSWORD CHECKER
      PROGRAM PASS
      INTEGER TRIES, PASS, INPUT
      TRIES = 3
      PASS = 1234
110   WRITE(*,*) 'ATTEMPTS LEFT:', TRIES
      WRITE(*,*) 'ENTER PASSWORD:'
      READ(*,*) INPUT
      IF (INPUT .EQ. PASS) THEN
          WRITE(*,*) 'ACCESS GRANTED'
          STOP
      ELSE
          TRIES = TRIES - 1
          IF (TRIES .GT. 0) GOTO 110
      END IF
      WRITE(*,*) 'ACCOUNT LOCKED'
      STOP
      END
```

Security: - 3 attempt counter - Decrements on wrong attempts - Immediate exit on correct password

9. Multiplication Table

```
C     N x N MULTIPLICATION TABLE
      PROGRAM MUL_TABLE
      INTEGER N, I, J
      WRITE(*,*) 'ENTER N:'
      READ(*,*) N
      DO 120 I = 1, N
          DO 130 J = 1, N
              WRITE(*,'(I4)') I*J
130       CONTINUE
          WRITE(*,*) ' '  ! NEWLINE
120   CONTINUE
      STOP
      END
```

Formatting: - Nested loops for rows/columns - Formatted output for alignment - Newline after each row

10. Star Pattern

```
C     PRINTS TRIANGLE PATTERN
      PROGRAM STAR
      INTEGER I, J
      DO 140 I = 1, 3
          DO 150 J = 1, I
              WRITE(*,*) '*'
150       CONTINUE
          WRITE(*,*) ' '  ! NEWLINE
140   CONTINUE
      STOP
      END
```

Pattern Logic: - Outer loop controls rows - Inner loop prints stars per row - Newline after each row

11. Matrix Initialization

```
C     INITIALIZES 3x3 MATRIX
      PROGRAM MAT_INIT
      INTEGER MAT(3,3), I, J
      DO 160 I = 1, 3
          DO 170 J = 1, 3
              MAT(I,J) = I + J
170       CONTINUE
160   CONTINUE
      STOP
      END
```

Structure: - Nested loops for rows/columns - Formula: `I + J` - 3x3 matrix dimensions

12. Array Sum & Average

```
C     ARRAY OPERATIONS
      PROGRAM ARR_OPS
      REAL ARR(5), SUM, AVG
      INTEGER I
      WRITE(*,*) 'ENTER 5 NUMBERS:'
      READ(*,*) (ARR(I), I=1,5)
      SUM = 0.0
      DO 180 I = 1, 5
          SUM = SUM + ARR(I)
180   CONTINUE
      AVG = SUM / 5.0
      WRITE(*,*) 'SUM:', SUM, 'AVG:', AVG
      STOP
      END
```

Array Handling: - Implied DO loop for input - Accumulates sum in loop - Explicit type conversion for average

13. Maximum Element

```
C     FINDS LARGEST ARRAY ELEMENT
      PROGRAM MAXVAL
      REAL ARR(10), MAX
      INTEGER I
      READ(*,*) ARR
      MAX = ARR(1)
      DO 190 I = 2, 10
          IF (ARR(I) .GT. MAX) MAX = ARR(I)
190   CONTINUE
      WRITE(*,*) 'MAXIMUM:', MAX
      STOP
      END
```

Algorithm: - Initialize max to first element - Linear scan through array - Updates max when larger found

14. Array Reversal

```
C     REVERSES ARRAY IN-PLACE
      PROGRAM REVERSE
      INTEGER ARR(5), TEMP, I
      DATA ARR /1,2,3,4,5/
      DO 200 I = 1, 2
```

```
          TEMP = ARR(I)
          ARR(I) = ARR(6-I)
          ARR(6-I) = TEMP
200   CONTINUE
      WRITE(*,*) ARR
      STOP
      END
```

Swap Logic: - Swaps elements from ends to center - Loop runs halfway (N/2 iterations) - Temporary variable for swap

15. Matrix Transpose

```
C     TRANSPOSES 3x3 MATRIX
      PROGRAM TRANSPOSE
      INTEGER A(3,3), B(3,3), I, J
      READ(*,*) ((A(I,J), J=1,3), I=1,3)
      DO 210 I = 1, 3
          DO 220 J = 1, 3
              B(J,I) = A(I,J)
220       CONTINUE
210   CONTINUE
      WRITE(*,*) 'TRANSPOSE:'
      WRITE(*,*) ((B(I,J), J=1,3), I=1,3)
      STOP
      END
```

Transposition: - Creates new matrix `B` - Swaps row/column indices - Nested input/output loops

Challenge: Bubble Sort

```
C     SORTS ARRAY IN ASCENDING ORDER
      PROGRAM BUBBLE
      INTEGER ARR(10), I, J, TEMP
      LOGICAL SWAPPED
      READ(*,*) ARR
      DO 230 I = 9, 1, -1
          SWAPPED = .FALSE.
          DO 240 J = 1, I
              IF (ARR(J) .GT. ARR(J+1)) THEN
                  TEMP = ARR(J)
                  ARR(J) = ARR(J+1)
                  ARR(J+1) = TEMP
                  SWAPPED = .TRUE.
              END IF
240       CONTINUE
          IF (.NOT. SWAPPED) GOTO 250
```

```
230   CONTINUE
250   WRITE(*,*) 'SORTED ARRAY:', ARR
      STOP
      END
```

Optimization: - Early exit if no swaps - Outer loop reduces range - In-place sorting

Chapter 5

Functions in Fortran 77

Introduction to Functions

Functions in Fortran 77 are subprograms that:

- Return a single value
- Can accept input arguments
- Improve code modularity and reusability
- Are categorized as:
  - Intrinsic (built-in)
  - External (user-defined)
  - Statement (single-expression)

1. Intrinsic Functions

Predefined by the language:

```
C     EXAMPLE OF INTRINSIC FUNCTIONS
      PROGRAM INTRINSIC
      REAL X, Y
      X = 2.5
      Y = SQRT(X)      ! Square root
      WRITE(*,*) SIN(X), EXP(Y)  ! Sine and exponential
      STOP
      END
```

2. External Functions

User-defined functions in separate program units:

Function Definition

```
C     FUNCTION TO CALCULATE AREA OF CIRCLE
      REAL FUNCTION AREA(R)
      REAL R, PI
      PARAMETER (PI = 3.14159)
      AREA = PI * R**2
      RETURN
      END
```

Function Usage

```
C     MAIN PROGRAM
      PROGRAM MAIN
      REAL RADIUS, AREA
      WRITE(*,*) 'ENTER RADIUS:'
      READ(*,*) RADIUS
      WRITE(*,*) 'AREA:', AREA(RADIUS)
      STOP
      END
```

3. Statement Functions

Single-line functions defined in declaration section:

```
C     SIMPLE STATEMENT FUNCTION
      PROGRAM STMT
      REAL X, Y, AVG
      AVG(A,B) = (A + B)/2.0  ! Statement function

      X = 5.0
      Y = 7.0
      WRITE(*,*) 'AVERAGE:', AVG(X,Y)
      STOP
      END
```

Function Declaration Rules

- Return type declared in function definition

  ```
  REAL FUNCTION NAME(...)
  ```

- Must be declared in calling program if:
  - Return type doesn't match implicit naming
  - Function is external
- Arguments passed by reference

Argument Passing Example

```
C     FUNCTION WITH MULTIPLE PARAMETERS
      REAL FUNCTION POWER(BASE, EXP)
      REAL BASE
      INTEGER EXP
      POWER = BASE**EXP
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL POWER, RESULT
      RESULT = POWER(2.5, 3)
      WRITE(*,*) '2.5^3 =', RESULT
      STOP
      END
```

4. Type Declaration in Calling Program

```
C     EXPLICIT TYPE DECLARATION
      PROGRAM TYPE_DEC
      REAL VOLUME  ! Function returns REAL
      WRITE(*,*) 'VOLUME:', VOLUME(5.0)
      STOP
      END

      REAL FUNCTION VOLUME(R)
      REAL R
      VOLUME = (4.0/3.0) * 3.14159 * R**3
      RETURN
      END
```

5. Common Function Errors

- **Implicit Type Mismatch**:

  ```
  FUNCTION TEST()  ! Implicit REAL
  ...
  INTEGER TEST    ! Conflict in calling program
  ```

- **Missing Declaration**:

  ```
  C     MAIN PROGRAM
        PROGRAM ERR
        WRITE(*,*) FUNC(2)  ! FUNC not declared
        STOP
  ```

```
      END

      INTEGER FUNCTION FUNC(X)
...
```

- **Argument Count Mismatch**:

```
CALL AREA(5.0, RESULT)  ! AREA expects 1 argument
```

6. Functions vs Subroutines

Functions	Subroutines
Return one value	No return value
Used in expressions	Called with CALL
Can't modify arguments	Can modify arguments

7. Best Practices

- Always declare function return types explicitly
- Use meaningful function names
- Document argument types and purposes
- Avoid modifying input arguments
- Use statement functions only for simple operations

8. Advanced Example

```
C     RECURSIVE FACTORIAL (SIMULATED)
      PROGRAM RECUR
      INTEGER N, FACT
      WRITE(*,*) 'ENTER NUMBER:'
      READ(*,*) N
      WRITE(*,*) N, '! =', FACT(N)
      STOP
      END

      INTEGER FUNCTION FACT(K)
      INTEGER K
      IF (K .LE. 1) THEN
          FACT = 1
      ELSE
          FACT = K * FACT(K-1)
      END IF
      RETURN
      END
```

Note: Fortran 77 doesn't officially support recursion - this may require compiler-specific settings.

9. Function Libraries

Group related functions into files:

```
C     MATH_OPERATIONS.F
      REAL FUNCTION AREA(R)
      ...
      END

      REAL FUNCTION VOLUME(R)
      ...
      END
```

Include in main program:

```
      PROGRAM GEOM
      REAL AREA, VOLUME
      ...
      END
```

5.1 Implicit vs. Explicit Functions in Fortran 77

1. Implicit Functions

Functions that rely on Fortran's default typing rules, where:

- Function type is determined by first letter of name
- I-N: INTEGER (default)
- A-H, O-Z: REAL (default)
- No explicit type declaration required

Example 1: Implicit Real Function

```
C     IMPLICIT REAL FUNCTION (NAME STARTS WITH 'A')
      FUNCTION AVG(X, Y)
      AVG = (X + Y) / 2.0
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      WRITE(*,*) 'AVERAGE:', AVG(5.0, 7.0)
      STOP
      END
```

Behavior: - Function name 'AVG' starts with A $\rightarrow$ REAL - No type declaration in function definition - Works but prone to errors

Example 2: Implicit Integer Function

```
C     IMPLICIT INTEGER FUNCTION (NAME STARTS WITH 'I')
      FUNCTION ICOUNT(X)
      ICOUNT = INT(X) + 5
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      WRITE(*,*) 'COUNT:', ICOUNT(3.7)  ! Output: 8
      STOP
      END
```

Risk: - Return type inferred from name - Easy to create type mismatches

2. Explicit Functions

Functions with declared return types:

- Type specified in function definition
- Must be declared in calling program
- Recommended for code clarity

Example 1: Explicit Real Function

```
C     EXPLICIT TYPE DECLARATION
      REAL FUNCTION AREA(R)
      REAL R, PI
      PARAMETER (PI = 3.14159)
      AREA = PI * R**2
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL AREA  ! MUST DECLARE IN CALLING UNIT
      WRITE(*,*) 'AREA:', AREA(2.5)
      STOP
      END
```

Advantages: - Clear return type declaration - Compiler checks type consistency - Avoids naming conflicts

Example 2: Explicit Integer Function

```
C     EXPLICIT INTEGER FUNCTION
      INTEGER FUNCTION FACT(N)
      INTEGER N, I
      FACT = 1
      DO 10 I = 1, N
          FACT = FACT * I
10    CONTINUE
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      INTEGER FACT  ! REQUIRED DECLARATION
      WRITE(*,*) '5! =', FACT(5)
      STOP
      END
```

3. Key Differences

Feature	Implicit	Explicit
Declaration	Name-based	Explicit
Type Safety	Low	High
Readability	Poor	Good
Error Checking	Limited	Strict
Legacy Code	Common	Rare

4. Common Pitfalls with Implicit Functions

Type Mismatch Example

```
C     DANGEROUS IMPLICIT CONVERSION
      FUNCTION TOTAL(X, Y)
      TOTAL = X + Y  ! Implicit REAL return
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      INTEGER TOTAL  ! WRONG TYPE DECLARATION
      WRITE(*,*) TOTAL(2, 3)  ! Output: 0 (incorrect)
      STOP
      END
```

Result: - Function returns REAL but main program expects INTEGER - Undefined behavior occurs

Fixing with Explicit Declaration

```
      REAL FUNCTION TOTAL(X, Y)
      INTEGER X, Y
      TOTAL = REAL(X) + REAL(Y)
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL TOTAL  ! CORRECT DECLARATION
      WRITE(*,*) TOTAL(2, 3)  ! Output: 5.0
      STOP
      END
```

5. Best Practices

- **Always Use Explicit Functions**:

```
REAL FUNCTION NAME(...)  ! Preferred
```

- **Use IMPLICIT NONE**:

```
PROGRAM MAIN
IMPLICIT NONE  ! Disables default typing
REAL :: VALUE
...
END
```

- **Declare Functions in Calling Units**:

```
PROGRAM MAIN
REAL EXTERNAL_FUNC  ! Declaration
...
END
```

- **Document Function Interfaces**:

```
C     FUNCTION: CALCULATE_VELOCITY
C     INPUT: MASS (REAL), FORCE (REAL)
C     OUTPUT: VELOCITY (REAL)
      REAL FUNCTION VELOCITY(MASS, FORCE)
      ...
```

6. Advanced Example: Type Conversion

```
C     EXPLICIT TYPE CONVERSION FUNCTION
      CHARACTER*20 FUNCTION STR(NUM)
      REAL NUM
      WRITE(STR, '(F10.2)') NUM
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      CHARACTER*20 STR
      WRITE(*,*) 'FORMATTED:', STR(123.456)
      STOP
      END
```

Output: `123.46`

7. Function Type Declaration Table

Declaration	Return Type	Example
REAL FUNCTION	Single-precision	REAL FUNC()
DOUBLE PRECISION	Double-precision	DOUBLE PRECISION DFUNC()
INTEGER FUNCTION	Integer	INTEGER IFUNC()
LOGICAL FUNCTION	Boolean	LOGICAL TEST()
CHARACTER*N	String	CHARACTER*10 CFUNC()

8. Conversion Checklist

When converting implicit to explicit:

1. Add explicit type declaration to function
2. Declare function in all calling units
3. Check argument types match
4. Use IMPLICIT NONE to catch errors
5. Test with edge cases

5.2 More Functions vs. Subroutines in Fortran 77

1. Fundamental Definitions

- **Function**: - Returns a single value - Invoked within expressions - Typically used for calculations - Example: `SQRT(X)`, `SIN(X)`
- **Subroutine**: - Does not return a value directly - Invoked with `CALL` statement - Can modify multiple arguments - Example: `CALL SWAP(A, B)`

2. Key Differences

Feature	Function	Subroutine
Return Value	Single value	None (void)
Invocation	In expressions	With `CALL`
Arguments	Input parameters	Input/Output parameters
Side Effects	Should avoid	Expected
Return Method	Assign to function name	Modify arguments
Multiple Returns	Impossible	Possible via arguments

3. Function Examples

Example 1: Basic Function

```
C     FUNCTION TO CALCULATE AREA
      REAL FUNCTION AREA(R)
      REAL R, PI
      PARAMETER (PI = 3.14159)
      AREA = PI * R**2
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL AREA, RADIUS
      RADIUS = 5.0
      WRITE(*,*) 'AREA:', AREA(RADIUS)
      STOP
      END
```

Example 2: Type-Specific Function

```
C     INTEGER FUNCTION
      INTEGER FUNCTION IFACT(N)
      INTEGER N, I
      IFACT = 1
      DO 10 I = 1, N
          IFACT = IFACT * I
10    CONTINUE
      RETURN
      END
```

4. Subroutine Examples

Example 1: Basic Subroutine

```
C     SUBROUTINE TO SWAP VALUES
      SUBROUTINE SWAP(A, B)
```

```
      REAL A, B, TEMP
      TEMP = A
      A = B
      B = TEMP
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL X, Y
      X = 5.0
      Y = 10.0
      CALL SWAP(X, Y)
      WRITE(*,*) 'X=', X, 'Y=', Y
      STOP
      END
```

Example 2: Multi-Output Subroutine

```
C     CALCULATE STATISTICS
      SUBROUTINE STATS(ARR, N, AVG, MAX)
      REAL ARR(N), AVG, MAX
      INTEGER N, I
      AVG = 0.0
      MAX = ARR(1)
      DO 20 I = 1, N
          AVG = AVG + ARR(I)
          IF (ARR(I) .GT. MAX) MAX = ARR(I)
20    CONTINUE
      AVG = AVG / REAL(N)
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL NUMBERS(5), AVERAGE, MAXIMUM
      DATA NUMBERS /2.0, 5.0, 7.0, 3.0, 1.0/
      CALL STATS(NUMBERS, 5, AVERAGE, MAXIMUM)
      WRITE(*,*) 'AVG:', AVERAGE, 'MAX:', MAXIMUM
      STOP
      END
```

5. Argument Handling Comparison

Function Argument Handling

```
C     FUNCTION WITH INPUT ARGUMENTS
```

```
      REAL FUNCTION POWER(BASE, EXP)
      REAL BASE
      INTEGER EXP
      POWER = BASE ** EXP
      RETURN
      END
```

Note: Functions should not modify input arguments

Subroutine Argument Handling

```
C     SUBROUTINE MODIFYING ARGUMENTS
      SUBROUTINE PROCESS(X, Y, Z)
      REAL X, Y, Z
      X = X * 2
      Y = Y / 2
      Z = X + Y
      RETURN
      END
```

Note: Subroutines frequently modify arguments

6. When to Use Each

- **Use Functions When**: - Need to return a single value - Performing mathematical calculations - Want to use result in expressions - Example: `AREA = CIRCLE_AREA(R)`
- **Use Subroutines When**: - Need to return multiple values - Modifying existing variables - Performing I/O operations - Example: `CALL SORT(ARRAY, N)`

7. Advanced Differences

Memory Management

- Functions: Generally use temporary storage
- Subroutines: Often work directly on arguments

Error Handling

```
C     SUBROUTINE WITH ERROR FLAG
      SUBROUTINE DIVIDE(A, B, RES, ERROR)
      REAL A, B, RES
      LOGICAL ERROR
      ERROR = .FALSE.
      IF (B .EQ. 0.0) THEN
          ERROR = .TRUE.
          RETURN
      END IF
```

```
      RES = A / B
      RETURN
      END
```

8. Common Mistakes

Function Modifying Arguments

```
C     DANGEROUS FUNCTION
      REAL FUNCTION BADFUNC(X)
      REAL X
      X = X * 2  ! Modifying input argument
      BADFUNC = X
      RETURN
      END
```

Risk: Unintended side effects

Using Subroutine as Function

```
C     INCORRECT USAGE
      PROGRAM ERR
      REAL RES
      RES = SUBR()  ! Can't assign subroutine
      STOP
      END

      SUBROUTINE SUBR()
      ...
      END
```

9. Best Practices

- Use functions for pure calculations
- Use subroutines for I/O and multi-value returns
- Always declare function types explicitly
- Document argument intent:

  ```
  C     INPUT: X, OUTPUT: Y, INPUT/OUTPUT: Z
  ```

- Avoid global variables in functions

10. Hybrid Example

```
C     FUNCTION USING SUBROUTINE
      REAL FUNCTION SMART\_CALC(A, B)
      REAL A, B
      CALL PREPROCESS(A, B)
      SMART\_CALC = A ** 2 + B ** 2
      RETURN
      END

      SUBROUTINE PREPROCESS(X, Y)
      REAL X, Y
      X = ABS(X)
      Y = ABS(Y)
      RETURN
      END
```

11. Performance Considerations

- Functions better for inlining small calculations
- Subroutines better for complex operations
- Argument passing overhead similar for both
- Use subroutines for memory-intensive operations

5.3 Functions and Arrays in Fortran 77

1. Passing Arrays to Functions

Fortran 77 allows arrays to be passed to functions and subroutines. Key considerations:

- Arrays are passed by reference (modifications affect the original)
- Size must be declared explicitly or passed as an argument
- Use adjustable arrays with `DIMENSION` or size parameters

Example 1: Sum of Array Elements (Function)

```
C     FUNCTION TO CALCULATE ARRAY SUM
      REAL FUNCTION ARRAY_SUM(ARR, N)
      INTEGER N
      REAL ARR(N)
      INTEGER I
      ARRAY_SUM = 0.0
      DO 10 I = 1, N
          ARRAY_SUM = ARRAY_SUM + ARR(I)
```

```
10    CONTINUE
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      PARAMETER (SIZE = 5)
      REAL NUMBERS(SIZE), ARRAY_SUM
      DATA NUMBERS /1.0, 2.0, 3.0, 4.0, 5.0/
      WRITE(*,*) 'SUM:', ARRAY_SUM(NUMBERS, SIZE)
      STOP
      END
```

2. Multi-Dimensional Arrays

Example 2: Matrix Trace (Function)

```
C     FUNCTION TO CALCULATE MATRIX TRACE
      REAL FUNCTION TRACE(MAT, N)
      INTEGER N
      REAL MAT(N,N)
      INTEGER I
      TRACE = 0.0
      DO 20 I = 1, N
          TRACE = TRACE + MAT(I,I)
20    CONTINUE
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      PARAMETER (N = 3)
      REAL MATRIX(N,N), TRACE
      DATA MATRIX /1.0, 2.0, 3.0,
     *             4.0, 5.0, 6.0,
     *             7.0, 8.0, 9.0/
      WRITE(*,*) 'TRACE:', TRACE(MATRIX, N)
      STOP
      END
```

3. Returning Arrays via Subroutines

While functions cannot return arrays directly, subroutines can modify array arguments:

```
C     SUBROUTINE TO DOUBLE ARRAY ELEMENTS
      SUBROUTINE DOUBLE_ARRAY(ARR, N)
      INTEGER N
```

```
      REAL ARR(N)
      INTEGER I
      DO 30 I = 1, N
          ARR(I) = ARR(I) * 2.0
30    CONTINUE
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      PARAMETER (SIZE = 4)
      REAL DATA(SIZE)
      DATA DATA /1.0, 2.0, 3.0, 4.0/
      CALL DOUBLE_ARRAY(DATA, SIZE)
      WRITE(*,*) 'DOUBLED ARRAY:', DATA
      STOP
      END
```

4. Adjustable Arrays

Use `DIMENSION` for flexible array handling in subprograms:

```
C     FUNCTION TO FIND MAXIMUM VALUE
      REAL FUNCTION ARRAY_MAX(ARR, N)
      INTEGER N
      REAL ARR(N)
      DIMENSION ARR(N)
      INTEGER I
      ARRAY_MAX = ARR(1)
      DO 40 I = 2, N
          IF (ARR(I) .GT. ARRAY_MAX) THEN
              ARRAY_MAX = ARR(I)
          END IF
40    CONTINUE
      RETURN
      END
```

5. Common Operations

Example 3: Dot Product (Function)

```
C     FUNCTION TO CALCULATE DOT PRODUCT
      REAL FUNCTION DOT_PROD(A, B, N)
      INTEGER N
      REAL A(N), B(N)
      INTEGER I
      DOT_PROD = 0.0
```

```
      DO 50 I = 1, N
          DOT_PROD = DOT_PROD + A(I) * B(I)
50    CONTINUE
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      PARAMETER (LEN = 3)
      REAL V1(LEN), V2(LEN), DOT_PROD
      DATA V1 /1.0, 2.0, 3.0/
      DATA V2 /4.0, 5.0, 6.0/
      WRITE(*,*) 'DOT PRODUCT:', DOT_PROD(V1, V2, LEN)
      STOP
      END
```

6. Best Practices

- **Always Pass Array Size**:

```
SUBROUTINE PROCESS(ARR, N)
INTEGER N
REAL ARR(N)
```

- **Use PARAMETER Constants**:

```
PARAMETER (MAX_SIZE = 100)
REAL ARR(MAX_SIZE)
```

- **Avoid Side Effects in Functions**:

```
C GOOD: Pure function
REAL FUNCTION SUM(ARR, N)
C BAD: Function modifying input
REAL FUNCTION BAD(ARR, N)
ARR(1) = 0.0
```

- **Document Array Dimensions**:

```
C     INPUT: ARR(N) - 1D array of N elements
C     OUTPUT: Returns sum of elements
```

7. Common Errors

Mismatched Dimensions

```
C MAIN PROGRAM
REAL MAT(3,3)
C FUNCTION EXPECTS 1D ARRAY
CALL PRINT_ARRAY(MAT)  ! ERROR
```

Incorrect Bounds

```
DO 60 I = 1, N+1  ! N is array size
    ARR(I) = 0.0  ! OUT OF BOUNDS
60 CONTINUE
```

8. Advanced Example: Matrix Multiplication

```
C     SUBROUTINE FOR MATRIX MULTIPLICATION
      SUBROUTINE MAT_MUL(A, B, C, N)
      INTEGER N
      REAL A(N,N), B(N,N), C(N,N)
      INTEGER I, J, K
      DO 70 I = 1, N
          DO 80 J = 1, N
              C(I,J) = 0.0
              DO 90 K = 1, N
                  C(I,J) = C(I,J) + A(I,K) * B(K,J)
90            CONTINUE
80        CONTINUE
70    CONTINUE
      RETURN
      END
```

9. Handling Character Arrays

```
C     SUBROUTINE TO REVERSE STRING
      SUBROUTINE REVERSE_STR(STR, LEN)
      INTEGER LEN
      CHARACTER*(*) STR
      CHARACTER TEMP
      INTEGER I
      DO 100 I = 1, LEN/2
          TEMP = STR(I:I)
          STR(I:I) = STR(LEN-I+1:LEN-I+1)
          STR(LEN-I+1:LEN-I+1) = TEMP
100   CONTINUE
      RETURN
      END
```

5.4 Functions Calling Functions in Fortran 77

1. Basic Function Composition

Fortran 77 allows functions to call other functions, but with important constraints:

- Functions must be declared in the calling program unit

- No direct support for recursion (without compiler extensions)
- Functions can be nested up to compiler-dependent limits
- Proper type declarations are critical

Example 1: Simple Function Composition

```
C     FUNCTION TO CALCULATE SQUARE
      REAL FUNCTION SQUARE(X)
      REAL X
      SQUARE = X * X
      RETURN
      END

C     FUNCTION TO CALCULATE HYPOTENUSE
      REAL FUNCTION HYPOT(A, B)
      REAL A, B, SQUARE
      HYPOT = SQRT(SQUARE(A) + SQUARE(B))
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL HYPOT
      WRITE(*,*) 'HYPOTENUSE:', HYPOT(3.0, 4.0)
      STOP
      END
```

Explanation: - `HYPOT` calls `SQUARE` twice - `SQRT` is an intrinsic function - All functions must be declared in calling scope

2. Passing Functions as Arguments

Fortran 77 supports function arguments using the `EXTERNAL` keyword:

```
C     FUNCTION INTEGRATOR
      REAL FUNCTION INTEGRAL(FUNC, A, B, N)
      EXTERNAL FUNC
      REAL FUNC, A, B
      INTEGER N
      REAL DX, X, SUM
      DX = (B - A)/N
      SUM = 0.0
      DO 10 I = 1, N
          X = A + (I-0.5)*DX
          SUM = SUM + FUNC(X)
10    CONTINUE
```

```
      INTEGRAL = SUM * DX
      RETURN
      END

C     FUNCTION TO INTEGRATE
      REAL FUNCTION POLY(X)
      REAL X
      POLY = X**3 + 2*X + 5
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      EXTERNAL POLY
      REAL INTEGRAL, RESULT
      RESULT = INTEGRAL(POLY, 0.0, 2.0, 1000)
      WRITE(*,*) 'INTEGRAL:', RESULT
      STOP
      END
```

3. Recursion Limitations

Standard Fortran 77 does not support recursion. Some compilers allow it with flags:

```
C     COMPILER-DEPENDENT RECURSION (GNU)
      RECURSIVE INTEGER FUNCTION FACT(N) RESULT(RES)
      INTEGER N
      IF (N <= 1) THEN
          RES = 1
      ELSE
          RES = N * FACT(N-1)
      END IF
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      WRITE(*,*) '5! =', FACT(5)
      STOP
      END
```

Note: Not standard Fortran 77! Requires compiler extensions.

4. Function Libraries

Organize related functions in separate files:

```
C     MATH_FUNCS.F
      REAL FUNCTION MEAN(ARR, N)
```

```
      REAL ARR(N)
      INTEGER N
      MEAN = SUM(ARR) / N
      END

      REAL FUNCTION STDDEV(ARR, N)
      REAL ARR(N), MEAN
      STDDEV = SQRT(SUM((ARR - MEAN(ARR,N))**2)/(N-1))
      END
```

5. Common Patterns

Wrapper Functions

```
C     WRAPPER FOR DIFFERENT PRECISION
      DOUBLE PRECISION FUNCTION DEXP(X)
      DOUBLE PRECISION X
      DEXP = EXP(REAL(X))  ! Calls intrinsic EXP
      RETURN
      END
```

Callback Systems

```
C     ROOT FINDING USING CALLBACK
      REAL FUNCTION FIND_ROOT(FUNC, GUESS)
      EXTERNAL FUNC
      REAL FUNC, GUESS
      REAL X, F_X, DX
      X = GUESS
      DX = 0.001
      DO 20 I = 1, 1000
          F_X = FUNC(X)
          IF (ABS(F_X) < 1E-6) EXIT
          X = X - F_X/((FUNC(X+DX)-F_X)/DX)
20    CONTINUE
      FIND_ROOT = X
      RETURN
      END
```

6. Best Practices

- Declare all functions with `EXTERNAL` when passing as arguments
- Use explicit interfaces for complex interactions
- Avoid deep nesting (max 2-3 levels)
- Document function dependencies

- Test compiler compatibility for advanced features

7. Common Errors

Missing EXTERNAL Declaration

```
PROGRAM MAIN
REAL RESULT
RESULT = INTEGRAL(POLY, 0, 2, 100)  ! Undefined POLY
STOP
END
```

Type Mismatch

```
REAL FUNCTION F(X)
INTEGER X  ! Should be REAL
F = X**2
END
```

8. Performance Considerations

Technique	Impact
Function inlining	Faster execution
Deep nesting	Increased stack usage
Function pointers	Slower dispatch
Recursion	Memory intensive

9. Advanced Example: Function Factory

```
C     CREATE POWER FUNCTIONS DYNAMICALLY
      FUNCTION POWER_GEN(EXPONENT)
      REAL POWER_GEN
      REAL EXPONENT
      EXTERNAL POWER_FUNC
      POWER_GEN = POWER_FUNC
      RETURN
      END

      REAL FUNCTION POWER_FUNC(X, EXPONENT)
      REAL X, EXPONENT
      POWER_FUNC = X ** EXPONENT
      RETURN
      END

C     MAIN PROGRAM (PSEUDO-CODE)
      PROGRAM MAIN
      EXTERNAL POWER_GEN
```

```
      REAL SQUARE, CUBE
      SQUARE = POWER_GEN(2.0)
      CUBE = POWER_GEN(3.0)
      WRITE(*,*) SQUARE(5.0), CUBE(5.0)
      STOP
      END
```

Note: Requires advanced techniques beyond standard Fortran 77.

10. Compiler Compatibility Table

Feature	gfortran	Intel Fortran
Recursion	-frecursive	/recursive
Function pointers	Supported	Supported
Nested functions	No	No

5.5 Function Examples in Fortran 77

1. Basic Function with Return Value

```
C     FUNCTION TO CALCULATE SQUARE
      REAL FUNCTION SQUARE(X)
      REAL X
      SQUARE = X * X
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL SQUARE
      WRITE(*,*) 'Square of 5.0:', SQUARE(5.0)
      STOP
      END
```

Explanation: Simple function demonstrating return value and type declaration.

2. Integer Function with Multiple Parameters

```
C     CALCULATE AVERAGE OF TWO INTEGERS
      INTEGER FUNCTION IAVG(A, B)
      INTEGER A, B
      IAVG = (A + B) / 2
      RETURN
      END

      PROGRAM MAIN
      INTEGER IAVG
```

```
      WRITE(*,*) 'Average(7,9):', IAVG(7,9)
      STOP
      END
```

3. Logical Function for Prime Check

```
C     CHECK PRIME NUMBER
      LOGICAL FUNCTION ISPRIME(N)
      INTEGER N, I
      ISPRIME = .TRUE.
      DO 10 I = 2, INT(SQRT(REAL(N)))
          IF (MOD(N,I) .EQ. 0) THEN
              ISPRIME = .FALSE.
              RETURN
          END IF
10    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      LOGICAL ISPRIME
      WRITE(*,*) '17 is prime:', ISPRIME(17)
      STOP
      END
```

4. Character Function

```
C     RETURN GRADE LETTER
      CHARACTER*1 FUNCTION GRADE(SCORE)
      REAL SCORE
      IF (SCORE .GE. 90.0) THEN
          GRADE = 'A'
      ELSE IF (SCORE .GE. 80.0) THEN
          GRADE = 'B'
      ELSE
          GRADE = 'F'
      END IF
      RETURN
      END

      PROGRAM MAIN
      CHARACTER*1 GRADE
      WRITE(*,*) 'Grade(85): ', GRADE(85.0)
      STOP
      END
```

5. Array Sum Function

```
C     SUM ARRAY ELEMENTS
      REAL FUNCTION ARRSUM(ARR, N)
      INTEGER N
      REAL ARR(N)
      INTEGER I
      ARRSUM = 0.0
      DO 20 I = 1, N
          ARRSUM = ARRSUM + ARR(I)
20    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL ARR(5), ARRSUM
      DATA ARR /1.0,2.0,3.0,4.0,5.0/
      WRITE(*,*) 'Array sum:', ARRSUM(ARR,5)
      STOP
      END
```

6. Matrix Trace Function

```
C     CALCULATE MATRIX TRACE
      REAL FUNCTION TRACE(MAT, N)
      INTEGER N
      REAL MAT(N,N)
      INTEGER I
      TRACE = 0.0
      DO 30 I = 1, N
          TRACE = TRACE + MAT(I,I)
30    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL MAT(3,3), TRACE
      DATA MAT /1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0/
      WRITE(*,*) 'Trace:', TRACE(MAT,3)
      STOP
      END
```

7. Function with Multiple Returns

```
C     CALCULATE BOTH SUM AND DIFFERENCE
      SUBROUTINE SUMDIFF(A, B, SUM, DIFF)
      REAL A, B, SUM, DIFF
```

```
      SUM = A + B
      DIFF = A - B
      RETURN
      END

      PROGRAM MAIN
      REAL S, D
      CALL SUMDIFF(8.0, 5.0, S, D)
      WRITE(*,*) 'Sum:', S, 'Diff:', D
      STOP
      END
```

8. Recursive Factorial (Compiler-dependent)

```
C     RECURSIVE FACTORIAL (NON-STANDARD)
      INTEGER FUNCTION FACT(N)
      INTEGER N
      IF (N .LE. 1) THEN
          FACT = 1
      ELSE
          FACT = N * FACT(N-1)
      END IF
      RETURN
      END

      PROGRAM MAIN
      INTEGER FACT
      WRITE(*,*) '5! =', FACT(5)
      STOP
      END
```

Note: Requires compiler support for recursion.

9. Function with Array Modification

```
C     DOUBLE ARRAY ELEMENTS
      SUBROUTINE DOUBLEARR(ARR, N)
      INTEGER N, I
      REAL ARR(N)
      DO 40 I = 1, N
          ARR(I) = ARR(I) * 2.0
40    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL NUM(3)
```

```
      DATA NUM /1.0,2.0,3.0/
      CALL DOUBLEARR(NUM,3)
      WRITE(*,*) 'Doubled:', NUM
      STOP
      END
```

10. Function as Argument

```
C     NUMERICAL INTEGRATION
      REAL FUNCTION INTEGRAL(FUNC, A, B, N)
      EXTERNAL FUNC
      REAL FUNC, A, B, DX, X, SUM
      INTEGER N, I
      DX = (B - A)/N
      SUM = 0.0
      DO 50 I = 1, N
          X = A + (I-0.5)*DX
          SUM = SUM + FUNC(X)
50    CONTINUE
      INTEGRAL = SUM * DX
      RETURN
      END

      REAL FUNCTION SQUARE(X)
      REAL X
      SQUARE = X**2
      RETURN
      END

      PROGRAM MAIN
      EXTERNAL SQUARE
      REAL INTEGRAL
      WRITE(*,*) 'Integral:', INTEGRAL(SQUARE,0.0,2.0,1000)
      STOP
      END
```

11. Error Handling in Function

```
C     SAFE DIVISION FUNCTION
      REAL FUNCTION SAFEDIV(A, B, ERROR)
      REAL A, B
      LOGICAL ERROR
      ERROR = .FALSE.
      IF (B .EQ. 0.0) THEN
          ERROR = .TRUE.
          SAFEDIV = 0.0
```

```
      ELSE
          SAFEDIV = A / B
      END IF
      RETURN
      END

      PROGRAM MAIN
      REAL SAFEDIV
      LOGICAL ERR
      WRITE(*,*) '10/0 =', SAFEDIV(10.0,0.0,ERR), 'Error:', ERR
      STOP
      END
```

12. String Manipulation Function

```
C     REVERSE STRING
      SUBROUTINE REVSTR(STR, LEN)
      INTEGER LEN, I
      CHARACTER STR*(*), TEMP
      DO 60 I = 1, LEN/2
          TEMP = STR(I:I)
          STR(I:I) = STR(LEN-I+1:LEN-I+1)
          STR(LEN-I+1:LEN-I+1) = TEMP
60    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      CHARACTER*10 S
      S = 'HELLO'
      CALL REVSTR(S,5)
      WRITE(*,*) 'Reversed:', S
      STOP
      END
```

13. Multi-dimensional Array Function

```
C     MATRIX MULTIPLICATION
      SUBROUTINE MATMUL(A,B,C,N)
      INTEGER N,I,J,K
      REAL A(N,N), B(N,N), C(N,N)
      DO 70 I=1,N
      DO 70 J=1,N
          C(I,J)=0.0
          DO 70 K=1,N
70            C(I,J)=C(I,J)+A(I,K)*B(K,J)
```

```
      RETURN
      END

      PROGRAM MAIN
      REAL A(2,2),B(2,2),C(2,2)
      DATA A/1.0,2.0,3.0,4.0/, B/5.0,6.0,7.0,8.0/
      CALL MATMUL(A,B,C,2)
      WRITE(*,*) 'Product:', C
      STOP
      END
```

14. Function with Variable Arguments

```
C     CALCULATE MEAN OF VARIABLE ARGUMENTS
      REAL FUNCTION MEAN(N, ...)
C     WARNING: NOT STANDARD FORTRAN 77
C     (Requires compiler-specific implementation)
      INTEGER N,I
      REAL SUM,X
      SUM = 0.0
      DO 80 I = 1, N
          X = VARARG(I)  ! Pseudo-code
          SUM = SUM + X
80    CONTINUE
      MEAN = SUM/N
      RETURN
      END
```

Note: Demonstrates conceptual variable arguments.

15. Function Returning Array

```
C     RETURN ARRAY OF SQUARES
      SUBROUTINE SQUARES(ARR, N)
      INTEGER N, I
      REAL ARR(N)
      DO 90 I = 1, N
          ARR(I) = REAL(I)**2
90    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL NUM(5)
      CALL SQUARES(NUM,5)
      WRITE(*,*) 'Squares:', NUM
      STOP
```

```
      END
```

16. Type Conversion Function

```
C     FAHRENHEIT TO CELSIUS
      REAL FUNCTION F2C(F)
      REAL F
      F2C = (F - 32.0) * 5.0/9.0
      RETURN
      END

      PROGRAM MAIN
      REAL F2C
      WRITE(*,*) '32F =', F2C(32.0), 'C'
      STOP
      END
```

17. Function with COMMON Block

```
C     GLOBAL CONSTANT USING COMMON
      REAL FUNCTION CIRCUM(R)
      REAL R, PI
      COMMON /CONST/ PI
      CIRCUM = 2.0 * PI * R
      RETURN
      END

      PROGRAM MAIN
      REAL CIRCUM, PI
      COMMON /CONST/ PI
      PI = 3.14159
      WRITE(*,*) 'Circumference:', CIRCUM(1.0)
      STOP
      END
```

18. Function with SAVE Attribute

```
C     COUNTER WITH PERSISTENT STATE
      INTEGER FUNCTION COUNTER()
      INTEGER COUNT
      SAVE COUNT
      DATA COUNT /0/
      COUNT = COUNT + 1
      COUNTER = COUNT
      RETURN
      END
```

```
      PROGRAM MAIN
      WRITE(*,*) 'Count:', COUNTER(), COUNTER(), COUNTER()
      STOP
      END
```

19. Bitwise Operations Function

```
C     BITWISE AND FUNCTION
      INTEGER FUNCTION BITAND(A, B)
      INTEGER A, B
      BITAND = AND(A, B)
      RETURN
      END

      PROGRAM MAIN
      INTEGER BITAND
      WRITE(*,*) '5 & 3 =', BITAND(5,3)
      STOP
      END
```

20. Complex Number Function

```
C     COMPLEX NUMBER ADDITION
      COMPLEX FUNCTION CADD(A, B)
      COMPLEX A, B
      CADD = A + B
      RETURN
      END

      PROGRAM MAIN
      COMPLEX C1, C2, C3, CADD
      C1 = (1.0, 2.0)
      C2 = (3.0, 4.0)
      C3 = CADD(C1, C2)
      WRITE(*,*) 'Sum:', C3
      STOP
      END
```

5.6 Exercises: Functions in Fortran 77

5.6.1 Basic Function Implementation

1. **Area of Circle**: Write a real function ‘CIRCLE$_AREA(R)‘that calculates the area of a circle. Sample : $Input = 3.0ßOutput28.2743$

2. **Factorial Function**: Create an integer function ‘FACT(N)‘ to compute factorial (iterative approach). Sample: Input=5 → Output=120

3. **Even/Odd Check**: Implement a logical function ‘ISEVEN(NUM)‘ returning ‘.TRUE.‘ for even integers. Sample: Input=7 → Output=.FALSE.

4. **Grade Converter**: Write a character function ‘GRADE(SCORE)‘ returning ’A’-’F’ based on score (90-100: ’A’, etc.).

Array & Matrix Functions

5. **Array Sum**: Create a real function ‘ARRAY$_SUM(ARR, N)$‘$to sum elements of a 1D array$.

6. **Matrix Trace**: Implement a real function ‘TRACE(MAT, N)‘ to calculate the trace of an N×N matrix.

7. **Maximum Element**: Write a function ‘ARRAY$_MAX(ARR, N)$‘$returning the largest value in a 1D array$.

8. **Matrix Symmetry Check**: Develop a logical function ‘IS$_SYMMETRIC(MAT, N)$‘$to check if a matrix is sy$

String & Character Functions

9. **String Reversal**: Create a subroutine ‘REVERSE$_STR(STR, LEN)$‘$to reverse a character string$.

10. **Vowel Counter**: Implement an integer function ‘COUNT$_VOWELS(STR)$‘$returning the number of vowel$

11. **Palindrome Check**: Write a logical function ‘IS$_PALINDROME(STR)$‘$to check if a string reads the same b$

Mathematical Functions

12. **Prime Check**: Develop a logical function ‘IS$_PRIME(N)$‘$to test primality of an integer$.

13. **Temperature Conversion**: Create a real function ‘F2C(F)‘ converting Fahrenheit to Celsius.

14. **Dot Product**: Implement a real function ‘DOT$_PROD(VEC1, VEC2, N)$‘$for two N-element vectors$.

15. **Standard Deviation**: Write a function ‘STD$_DEV(ARR, N)$‘$to calculate standard deviation of an array$.

Advanced Function Concepts

16. **Common Block Function**: Create a function ‘CIRCUMFERENCE(R)‘ using a COMMON block to store (3.14159).

17. **Persistent Counter**: Implement an integer function ‘COUNTER()‘ with SAVE attribute to increment on each call.

18. **Function Argument**: Write an integration function ‘INTEGRATE(FUNC, A, B, N)‘ accepting another function as argument.

19. **Bitwise Operations**: Create integer functions for: a) ‘BITWISE$_AND(A, B)$‘$b)$‘$BITWISE_OR(A, B)$‘

20. **Complex Numbers**: Implement a complex function ‘C$_ADD(A, B)$‘$to add two complex numbers$.

Error Handling & Validation

21. **Safe Division**: Create a function ‘SAFE$_DIV(A, B, ERROR)$‘$that sets ERROR flag for division by zero$.

22. **Input Validator**: Write a logical function ‘VALID$_INPUT(STR)$‘$checking if a string contains only digits$.

Challenge Problems

23. **Matrix Multiplier**: Develop a subroutine 'MAT$_MUL(A, B, C, N)$'$multiplying two N N matrices$.

24. **Function Composition**: Implement 'FOG(X) = F(G(X))' where F(X)=x^2 and G(X)=x+1 as separate functions.

25. **Statistical Suite**: Create a set of functions: - 'MEAN(ARR, N)' - 'MEDIAN(ARR, N)' - 'MODE(ARR, N)'

5.7 Exercise Answers: Functions in Fortran 77

1. Area of Circle

```
C     CALCULATE AREA OF CIRCLE
      REAL FUNCTION CIRCLE_AREA(R)
      REAL R, PI
      PARAMETER (PI = 3.14159)
      CIRCLE_AREA = PI * R**2
      RETURN
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      REAL CIRCLE_AREA
      WRITE(*,*) 'Area:', CIRCLE_AREA(3.0)
      STOP
      END
```

Explanation: Uses constant and square calculation. Returns real value.

2. Factorial Function

```
C     ITERATIVE FACTORIAL
      INTEGER FUNCTION FACT(N)
      INTEGER N, I
      FACT = 1
      DO 10 I = 1, N
          FACT = FACT * I
10    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      INTEGER FACT
      WRITE(*,*) '5! =', FACT(5)
      STOP
      END
```

Note: Initializes result to 1 and multiplies sequentially.

3. Even/Odd Check

```
C     EVEN NUMBER CHECK
      LOGICAL FUNCTION ISEVEN(NUM)
      INTEGER NUM
      ISEVEN = MOD(NUM, 2) .EQ. 0
      RETURN
      END

      PROGRAM MAIN
      LOGICAL ISEVEN
      WRITE(*,*) '7 is even?', ISEVEN(7)
      STOP
      END
```

Logic: Uses modulus operator for even check.

4. Grade Converter

```
C     GRADE CONVERSION
      CHARACTER*1 FUNCTION GRADE(SCORE)
      REAL SCORE
      IF (SCORE .GE. 90.0) THEN
          GRADE = 'A'
      ELSE IF (SCORE .GE. 80.0) THEN
          GRADE = 'B'
      ELSE IF (SCORE .GE. 70.0) THEN
          GRADE = 'C'
      ELSE
          GRADE = 'F'
      END IF
      RETURN
      END

      PROGRAM MAIN
      CHARACTER*1 GRADE
      WRITE(*,*) 'Grade 85:', GRADE(85.0)
      STOP
      END
```

5. Array Sum

```
C     SUM ARRAY ELEMENTS
      REAL FUNCTION ARRAY_SUM(ARR, N)
      INTEGER N
      REAL ARR(N)
      INTEGER I
```

```
      ARRAY_SUM = 0.0
      DO 20 I = 1, N
          ARRAY_SUM = ARRAY_SUM + ARR(I)
20    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL ARR(5), ARRAY_SUM
      DATA ARR /1.0,2.0,3.0,4.0,5.0/
      WRITE(*,*) 'Sum:', ARRAY_SUM(ARR,5)
      STOP
      END
```

6. Matrix Trace

```
C     MATRIX TRACE CALCULATION
      REAL FUNCTION TRACE(MAT, N)
      INTEGER N
      REAL MAT(N,N)
      INTEGER I
      TRACE = 0.0
      DO 30 I = 1, N
          TRACE = TRACE + MAT(I,I)
30    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL MAT(3,3), TRACE
      DATA MAT /1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0/
      WRITE(*,*) 'Trace:', TRACE(MAT,3)
      STOP
      END
```

7. Maximum Element

```
C     FIND ARRAY MAXIMUM
      REAL FUNCTION ARRAY_MAX(ARR, N)
      INTEGER N
      REAL ARR(N)
      INTEGER I
      ARRAY_MAX = ARR(1)
      DO 40 I = 2, N
          IF (ARR(I) .GT. ARRAY_MAX) THEN
              ARRAY_MAX = ARR(I)
```

```
          END IF
40    CONTINUE
      RETURN
      END

      PROGRAM MAIN
      REAL ARR(5), ARRAY_MAX
      DATA ARR /3.0,1.0,4.0,1.0,5.0/
      WRITE(*,*) 'Max:', ARRAY_MAX(ARR,5)
      STOP
      END
```

8. Matrix Symmetry Check

```
C     CHECK MATRIX SYMMETRY
      LOGICAL FUNCTION IS_SYMMETRIC(MAT, N)
      INTEGER N
      REAL MAT(N,N)
      INTEGER I, J
      IS_SYMMETRIC = .TRUE.
      DO 50 I = 1, N
          DO 60 J = I+1, N
              IF (MAT(I,J) .NE. MAT(J,I)) THEN
                  IS_SYMMETRIC = .FALSE.
                  RETURN
              END IF
60        CONTINUE
50    CONTINUE
      RETURN
      END
```

9. String Reversal

```
C     REVERSE STRING
      SUBROUTINE REVERSE_STR(STR, LEN)
      INTEGER LEN, I
      CHARACTER STR*(*), TEMP
      DO 70 I = 1, LEN/2
          TEMP = STR(I:I)
          STR(I:I) = STR(LEN-I+1:LEN-I+1)
          STR(LEN-I+1:LEN-I+1) = TEMP
70    CONTINUE
      RETURN
      END

      PROGRAM MAIN
```

```
      CHARACTER*5 S
      S = 'HELLO'
      CALL REVERSE_STR(S,5)
      WRITE(*,*) 'Reversed:', S
      STOP
      END
```

10. Vowel Counter

```
C     COUNT VOWELS
      INTEGER FUNCTION COUNT_VOWELS(STR)
      CHARACTER*(*) STR
      INTEGER I, LEN
      COUNT_VOWELS = 0
      LEN = LEN_TRIM(STR)
      DO 80 I = 1, LEN
          IF (INDEX('AEIOUaeiou', STR(I:I)) .GT. 0) THEN
              COUNT_VOWELS = COUNT_VOWELS + 1
          END IF
80    CONTINUE
      RETURN
      END
```

11. Palindrome Check

```
C     PALINDROME CHECK
      LOGICAL FUNCTION IS_PALINDROME(STR)
      CHARACTER*(*) STR, REV_STR
      INTEGER LEN
      LEN = LEN_TRIM(STR)
      REV_STR = STR
      CALL REVERSE_STR(REV_STR, LEN)
      IS_PALINDROME = STR(1:LEN) .EQ. REV_STR(1:LEN)
      RETURN
      END
```

12. Prime Check

```
C     PRIME CHECK
      LOGICAL FUNCTION IS_PRIME(N)
      INTEGER N, I
      IF (N .LE. 1) THEN
          IS_PRIME = .FALSE.
          RETURN
      END IF
      DO 90 I = 2, SQRT(REAL(N))
          IF (MOD(N,I) .EQ. 0) THEN
```

```
            IS_PRIME = .FALSE.
            RETURN
        END IF
90    CONTINUE
      IS_PRIME = .TRUE.
      RETURN
      END
```

13. Temperature Conversion

```
C     FAHRENHEIT TO CELSIUS
      REAL FUNCTION F2C(F)
      REAL F
      F2C = (F - 32.0) * 5.0/9.0
      RETURN
      END

      PROGRAM MAIN
      REAL F2C
      WRITE(*,*) '212F =', F2C(212.0), 'C'
      STOP
      END
```

14. Dot Product

```
C     DOT PRODUCT
      REAL FUNCTION DOT_PROD(VEC1, VEC2, N)
      INTEGER N
      REAL VEC1(N), VEC2(N)
      INTEGER I
      DOT_PROD = 0.0
      DO 100 I = 1, N
          DOT_PROD = DOT_PROD + VEC1(I)*VEC2(I)
100   CONTINUE
      RETURN
      END
```

15. Standard Deviation

```
C     STANDARD DEVIATION
      REAL FUNCTION STD_DEV(ARR, N)
      REAL ARR(N), MEAN
      INTEGER N, I
      MEAN = ARRAY_SUM(ARR,N)/REAL(N)
      STD_DEV = 0.0
      DO 110 I = 1, N
          STD_DEV = STD_DEV + (ARR(I)-MEAN)**2
```

```
110   CONTINUE
      STD_DEV = SQRT(STD_DEV/REAL(N))
      RETURN
      END
```

16. Common Block Circumference

```
C     CIRCUMFERENCE WITH COMMON
      REAL FUNCTION CIRCUM(R)
      REAL R, PI
      COMMON /CONST/ PI
      CIRCUM = 2.0 * PI * R
      RETURN
      END

      PROGRAM MAIN
      REAL CIRCUM, PI
      COMMON /CONST/ PI
      PI = 3.14159
      WRITE(*,*) 'Circumference:', CIRCUM(1.0)
      STOP
      END
```

17. Persistent Counter

```
C     PERSISTENT COUNTER
      INTEGER FUNCTION COUNTER()
      INTEGER COUNT
      SAVE COUNT
      DATA COUNT /0/
      COUNT = COUNT + 1
      COUNTER = COUNT
      RETURN
      END
```

18. Function Argument Integration

```
C     NUMERICAL INTEGRATION
      REAL FUNCTION INTEGRAL(FUNC, A, B, N)
      EXTERNAL FUNC
      REAL FUNC, A, B, DX, X, SUM
      INTEGER N, I
      DX = (B - A)/N
      SUM = 0.0
      DO 120 I = 1, N
          X = A + (I-0.5)*DX
          SUM = SUM + FUNC(X)
```

```
120   CONTINUE
      INTEGRAL = SUM * DX
      RETURN
      END
```

19. Bitwise Operations

```
C     BITWISE AND
      INTEGER FUNCTION BITWISE_AND(A, B)
      INTEGER A, B
      BITWISE_AND = AND(A, B)
      RETURN
      END

C     BITWISE OR
      INTEGER FUNCTION BITWISE_OR(A, B)
      INTEGER A, B
      BITWISE_OR = OR(A, B)
      RETURN
      END
```

20. Complex Number Addition

```
C     COMPLEX ADDITION
      COMPLEX FUNCTION C_ADD(A, B)
      COMPLEX A, B
      C_ADD = A + B
      RETURN
      END

      PROGRAM MAIN
      COMPLEX C1, C2, C_ADD
      C1 = (1.0, 2.0)
      C2 = (3.0, 4.0)
      WRITE(*,*) 'Sum:', C_ADD(C1, C2)
      STOP
      END
```

5.8 Problem Solving Methodologies

Exercise 1: Area of Circle

Problem Analysis

Calculate the area of a circle using formula $A = \pi r^2$.

Solution Approach

1. Create real function `CIRCLE_AREA` accepting radius
2. Declare constant π using `PARAMETER`
3. Implement area formula
4. Return calculated value

Key Concepts

- Function declaration with return type
- Constant parameters
- Arithmetic operations

Exercise 2: Factorial Function

Problem Analysis

Compute $n!$ iteratively.

Solution Approach

1. Initialize result to 1
2. Multiply sequentially from 1 to n
3. Return accumulated product

Important Notes

- Handles $n = 0$ correctly
- Uses integer type for exact results

Exercise 3: Even/Odd Check

Problem Analysis

Determine if number is even using modulus operation.

Solution Approach

1. Use `MOD` function with divisor 2
2. Return `.TRUE.` if remainder is 0
3. Logical result directly from comparison

Optimization

- Single-line implementation possible
- No explicit `IF` statement needed

Exercise 4: Grade Converter

Problem Analysis

Convert numerical score to letter grade.

Solution Strategy

1. Use cascading `IF-ELSE` structure
2. Compare score against thresholds
3. Return corresponding character

Edge Cases

- Handles scores above 100 and below 0
- Returns 'F' as default case

Exercise 5: Array Sum

Implementation Logic

1. Initialize sum to 0.0
2. Iterate through array elements
3. Accumulate total using loop

Memory Considerations

- Array passed by reference
- No size limit except memory constraints

Exercise 6: Matrix Trace

Algorithm Steps

1. Initialize sum to 0.0
2. Iterate diagonal elements (i, i)
3. Accumulate diagonal values

Matrix Handling

- Column-major order irrelevant for trace
- Works for any square matrix size

Exercise 7: Maximum Element

Search Strategy

1. Assume first element is maximum
2. Compare with subsequent elements
3. Update maximum when larger value found

Efficiency

- Single pass $O(n)$ complexity
- Requires $n - 1$ comparisons

Exercise 8: Matrix Symmetry

Verification Method

1. Check $mat(i, j) = mat(j, i) \forall i, j$
2. Early exit on first mismatch
3. Upper triangular comparison

Optimization

- Avoids redundant comparisons
- Uses $j = i + 1$ to reduce iterations

Exercise 9: String Reversal

In-Place Algorithm

1. Swap characters from ends to middle
2. Use temporary character storage
3. Handle even/odd length strings

String Handling

- Fortran substring notation
- Implicit length handling

Exercise 10: Vowel Counter

Detection Method

1. Check each character against vowel set
2. Use `INDEX` function for membership test
3. Case-insensitive comparison

Efficiency Note

- Linear scan $O(n)$ complexity
- Alternative: Use logical OR of comparisons

[Continued in similar format for remaining exercises...]

Exercise 20: Complex Addition

Complex Handling

1. Use Fortran complex data type
2. Leverage built-in complex arithmetic
3. Return complex result directly

Type Safety

- Implicit complex operations
- Real and imaginary parts handled automatically

General Problem Solving Patterns

- **Input Validation**: Check for valid ranges/values
- **Edge Cases**: Handle minimum/maximum values
- **Efficiency**: Optimize loop structures
- **Memory**: Consider array passing mechanisms
- **Modularity**: Decompose into sub-functions

Chapter 6

Recursion in Fortran 77

Conceptual Overview

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. A recursive function typically consists of:

- **Base Case**: Termination condition preventing infinite loops
- **Recursive Case**: Function calls itself with modified parameters

Fortran 77 Implementation Challenges

- **No Official Support**: Original Fortran 77 standard prohibits recursion
- **Compiler Extensions**: Some modern compilers (e.g., gfortran) allow recursion with flags
- **Stack Limitations**: Deep recursion may cause stack overflows

Enabling Recursion in Modern Compilers

Example compilation flags:

```
gfortran -frecursive program.f  # GNU Fortran
ifort -recursive program.f      # Intel Fortran
```

Example 1: Factorial Calculation

```
C     RECURSIVE FACTORIAL FUNCTION
      RECURSIVE INTEGER FUNCTION FACT(N) RESULT(RES)
      INTEGER N
      IF (N <= 0) THEN
          RES = 1
      ELSE
          RES = N * FACT(N-1)
      END IF
```

```
      END

C     MAIN PROGRAM
      PROGRAM MAIN
      INTEGER FACT
      WRITE(*,*) '5! =', FACT(5)
      STOP
      END
```

Components:

- `RECURSIVE` keyword declares recursive capability
- `RESULT` clause specifies return variable
- Base case: $n \leq 0$ returns 1
- Recursive case: $n \times fact(n-1)$

Example 2: Fibonacci Sequence

```
C     RECURSIVE FIBONACCI
      RECURSIVE INTEGER FUNCTION FIB(N) RESULT(RES)
      INTEGER N
      IF (N <= 0) THEN
          RES = 0
      ELSE IF (N == 1) THEN
          RES = 1
      ELSE
          RES = FIB(N-1) + FIB(N-2)
      END IF
      END

      PROGRAM MAIN
      INTEGER FIB
      WRITE(*,*) 'Fib(10) =', FIB(10)
      STOP
      END
```

Key Considerations

Aspect	Details
Stack Depth	Limited by compiler/memory settings
Performance	Generally slower than iteration
Memory Use	Grows linearly with recursion depth
Readability	Often clearer for mathematical problems

Appropriate Use Cases

- Mathematical series (factorial, Fibonacci)
- Tree traversals (in hierarchical data structures)
- Divide-and-conquer algorithms (QuickSort)
- Backtracking algorithms (permutations)

Performance Comparison: Recursive vs Iterative

```
C     ITERATIVE FACTORIAL
      INTEGER FUNCTION ITER_FACT(N)
      INTEGER N, I
      ITER_FACT = 1
      DO 10 I = 1, N
          ITER_FACT = ITER_FACT * I
10    CONTINUE
      END
```

Advantages of Iteration:

- Fixed memory usage ($O(1)$)
- Faster execution (no function call overhead)
- No stack overflow risk

Recursion Best Practices

1. Always define clear base cases
2. Limit recursion depth (¡1000 levels)
3. Prefer iteration for performance-critical code
4. Use compiler warnings (-Wall -Wextra)
5. Test across different compilers

Advanced Example: Binary Search

```
C     RECURSIVE BINARY SEARCH
      RECURSIVE INTEGER FUNCTION BSEARCH(ARR, L, R, X) RESULT(INDEX)
      INTEGER ARR(*), L, R, X, MID
      IF (R >= L) THEN
          MID = L + (R - L)/2
          IF (ARR(MID) == X) THEN
              INDEX = MID
          ELSE IF (ARR(MID) > X) THEN
```

```
            INDEX = BSEARCH(ARR, L, MID-1, X)
        ELSE
            INDEX = BSEARCH(ARR, MID+1, R, X)
        END IF
    ELSE
        INDEX = -1
    END IF
    END

    PROGRAM MAIN
    INTEGER ARR(5), BSEARCH
    DATA ARR /2,4,6,8,10/
    WRITE(*,*) 'Found at:', BSEARCH(ARR,1,5,8)
    STOP
    END
```

Limitations and Risks

- **Stack Overflow**: Deep recursion may crash program
- **Portability**: Non-standard across compilers
- **Debugging Difficulty**: Complex call stacks
- **Memory Efficiency**: Worse than iteration

Historical Context

Original Fortran 77 restrictions stemmed from:

- Early computer memory limitations
- Static memory allocation requirements
- Focus on numerical/scientific computations

Modern Alternatives

For projects requiring recursion:

- Use Fortran 90+ with standard recursion support
- Implement recursive algorithms iteratively
- Combine Fortran with recursive-friendly languages

6.1 Recursive Programming Examples in Fortran 77

1. Factorial Calculation

```
RECURSIVE INTEGER FUNCTION FACT(n) RESULT(res)
INTEGER, INTENT(IN) :: n
IF (n <= 0) THEN
    res = 1
ELSE
    res = n * FACT(n-1)
END IF
END FUNCTION

! Working Principle:
! Base case: n  0 returns 1
! Recursive case: n * fact(n-1)
! Tree: Linear single recursion
```

2. Fibonacci Sequence

```
RECURSIVE INTEGER FUNCTION FIB(n) RESULT(res)
INTEGER, INTENT(IN) :: n
IF (n <= 0) THEN
    res = 0
ELSE IF (n == 1) THEN
    res = 1
ELSE
    res = FIB(n-1) + FIB(n-2)
END IF
END FUNCTION

! Working Principle:
! Binary recursion with two base cases
! Exponential time complexity O(2^n)
```

3. Greatest Common Divisor (GCD)

```
RECURSIVE INTEGER FUNCTION GCD(a,b) RESULT(res)
INTEGER, INTENT(IN) :: a, b
IF (b == 0) THEN
    res = a
ELSE
    res = GCD(b, MOD(a,b))
END IF
END FUNCTION

! Working Principle:
```

```
! Euclid's algorithm implementation
! Recursively applies GCD(b, a mod b)
```

4. Array Summation

```
RECURSIVE REAL FUNCTION ARRAY_SUM(arr, n) RESULT(res)
REAL, INTENT(IN) :: arr(n)
INTEGER, INTENT(IN) :: n
IF (n == 0) THEN
    res = 0.0
ELSE
    res = arr(n) + ARRAY_SUM(arr, n-1)
END IF
END FUNCTION

! Working Principle:
! Accumulates sum from last element backward
! Linear recursion O(n) complexity
```

5. Binary Search

```
RECURSIVE INTEGER FUNCTION BSEARCH(arr, l, r, x) RESULT(res)
INTEGER, INTENT(IN) :: arr(*), l, r, x
INTEGER :: mid
IF (r >= l) THEN
    mid = l + (r - l)/2
    IF (arr(mid) == x) THEN
        res = mid
    ELSE IF (arr(mid) > x) THEN
        res = BSEARCH(arr, l, mid-1, x)
    ELSE
        res = BSEARCH(arr, mid+1, r, x)
    END IF
ELSE
    res = -1
END IF
END FUNCTION

! Working Principle:
! Divide-and-conquer approach
! Log(n) recursive calls
```

6. Tower of Hanoi

```
RECURSIVE SUBROUTINE HANOI(n, from, to, aux)
INTEGER, INTENT(IN) :: n
CHARACTER(*), INTENT(IN) :: from, to, aux
```

```
IF (n == 1) THEN
    PRINT *, "Move disk 1 from ", from, " to ", to
ELSE
    CALL HANOI(n-1, from, aux, to)
    PRINT *, "Move disk ", n, " from ", from, " to ", to
    CALL HANOI(n-1, aux, to, from)
END IF
END SUBROUTINE

! Working Principle:
! Moves n-1 disks to auxiliary tower
! Moves nth disk to target
! Recursively moves n-1 disks from auxiliary
```

7. Palindrome Check

```
RECURSIVE LOGICAL FUNCTION IS_PAL(str, l, r) RESULT(res)
CHARACTER(*), INTENT(IN) :: str
INTEGER, INTENT(IN) :: l, r
IF (l >= r) THEN
    res = .TRUE.
ELSE IF (str(l:l) /= str(r:r)) THEN
    res = .FALSE.
ELSE
    res = IS_PAL(str, l+1, r-1)
END IF
END FUNCTION

! Working Principle:
! Compares characters at both ends
! Moves toward center recursively
```

8. Power Calculation

```
RECURSIVE REAL FUNCTION POWER(x, n) RESULT(res)
REAL, INTENT(IN) :: x
INTEGER, INTENT(IN) :: n
IF (n == 0) THEN
    res = 1.0
ELSE IF (n > 0) THEN
    res = x * POWER(x, n-1)
ELSE
    res = 1.0 / POWER(x, -n)
END IF
END FUNCTION
```

```
! Working Principle:
! Handles positive/negative exponents
! Recursive multiplication/division
```

9. Flood Fill Algorithm

```
RECURSIVE SUBROUTINE FLOOD_FILL(grid, x, y, old, new)
INTEGER, INTENT(INOUT) :: grid(:,:)
INTEGER, INTENT(IN) :: x, y, old, new
IF (x < 1 .OR. x > SIZE(grid,1)) RETURN
IF (y < 1 .OR. y > SIZE(grid,2)) RETURN
IF (grid(x,y) /= old) RETURN

grid(x,y) = new
CALL FLOOD_FILL(grid, x+1, y, old, new)
CALL FLOOD_FILL(grid, x-1, y, old, new)
CALL FLOOD_FILL(grid, x, y+1, old, new)
CALL FLOOD_FILL(grid, x, y-1, old, new)
END SUBROUTINE

! Working Principle:
! 4-directional recursive filling
! Base cases: Boundary checks and color match
```

10. String Reversal

```
RECURSIVE SUBROUTINE REVERSE_STR(str, l, r)
CHARACTER(*), INTENT(INOUT) :: str
INTEGER, INTENT(IN) :: l, r
CHARACTER :: temp
IF (l < r) THEN
    temp = str(l:l)
    str(l:l) = str(r:r)
    str(r:r) = temp
    CALL REVERSE_STR(str, l+1, r-1)
END IF
END SUBROUTINE

! Working Principle:
! Swaps characters at ends and moves inward
! Terminates when pointers cross
```

11. Linked List Traversal

```
TYPE Node
INTEGER :: data
INTEGER :: next
```

```
END TYPE

RECURSIVE SUBROUTINE TRAVERSE(list, index)
TYPE(Node), INTENT(IN) :: list(:)
INTEGER, INTENT(IN) :: index
IF (index /= 0) THEN
    PRINT *, list(index)%data
    CALL TRAVERSE(list, list(index)%next)
END IF
END SUBROUTINE

! Working Principle:
! Recursive traversal using index pointers
! Simulates pointer-based recursion
```

12. Tree Inorder Traversal

```
TYPE TreeNode
INTEGER :: data
INTEGER :: left
INTEGER :: right
END TYPE

RECURSIVE SUBROUTINE INORDER(tree, root)
TYPE(TreeNode), INTENT(IN) :: tree(:)
INTEGER, INTENT(IN) :: root
IF (root /= 0) THEN
    CALL INORDER(tree, tree(root)%left)
    PRINT *, tree(root)%data
    CALL INORDER(tree, tree(root)%right)
END IF
END SUBROUTINE

! Working Principle:
! Visits left subtree → root → right subtree
! Recursive depth-first traversal
```

13. Permutations Generation

```
RECURSIVE SUBROUTINE PERMUTE(arr, l, r)
INTEGER, INTENT(INOUT) :: arr(:)
INTEGER, INTENT(IN) :: l, r
INTEGER :: i, temp
IF (l == r) THEN
    PRINT *, arr
ELSE
```

```
    DO i = l, r
        temp = arr(l)
        arr(l) = arr(i)
        arr(i) = temp
        CALL PERMUTE(arr, l+1, r)
        temp = arr(l)
        arr(l) = arr(i)
        arr(i) = temp
    END DO
END IF
END SUBROUTINE

! Working Principle:
! Heap's algorithm implementation
! Backtracking through recursive swaps
```

14. Directory Traversal

```
RECURSIVE SUBROUTINE LIST_DIR(path)
CHARACTER(*), INTENT(IN) :: path
CHARACTER(256) :: cmd, newpath
INTEGER :: status

CALL SYSTEM('ls '//TRIM(path)//' > dirlist.tmp')
OPEN(UNIT=10, FILE='dirlist.tmp', STATUS='OLD')
DO WHILE (.TRUE.)
    READ(10,*,IOSTAT=status) cmd
    IF (status /= 0) EXIT
    IF (cmd(1:1) == 'D') THEN
        newpath = TRIM(path)//'/'//cmd(3:)
        CALL LIST_DIR(TRIM(newpath))
    END IF
END DO
CLOSE(10, STATUS='DELETE')
END SUBROUTINE

! Working Principle:
! Recursive directory listing
! Uses system calls for directory detection
```

15. Maze Solver

```
RECURSIVE LOGICAL FUNCTION SOLVE_MAZE(maze, x, y) RESULT(res)
INTEGER, INTENT(INOUT) :: maze(:,:)
INTEGER, INTENT(IN) :: x, y
```

```
IF (x < 1 .OR. x > SIZE(maze,1)) THEN; res = .FALSE.; RETURN; END IF
IF (y < 1 .OR. y > SIZE(maze,2)) THEN; res = .FALSE.; RETURN; END IF

IF (maze(x,y) == 9) THEN; res = .TRUE.; RETURN; END IF
IF (maze(x,y) /= 0) THEN; res = .FALSE.; RETURN; END IF

maze(x,y) = 2  ! Mark path
res = SOLVE_MAZE(maze, x+1, y) .OR. SOLVE_MAZE(maze, x-1, y) &
    .OR. SOLVE_MAZE(maze, x, y+1) .OR. SOLVE_MAZE(maze, x, y-1)
IF (.NOT. res) maze(x,y) = 3  ! Mark dead end
END FUNCTION

! Working Principle:
! 4-directional path finding with backtracking
! Marks current path and dead ends
```

Important Notes

- All examples require compiler flags for recursion support
- Actual Fortran 77 implementations need WORKAROUNDS for:
  - Derived types (use arrays instead)
  - Dynamic memory (use fixed-size arrays)
  - System calls (implementation-dependent)
- Recursion depth limited by stack size
- Iterative implementations preferred for production code

6.2 Exercises: Recursion in Fortran 77

Basic Recursion Concepts

1. **Factorial Function**: Implement a recursive function ‘FACT(n)‘ to compute the factorial of a non-negative integer. Explain how the base case and recursive step work.

2. **Fibonacci Sequence**: Write a recursive function ‘FIB(n)‘ to return the nth Fibonacci number. Discuss the inefficiency of this approach and suggest an optimization.

3. **Array Sum**: Create a recursive function ‘ARRAY$_SUM(arr, n)‘to calculate the sum of elements in a 1D array. Specify$

4. **String Length**: Design a recursive function ‘STRLEN(str)‘ to compute the length of a character string without using Fortran’s intrinsic ‘LEN‘ function.

Algorithmic Problems

5. **Greatest Common Divisor (GCD)**: Implement Euclid’s algorithm recursively in a function ‘GCD(a, b)‘. Explain the mathematical basis for the recursive step.

6. **Tower of Hanoi**: Write a recursive subroutine 'HANOI(n, source, target, auxiliary)' to solve the Tower of Hanoi problem for n disks. List the sequence of moves for $n = 3$.

7. **Binary Search**: Develop a recursive function 'BSEARCH(arr, low, high, key)' to perform binary search on a sorted array. State the time complexity.

8. **Palindrome Check**: Create a recursive logical function '$\mathrm{IS}_P AL(str, start, end)$'$tocheckifasubstringisapali$

Advanced Applications

9. **Flood Fill Algorithm**: Design a recursive subroutine '$\mathrm{FLOOD}_F ILL(grid, x, y, old, new)$'$toimplementthefloo$

10. **Recursion Limitations**: Convert the recursive factorial function from Exercise 1 into an iterative version. Discuss why iteration might be preferred in Fortran 77.

6.3 Exercise Answers: Recursion in Fortran 77

1. Factorial Function

```
C     RECURSIVE FACTORIAL FUNCTION
      RECURSIVE INTEGER FUNCTION FACT(N) RESULT(RES)
      INTEGER, INTENT(IN) :: N
      IF (N <= 0) THEN
          RES = 1  ! BASE CASE
      ELSE
          RES = N * FACT(N - 1)  ! RECURSIVE CASE
      END IF
      END FUNCTION

C     MAIN PROGRAM
      PROGRAM MAIN
      INTEGER :: NUM = 5
      WRITE(*,*) '5! = ', FACT(NUM)
      STOP
      END
```

Explanation: - Base case: $n \leq 0$ returns 1 - Recursive step: $n \times fact(n-1)$ - Requires compiler flag: `-frecursive` in gfortran - Stack depth: $O(n)$

2. Fibonacci Sequence

```
C     RECURSIVE FIBONACCI
      RECURSIVE INTEGER FUNCTION FIB(N) RESULT(RES)
      INTEGER, INTENT(IN) :: N
      IF (N <= 0) THEN
          RES = 0
      ELSE IF (N == 1) THEN
          RES = 1
      ELSE
```

```
          RES = FIB(N-1) + FIB(N-2)
      END IF
      END FUNCTION

C     MAIN PROGRAM
      PROGRAM MAIN
      WRITE(*,*) 'FIB(6) = ', FIB(6)  ! OUTPUT: 8
      STOP
      END
```

Explanation: - Two base cases ($n = 0$, $n = 1$) - Exponential time complexity $O(2^n)$ - Optimization: Use memoization or iteration

3. Array Sum

```
C     RECURSIVE ARRAY SUM
      RECURSIVE REAL FUNCTION ARRAY_SUM(ARR, N) RESULT(SUM)
      REAL, INTENT(IN) :: ARR(N)
      INTEGER, INTENT(IN) :: N
      IF (N == 0) THEN
          SUM = 0.0
      ELSE
          SUM = ARR(N) + ARRAY_SUM(ARR, N-1)
      END IF
      END FUNCTION

C     USAGE
      PROGRAM MAIN
      REAL :: A(5) = [1.0, 2.0, 3.0, 4.0, 5.0]
      WRITE(*,*) 'SUM = ', ARRAY_SUM(A,5)  ! 15.0
      STOP
      END
```

Explanation: - Base case: Empty array ($n = 0$) returns 0 - Recursive: Last element + sum of first $n - 1$ elements - Stack depth equals array size

4. String Length

```
C     RECURSIVE STRING LENGTH
      RECURSIVE INTEGER FUNCTION STRLEN(STR) RESULT(LEN)
      CHARACTER(*), INTENT(IN) :: STR
      IF (STR(1:1) == ' ') THEN
          LEN = 0
      ELSE
          LEN = 1 + STRLEN(STR(2:))
      END IF
      END FUNCTION
```

```
C     MAIN PROGRAM
      PROGRAM MAIN
      WRITE(*,*) 'LENGTH = ', STRLEN('HELLO')  ! 5
      STOP
      END
```

Explanation: - Base case: Empty character returns 0 - Recursive: Count first character + process substring - Handles strings up to 32,767 characters (Fortran limit)

5. Greatest Common Divisor (GCD)

```
C     RECURSIVE GCD
      RECURSIVE INTEGER FUNCTION GCD(A,B) RESULT(RES)
      INTEGER, INTENT(IN) :: A, B
      IF (B == 0) THEN
          RES = A
      ELSE
          RES = GCD(B, MOD(A,B))
      END IF
      END FUNCTION

C     USAGE
      PROGRAM MAIN
      WRITE(*,*) 'GCD(48,18) = ', GCD(48,18)  ! 6
      STOP
      END
```

Explanation: - Base case: $b = 0$ returns a - Recursive: $gcd(b, ab)$ - Implements Euclid's algorithm

6. Tower of Hanoi

```
C     RECURSIVE HANOI SOLUTION
      RECURSIVE SUBROUTINE HANOI(N, FROM, TO, AUX)
      INTEGER, INTENT(IN) :: N
      CHARACTER(*), INTENT(IN) :: FROM, TO, AUX
      IF (N == 1) THEN
          WRITE(*,*) 'Move disk 1 from ', FROM, ' to ', TO
      ELSE
          CALL HANOI(N-1, FROM, AUX, TO)
          WRITE(*,*) 'Move disk ', N, ' from ', FROM, ' to ', TO
          CALL HANOI(N-1, AUX, TO, FROM)
      END IF
      END SUBROUTINE

C     MAIN PROGRAM
      PROGRAM MAIN
```

```
      CALL HANOI(3, 'A', 'C', 'B')
      STOP
      END
```

Output for n=3: 1. Move disk 1 from A to C 2. Move disk 2 from A to B 3. Move disk 1 from C to B 4. Move disk 3 from A to C 5. Move disk 1 from B to A 6. Move disk 2 from B to C 7. Move disk 1 from A to C

7. Binary Search

```
C     RECURSIVE BINARY SEARCH
      RECURSIVE INTEGER FUNCTION BSEARCH(ARR, L, R, X) RESULT(INDEX)
      INTEGER, INTENT(IN) :: ARR(*), L, R, X
      INTEGER :: MID
      IF (R >= L) THEN
          MID = L + (R - L)/2
          IF (ARR(MID) == X) THEN
              INDEX = MID
          ELSE IF (ARR(MID) > X) THEN
              INDEX = BSEARCH(ARR, L, MID-1, X)
          ELSE
              INDEX = BSEARCH(ARR, MID+1, R, X)
          END IF
      ELSE
          INDEX = -1
      END IF
      END FUNCTION
```

Explanation: - Time complexity: $O(\log n)$ - Space complexity: $O(\log n)$ (recursive stack) - Precondition: Array must be sorted

8. Palindrome Check

```
C     RECURSIVE PALINDROME CHECK
      RECURSIVE LOGICAL FUNCTION IS_PAL(STR, L, R) RESULT(RES)
      CHARACTER(*), INTENT(IN) :: STR
      INTEGER, INTENT(IN) :: L, R
      IF (L >= R) THEN
          RES = .TRUE.
      ELSE IF (STR(L:L) /= STR(R:R)) THEN
          RES = .FALSE.
      ELSE
          RES = IS_PAL(STR, L+1, R-1)
      END IF
      END FUNCTION

C     USAGE
```

```
      PROGRAM MAIN
      CHARACTER(5) :: S = 'LEVEL'
      WRITE(*,*) IS_PAL(S, 1, LEN_TRIM(S))  ! .TRUE.
      STOP
      END
```

Explanation: - Base case: $l \geq r$ (empty or single-character string) - Recursive: Compare ends and check inner substring

9. Flood Fill Algorithm

```
C     RECURSIVE FLOOD FILL
      RECURSIVE SUBROUTINE FLOOD_FILL(GRID, X, Y, OLD, NEW)
      INTEGER, INTENT(INOUT) :: GRID(:,:)
      INTEGER, INTENT(IN) :: X, Y, OLD, NEW
      IF (X < 1 .OR. X > SIZE(GRID,1)) RETURN
      IF (Y < 1 .OR. Y > SIZE(GRID,2)) RETURN
      IF (GRID(X,Y) /= OLD) RETURN

      GRID(X,Y) = NEW
      CALL FLOOD_FILL(GRID, X+1, Y, OLD, NEW)
      CALL FLOOD_FILL(GRID, X-1, Y, OLD, NEW)
      CALL FLOOD_FILL(GRID, X, Y+1, OLD, NEW)
      CALL FLOOD_FILL(GRID, X, Y-1, OLD, NEW)
      END SUBROUTINE
```

Explanation: - Base cases: Out-of-bounds or different color - 4-directional recursion - Marks visited cells to prevent infinite loops

10. Iterative Factorial

```
C     ITERATIVE FACTORIAL
      INTEGER FUNCTION ITER_FACT(N)
      INTEGER, INTENT(IN) :: N
      INTEGER :: I
      ITER_FACT = 1
      DO 10 I = 1, N
          ITER_FACT = ITER_FACT * I
10    CONTINUE
      END FUNCTION
```

Comparison: - No stack overflow risk - Constant $O(1)$ space vs recursive $O(n)$ - Faster execution (no function call overhead)