155

Chapter 2

Getting Started

2.1 INTRODUCTION

In the last chapter, we presented the basic building blocks required for programming. So now it is time to introduce the specific details needed to construct simple Fortran programs using some of these basic ideas. To allow you to write your first program, we need to discuss four items:

- How a program is organized
- The different types of data, constants, and variables
- The assignment statement as a means of calculating and storing data
- Simple input and output

We will also review selected library functions for performing common mathematical operations and debugging tips. When you complete this chapter, you should be able to write a simple program consisting of input, sequential calculations and output to a terminal screen.

2.2 PROGRAM ORGANIZATION

A program is constructed with a *text editor*, which is nothing more than a simple word processor. The organization of each program line must follow very specific rules and you must be very careful to follow them. For instance, the various columns have significance as summarized in the table below. Columns 1 through 5 are reserved for statement labels and column 6 is reserved for a continuation character that indicates that the current line is a continuation of the previous line. Columns 7 through 72 store the Fortran commands. Finally, columns 73 and higher are ignored and can be used for comments. Sometimes, you may wish to document various parts of your program. This can be done by using *Comments*, which are created by placing the letter C or an asterisk (*) in column 1. Any text that follows on that line will be ignored.

Column	Purpose
1	C or * in column indicates a comment line
1—5	Statement labels that are used to identify specific lines
6	Continuation character (any character is allowed)
7—72	Fortran program statements
73—	Everything that follows is ignored; can be used for short comments

EXAMPLE 2.1

```
Column→ 10        20        30        40        50        60
        |         |         |         |         |         |
C This Fortran program calculates the area
C of a triangle.
      PRINT *, 'ENTER HEIGHT: '
      READ *, H
      PRINT *, 'ENTER BASE: '
      READ *, B
      A = 0.5 * H * B
C
C The following demonstrates the use of a continuation line
C
      PRINT *,
     1      'AREA= ', A
C
C The above two lines are the same as PRINT *, 'AREA= ', A
C
      END
```

The END statement indicates the end of the *main* program. Later on, we will introduce the concept of *subprograms*, which can be considered as separate programs. Therefore, the END statement is needed to divide the different parts, or *modules*. Without the END statement, we would have no way of knowing where one module ends and next one begins.

Optionally, a program may start with the PROGRAM *name* statement, where you can substitute a convenient label for *name* that indicates the program function. The sole purpose of this name is to make it easier for you to recognize each program at a later date.

EXAMPLE 2.2

```
C The program name (AREAOFCIRCLE) suggests its use
C
      PROGRAM AREAOFCIRCLE
C
C This example calculates the area of a circle.
C
      PRINT *, 'Enter Circle Radius'
      READ *, R
      A = 3.1416 * R * R
      PRINT *, 'Area of circle is ', A
      END
```

2.3 DATA TYPES AND INTEGER CONSTANTS

Fortran 77 contains six *intrinsic* data types that are built automatically into the language. These are divided into two categories: *numerical* and *nonnumerical*. Numerical types are *integer, real, double precision*, and *complex*. Nonnumerical types are *character* and *logical*. In this chapter, we are mostly concerned with the numerical types.

Integer Constants

Integer values are those that represent whole numbers. The range of values that can be represented on a computer depends specifically on the computer. However, a typical range is from -2^{32-1} to $+2^{32-1} - 1$ (approximately $\pm 2 \times 10^9$) for a 32-bit computer.

EXAMPLE 2.3

The following illustrate correct and incorrect examples of integer constants:

Valid Examples	Invalid Examples	Comment
-999		*Negative sign required*
+10		*Plus sign optional*
111 111 111		*Spaces ignored – useful for large numbers*
	111,111,111	*No commas*
	174.00	*No decimal points in integer numbers*
	−7 1/2	*No fractions*

Real and Double Precision Constants

The second type of numerical constant is called *real*. Real numbers are stored in the computer as two components: a mantissa ranging between 0.1 and 1.0 and an exponent that indicates the appropriate power of 10. Real constants are those that we think of as fractional numbers which may be positive or negative and *always* have a decimal point.

EXAMPLE 2.4

The following examples illustrate valid and invalid uses of real constants:

Valid Examples	Invalid Examples	Comment
-21.4		*Negative required*
+132.7		*Plus sign optional*
0.0000034		*Small numbers permitted*
123 456.0		*Spaces ignored*
	$ 1.23	*Only numbers permitted (no $)*
	0	*Requires a decimal point, otherwise this is an integer*
	123,456.00	*No commas*

Real constants can also be used with *scientific notation*. Recall that this is a useful method for noting very large or very small numbers. It relies on the use of a mantissa between 0.1 and 1.0 and an exponent that is a power of 10 and given by <mantissa> × 10 <exponent>. The limit of accuracy of real constants is approximately seven digits with a magnitude from 10^{-39} to 10^{+38}.

EXAMPLE 2.5

The following examples show the use of real constants using scientific notation:

Valid Examples	Invalid Examples	Comment
0.6023E24		*Avogadro's number. 6.023 × 10^{23}*
−0.123E24		*Negative mantissa permitted*
0.123E-24		*Negative exponent permitted*
0.0E0		*Zero!*
1E2		*Decimal point not required*
	0.1E−12.5	*Exponent must be integer*
	0.1E−123	*Value too small on most computers*
	0.1E+123	*Value too large on most computers*

If more than 7 digits of accuracy are required, you can use a *double precision* constant, which is accurate to 14 to 16 decimal places, depending on the machine. Double precision is simple to use. Instead of using E for the exponent, double precision simply substitutes the letter D.

EXAMPLE 2.6

The following examples illustrate double precision constants using scientific notation:

Valid Examples	Invalid Examples	Comment
0.0D0		*Double precision form of zero*
0.23D-94		*Double precision will give greater range*
	0.123456789E23	*Not double precision! Extra digits ignored*

A word of caution: double precision numbers require two to ten times the computational time compared to single precision real numbers. Therefore, you should be careful to use double precision only when absolutely required.

Complex Constants

We often need to use complex numbers such as $4+3i$, which contain real and imaginary parts. Since computers cannot work with imaginary numbers programmers have developed a convention where a complex constant is represented by two real components.

Fortran representation: ($REAL_1$, $REAL_2$)
Algebraic representation: $real_1 + i\,(real_2)$

The first number ($REAL_1$) represents the real part of the complex number, and the second number ($REAL_2$) represents the imaginary part ($i^2 = -1$). The rules for complex constants are not standard for Fortran 77, yet most commercial versions of the Fortran 77 language use a similar set of rules in defining complex data. The following example summarizes these rules.

EXAMPLE 2.7

Here are some examples of commonly encountered complex constants:

Valid Examples	Invalid Examples	Comment
(1.23, -3.45)		*Either component may be negative*
(+1.23, 0.0)		*Positive sign is optional*
(1.23E-2, 3.45)		*Exponential format is permitted*
	(1.23D-128, 3.45)	*Both components must match in precision*
	(1, 2)	*Integers not allowed*

Complex numbers are not officially part of the Fortran 77 standard. However, almost all compilers support them as extensions, so that there are few problems in using them.

Character Constants

There are occasions when we need to work with nonnumerical data, which cannot be handled with the data types just discussed. Examples would be data such as names and addresses. Accordingly, we will use a different type of constant, the *character* constant. A character constant is any set of the allowed symbols defined below and enclosed in single quote marks (').

Letters of the alphabet (upper- or lower-case)
Numbers 0 through 9
Special characters + − ( ) . , * / = ' $
Blank space

Even though you can create symbols such as ☺ on your computer, Fortran will not accept them.

EXAMPLE 2.8

Here are some commonly encountered examples of character constants:

Valid Examples	Invalid Examples	Comment
'Helen'		*Mixing upper/lower case OK*
'12345'		*All numbers OK*
'I''M OK'		*If you want an apostrophe inside the single quotes, you must use two apostrophes. Result is I'M OK.*
	"Helen"	*Must use single quotes (apostrophe)*
	Helen	*Missing quote marks*
	'I ♥ NY'	*Illegal character (♥)*

Be sure not to confuse the character constant '1234' with its numerical counterpart 1234. While it is possible to perform mathematical operations with numerical constants, you cannot do the same thing with numbers stored as character constants. For example, you can add 123 to 456 (both numerical constants), but you cannot write '123' + '456', since these are character constants.

Logical Constants

The final intrinsic data type is the *logical* constant, which can take on only two values. Thus, the rules are very simple since the only allowed values of logical constants are .TRUE. and .FALSE. (note the use of the periods). The role of the logical constant will be made more apparent in the following chapters when we discuss control structures.

EXAMPLE 2.9

Here are some examples of common uses of logical constants:

Valid Examples	Invalid Examples	Comment
.True.		*Mixed case is acceptable*
	FALSE	*Requires periods (.FALSE.)*
	.T.	*Must spell out complete word*

2.4 VARIABLES AND SIMPLE INPUT/OUTPUT

Variables provide a means by which you can manipulate data. By using input and output statements, variables become another way by which you can introduce data into your program. When you studied algebra, you learned that variables could be used to represent a quantity in a formula. In programming, variables have this function also. However, we also use variables to represent memory in the computer. The following example is a program that requests the radius of a circle and returns its circumference and area.

EXAMPLE 2.10

Below is a simple program to compute the area and circumference of a circle of radius *r*. In the program, the variables used are PI, AREA, CIRCUM, and R. Note that we try to choose variable names that indicate their function in the program.

```
      PROGRAM AREAOFCIRCLE
C The following statements request the user to type in
C a value of the radius
      PRINT * , 'Enter circle radius'
      READ * , R
C Once the radius is fed in, the area is calculated
      PI = 3.1416
      AREA = PI * R * R
      CIRCUM = 2 * PI * R
C The value of the area is now printed out
      PRINT * , 'Area of circle is ', AREA
      PRINT * , 'Circumference of circle is ', CIRCUM
      END
```

When we execute this program, the following sequence of events will occur:

```
Enter circle radius                  (Printed by computer — line #4)
5                                    (Value typed in by user —line #5)
Area of circle is 78.54              (Printed by computer — line #11)
Circumference of circle is 31.416    (Printed by computer — line #12)
```

The value of R was entered into the program with a READ Statement and PI was given a value by using a real constant. AREA and CIRCUM were calculated by using simple mathematical expressions. Finally, the calculated values of AREA and CIRCUM were displayed at the terminal screen by using the two PRINT statements in lines 11 and 12.

When you give variables their names, try to choose names that describe their function within the program. The rules for defining Fortran 77 variable names are as follows:

- Names are 1 to 6 characters long
- Only letters (A — Z) and numbers (0 — 9) are allowed
- First character must be a letter
- Upper/lower case are equivalent
- Blank spaces are ignored

EXAMPLE 2.11

Here are some common forms of variable names:

Valid Examples	Invalid Examples	Comment
X		*OK, but not very illustrative*
TAXDUE		*Better, since it describes its function*
TEMP1		*OK to mix letters and numbers*
AMT DUE		*OK, spaces are ignored*
Amt Due		*Same as previous example, since lower case is treated the same as upper case in Fortran*
	AMOUNTDUE	*Too many characters (max of 6)*
	$OWED	*Illegal character ($)*
	2BEES	*Must start with a letter*

Implicit Data Typing

In the previous sections, we discussed the six basic data types, but we did not discuss how to tell the computer how to define the variables. With constants, it was obvious what data type each constant was. For example, if a number had a decimal point it was real, and if it had no decimal point, it was treated as an integer, and so forth. But with variables we must develop another way. With Fortran, we have two options, *implicit* or *explicit* typing.

The variables in Example 2.10 were *implicitly* defined, which means that each was assigned to a data type based on the first letter of the variable name and the following rules:

Variable names that begin with the letters A—H or 0—Z are real.
Variables names that begin with the letters I—N are integer.

EXAMPLE 2.12

Here are some examples of implicit typing:

Variable	Type
R	Real
PI	Real
AREA	Real

Variable	Type
CIRCUM	Real
LENGTH	Integer
ICOUNT	Integer

Explicit Data Typing

Implicit typing rules make it easy to define whether variables will be real or integer. But these rules do not apply to complex, character or logical variables. To use these types, you must use *explicit* typing rules. Explicit typing is simply the procedure of specifying how to treat each variable. These rules are also used if you want to override the implicit typing for integers and reals.

To declare a variable to be a specific type, enter the type followed by a list of the variables to be so treated, with each variable separated by a comma. This so-called *declaration statement* must come before any executable statement where some sort of processing takes place. There may be several declaration statements at the beginning of the program, and their form will always be:

TYPE variable1 , variable2 , . . .

EXAMPLE 2.13

Here are some examples of explicit typing:

Declaration Statement	Result
REAL X , Y , Z	Declares X, Y, and Z as a real variables
REAL LENGTH	Defines LENGTH as a real variable
INTEGER COUNT	Defines COUNT as an integer variable
CHARACTER GRADE	Defines GRADE as a character variable of length 1
CHARACTER*20 NAME	Defines NAME as a character variable of length 20
COMPLEX PHASE	Defines PHASE as a complex variable
LOGICAL YESNO	Defines YESNO as a logical variable
DOUBLE PRECISION X	Defines X as a double precision variable
CHARACTER A*10, B*20	Defines A as a character variable of length 10 and B as a character variable also, but of length 20

The following example shows a full program with explicit variable typing to define the several variables.

EXAMPLE 2.14

The following program is similar to Example 2.10, except that the types of the variables are now explicitly stated:

```
      PROGRAM AREAOFCIRCLE
C The following statements requests the user to type in
C a value of the radius
      REAL R, PI, AREA, CIRCUM
      PRINT * , 'Enter circle radius'
```

(Program continues on next page)


```
      READ * , R
C Once the radius is fed in, the area is calculated
      PI = 3.1416
      AREA = PI * R * R
      CIRCUM = 2 * PI * R
C The value of the area is now printed out
      PRINT * , 'Area of circle is ', AREA
      PRINT * , 'Circumference of circle is ', CIRCUM
      END
```

Simple Input and Output

Most programs require the user to enter data into the program. And once calculations have been performed, the results must be sent to some sort of display device such as a CRT screen. These two functions are known as *input* and *output*, or collectively as I/O. We have already seen examples of I/O in Example 2.10. For the purpose of this section, only free formatted output (also called list directed) will be presented, and we will assume that all I/O will be at the terminal screen.

To input a value to a variable, we use the READ * statement with the general form:

READ *, *variable1, variable2, . . .*

To display the value of a variable or variables on the terminal screen, we use the PRINT * statement, whose general form is:

PRINT *, *variable1, variable2, . . .*

Character constants can be included in the output list of the PRINT * command, by placing the string to be printed inside single quotation marks.

EXAMPLE 2.15

The following example reads in a person's name and age in years. It then converts the age from years into months:

```
      PROGRAM AGEINMONTHS
C The declaration statement must come first
      CHARACTER*10 NAME
      REAL AGEYRS, AGEMTH
C Here is where we input the person's name and age
      PRINT *, 'Enter your name and your age in years'
      READ *, NAME, AGEYRS
C Now we convert the age from years into months
      AGEMTH = AGEYRS * 12
C Print out the results
      PRINT *, NAME, ' is approximately ', AGEMTH, ' months old'
      END
```

This is how the input and output would appear on the CRT screen:

```
Enter your name and your age in years                    (Prompt from line #6)
'Martin C.', 32                                          (Entered by user; note apostrophes)
Martin C.  is approximately 384.000 months old           (Printed by computer from line #11)
```

Note that in the output produced by the computer, any unused characters in NAME are given blank spaces. For example, when the name was entered, it contained only 9 characters. So, when the computer goes to print out the name, there is an extra space that has not been filled. In such cases, the computer will pad the variable with blank spaces.

2.5 ASSIGNMENT STATEMENTS, EXPRESSIONS, AND HIERARCHY

The *assignment statement* is the primary means of storing data in variables. We have seen a number of simple examples of assignment statements in the "AREAOFCIRCLE" program (Example 2.10) and the "AGEINMONTHS" program (Example 2.15). As the name assignment statement implies, we are telling the computer to assign a value to a given variable. The general form of the assignment statement is:

Target ← Value from an expression

The interpretation of this statement is "The target receives a value obtained from the expression." The way this is implemented in Fortran is:

Variable = Value from an expression

The expression on the right-hand side (RHS) of the equal sign can be one of several types as discussed below.

EXAMPLE 2.16

In the table below are several examples of assignment statements involving constants, variables, and mathematical operators:

Expression	Type of Expression
PAY = 5.12	Constant assigned to the variable *pay*
TAXES = CALC	The value of the variable *calc* assigned to *taxes*
PAY = GROSS - NET + 5.00	Value of the numerical expression assigned to *pay*
X = SQRT(Y)	Function used to evaluate the square root of y ($\sqrt{y}$)

In each of these examples, something happens on the right-hand side to determine what value goes into the left-hand side. This is an important difference between an algebraic equation

and an assignment statement. You must keep in mind that these assignment statements are not equations to be solved. Instead, the right hand-side is evaluated first, and the answer is then assigned to the variable on the left-hand side.

EXAMPLE 2.17

In a conventional algebraic equation such as:

$$x = 1 - x$$

we could solve for x very easily and obtain:

$$x = 1/2$$

But, the same line (X = 1 − X) in a program has a very different meaning. It is not an equation to be solved. Rather, it is an expression to be evaluated followed by an assignment of the result to a specific variable. For example, consider the following lines of code (program) and try to predict the final value of X:

```
X = 1.0
X = 1.0 - X
PRINT *, X
```

When the first command (X=1.0) is executed, the real variable X is given a value 1.0. When the second line (X=1.0 − X) executes, the expression is evaluated as 1.0 − (1.0), since the old value of X is retrieved and substituted into the expression. The result, which is 0, is placed into the variable X. X now has the value 0. This process of following the logic of a program is known as *tracing*. It is a very useful device, especially when you are attempting to debug a program. To aid in tracing a program, you should create a table of all the variables in the program, where each row represents one of the variables. Whenever a variable is assigned a value, it is entered into the table and whenever a value is required, the last value entered is used.

EXAMPLE 2.18

Trace through the following program segment and predict its output:

```
X = 1.0
Y = 2.0
Z = 3.0
X = -X
PRINT *, 'Value of X is: ', X
Y = Y - 1.0
PRINT *, 'Value of Y is: ', Y
Z = Z + X
Z = Z + X - Y
PRINT *, 'Value of Z is: ', Z
```

The variable table would look like this after performing the trace:

X	1.0	− 1.0	
Y	2.0	1.0	
Z	3.0	2.0	0.0

Note that the values of X and Y were changed once during the program after the initial assignment, but that Z changed value two times. So, after execution, here is the final output:

```
Value of X is:  -1.000000
Value of Y is:   1.000000
Value of Z is:   0.000000
```

Lines 1, 2, and 3 are constant assignment statements. They initialize the variables X, Y, and Z to the values 1.0, 2.0, and 3.0, respectively. Line 4 tells the computer to take the current value of X (which is 1.0) and change its sign. The value is placed back into X. X is now −1. Line 6 states to take Y and subtract 1 from it, or 2 − 1 is 1. That value is placed back into Y. Y is now 1. Line 8 states to take Z + X and place that value back into Z, or 3 + (−1) is 2. Z is now 2. Line 9 states to calculate Z + X − Y and place the result back into Z, or 2 + (−1) − 1 is 0. Finally, Z is assigned the value 0. Note that during these evaluations the right-hand side is processed first, and then the answer is placed into the variable on the left-hand side.

Expressions and Hierarchy of Operations

All of the examples of expressions have been simple ones. They've consisted simply of multiplication, or addition and subtraction. There are only five basic arithmetic operations possible with Fortran. They are addition, subtraction, multiplication, division, and exponentiation, as presented in the following table:

Priority	Algebraic Symbol	Fortran Symbol	Meaning
1	(.....)	(.....)	Parentheses
2	A^b	**	Exponentiation
3	×	*	Multiplication
3	÷	/	Division
4	+	+	Addition
4	−	−	Subtraction

In order to create mathematical expressions, you must use the symbols listed in the table on a single text line entered into your program. While algebra permits the use of multiple line expressions, Fortran requires you to place the expression on a single line.

For all of the operators with equivalent position within the hierarchy (except **), evaluation is from left to right. For exponentiation, the direction is from right to left. For example 8.0/2.0*4.0 gives 16.0, since the division is done first and then the multiplication.

EXAMPLE 2.19

Here is how you might write a mathematical expression in algebra:

$$y = \frac{a + 2b + c}{d}$$

but in Fortran, this is how we would write the same expression:

```
Y = (A+2*B+C)/D
```

There are several key points that you should notice in this simple example:

- *Implied* operations are not allowed in Fortran. In algebra, we know that 2B means 2 multiplied by B. But, in Fortran, you must explicitly write out the implied multiplication as 2*B.
- Everything is written on one line. In the algebraic expression, the numerator is written above the denominator, like a fraction. But, in Fortran, we place the numerator and the denominator on the same line and separate them with a slash (/) to indicate division.

In algebra it is understood that you should perform the multiplication (2B) before any addition. Thus, in the expression A + 2B + C, 2B is evaluated first and then added to A and C. Therefore, all mathematical operations have a well-defined hierarchy. This is true also for Fortran as summarized in the preceding table.

EXAMPLE 2.20

Based on the hierarchy of mathematical operations, evaluate the following expression:

```
9.2 - ( 2.0**3 - 14.0 / 7.0 ) + 14.0 * 0.1
```

1. First priority is ( ), so the expression inside the parentheses (2.0**3 − 14.0 / 7.0) is evaluated first.
2. Next in the hierarchy is exponentiation. Thus, the expression inside the ( ) is evaluated by performing the exponentiation first, which gives (8.0 − 14.0 / 7.0).
3. The next priority is the division, resulting in (8.0 − 2.0).
4. Finally, perform the subtraction (6.0).
5. Return to the original expression with this result, so the expression becomes 9.2 − 6.0 + 14.0 * 0.1.
6. Next is multiplication and the expression becomes 9.2 − 6.0 + 1.4.
7. Finally, addition and subtraction have the same priority. Therefore, they are evaluated left to right, which gives 3.2 + 1.4 = 4.6.

EXAMPLE 2.21

When two exponentiation operations appear together, they are evaluated right to left:

```
2 ** 3 ** 2    →    2 ** 9    →    512
```

Here are some more examples to see if you fully understand the rules for evaluating an expression.

EXAMPLE 2.22

For the examples below, we supply the answer. Trace through each and make sure you get the same result:

Expression	Value	Comments
16.0 − 4.0 − 2.0	10.0	*Left to right*
16.0 − (4.0 − 2.0)	14.0	*Evaluate expression within ( ) first*
16.0 + 4.0 * 2.0	24.0	*Multiplication first*
16.0 / 4.0 / 2.0	2.0	*Left to right*
16.0 ** 4.0 * 2.0	131072.0	*Exponentiation first*
16.0 ** (4.0 * 2.0)	4294967296.0	*Expression within ( ) first*

In the preceding examples, we have been careful to make all the constants and variables real. This was done because there are special rules that govern integer arithmetic. Also, when you try to mix real and integer data types, complications arise as illustrated in the following section.

2.6 INTEGER AND MIXED-MODE ARITHMETIC

When performing arithmetic with real numbers, the results show what you would algebraically expect. However, when performing calculations with integers or a mixture of integers and real numbers, different results may be obtained.

The two operations that are affected by data type are division and exponentiation. Division of two integers results in an integer value. This value is equal to the real number result with the decimal portion deleted.

EXAMPLE 2.23

The result can be very different if we do mathematics with real numbers and integers. Note in the following example that the integer division produces an unexpected result:

Using reals: 3.0 / 2.0 = 1.5 (note that 3.0 and 2.0 are real as is 1.5)

Using integers: 3 / 2 = 1 (not 1.5! Note that 3 and 2 are integers as is 1)

In the second example above, both 3 and 2 were integers because we did not use decimal points. Therefore, when the computer does the division, it will give an integer result. This is obtained by *truncating* any noninteger remainder.

You must be careful when using integer arithmetic, since unintended results can creep into your program. So be careful! But sometimes this effect is desired, as shown in the next example.

EXAMPLE 2.24

The following program makes change in terms of dollars, quarters, dimes, nickels, and pennies by making use of integer division:

```
      INTEGER CENTS, DOLLAR, QUARTR, NICKEL, DIME, PENNY
      PRINT *, ' Enter value in cents '
      READ *, CENTS
C
C  Whole dollar part is the integer division of CENTS by 100
C
      DOLLAR = CENTS / 100
C
C  What is left is the remaining change in CENTS
C
      CENTS = CENTS - DOLLAR * 100
C
C  Repeat this process for QUARTR, DIME, NICKEL and PENNY
C
      QUARTR = CENTS / 25
      CENTS = CENTS - QUARTR * 25
      DIME = CENTS / 10
      CENTS = CENTS - DIME * 10
      NICKEL = CENTS / 5
      PENNY = CENTS - NICKEL * 5
C
C  Print out the results
C
      PRINT *, 'Dollars: ', DOLLAR
      PRINT *, 'Quarters: ', QUARTR
      PRINT *, 'Dimes: ', DIME
      PRINT *, 'Nickels: ', NICKEL
      PRINT *, 'Pennies: ',PENNY
      END
```

We took advantage of integer arithmetic in this example to give us the desired results. For example, if the change were 78 cents, division by 25 would produce exactly 3 (not 3 plus a remainder). Thus, the program would say to return 3 quarters.

The second area where the results will depend on whether you use reals or integers is exponentiation. Under certain circumstances, an error will occur depending on the choice of variable type. This is because the method of performing the calculation is different depending on

whether the exponent is integer or real.

In the case where the exponent is an integer, the value is determined by successive multiplications. But when the exponent is a real number, Fortran will take the log of the base, multiply the result by the exponent, and then take the inverse log of that result. This may cause the computer to take the logarithm of a negative number, which produces an error.

EXAMPLE 2.25

Try to use integer values for exponents. Otherwise, Fortran will use logarithmic functions to calculate the result.

2**6 is calculated as 2*2*2*2*2*2 = 64

but

2**6.0 is calculated as $\text{Log}^{-1}(6*\text{Log}(2)) = 64.0$

The result is the same, but there *may* be a problem, as shown below:

(−2)**3 is calculated as −2*−2*−2 = −8

but

(−2)**0.3 is calculated as $\text{Log}^{-1}(0.3*\text{Log}(-2)) =$ ERROR since log of a negative number is not defined.

In situations where integer and real numbers are mixed during division, the integer value is converted to a real number. The result is the expected value. What must be realized is that these rules for integer and mixed-type division are applied on an operator by operator basis.

EXAMPLE 2.26

Evaluate the following mixed-mode arithmetic expression:

J = 2.3 * (3 / 2) − 5

First evaluate 3 / 2:

3 / 2 = 1 *(Fraction is truncated because both numbers are integers)*

Next perform multiplication. An integer times a real yields a real number.

J = 2.3 − 5

Finally perform the subtraction. A real minus an integer yields a real.

J = −2.7 = 2 *(Fraction is truncated because J is an integer)*

2.7 SELECTED LIBRARY FUNCTIONS

Fortran functions behave much like the definitions for mathematical functions such as square root, sine, and so forth. You use a function by placing its name (followed by its arguments) in an expression. You must take great care to match the type, number and order of arguments required for the function. We summarize the most common functions below.

Name	Description	Argument	Result	Example
ABS(X)	absolute value	integer	integer	J = ABS(−51)
		real	real	X = ABS(−17.3)
		double	double	Z = ABS(−0.1D04)
ACOS(X)	arccosine	real	real (rad)	X = ACOS(0.5)
		double	double (rad)	X = ACOS(0.5D0)
ALOG(X)	natural logarithm	real	real	X = ALOG(2.71828)
		double	double	X = ALOG(0.2718D01)
ALOG10(X)	logarithm base 10	real	real	X = ALOG10(10.0)
		double	double	X = ALOG10(0.1D0)
AMAX(...)	returns largest value	integer	integer	I = AMAX(5,1,6,2)
		real	real	X = AMAX(0.2,5.6)
		double	double	X = AMAX(1D0,3D3)
AMIN(...)	returns smallest value	integer	integer	I = AMIN(4,3,−4)
		real	real	X = AMIN(0.2,5.6)
		double	double	X = AMIN(1D0,3D3)
ASIN(X)	arcsine	real	real (rad)	X = ASIN(0.5)
		double	double (rad)	X = ASIN(0.5D0)
ATAN(X)	arctangent	real	real (rad)	X = ATAN(1.0)
		double	double (rad)	X = ATAN(1.0D0)
COS(X)	cosine	real (rad)	real	X = COS(1.04712)
		double	double	X = COS(1.04712D0)
DBLE(X)	converts to double	integer	double	X = DBLE(3)
		real	double	X = DBLE(3.0)

(table continues on next page)

Name	Description	Argument	Result	Example
EXP(X)	exponential, e^x	real double	real double	X = EXP(1.0) X = EXP(1.0D0)
INT(X)	converts to integer	real double	integer integer	J = INT(3.9999) J = INT(0.3999D01)
FLOAT(I)	converts to real	integer double	real real	X = FLOAT(4) X = FLOAT(0.4D01)
MOD(I,J)	integer remainder of I/J	integer	integer	J = MOD(29,4)
NINT(X)	round to nearest integer	real double	integer integer	J = NINT(3.99) J = NINT(0.6D01)
REAL(I)	convert to real	integer double	real real	X = REAL(3) X = REAL(0.23D02)
SIN(X)	sine	real (rad) double (rad)	real double	X = SIN(0.5202) X = SIN(0.52D0)
SQRT(X)	square root	real double	real double	X = SQRT(17.6) X = SQRT(0.17D2)
TAN(X)	tangent	real (rad) double	real double	X = TAN(0.785) X = TAN(0.785D0)

Be careful when using the trigonometric functions, since they require angles in radians (rad), not degrees. Similarly, the inverse trigonometric functions will report the results in radians.

EXAMPLE 2.27

The following program reads in two points and calculates the distance between them:

```
C Distance Between Two Points (X1,Y1) and (X2,Y2)
C
      PRINT *, 'Enter X,Y location for first point'
      READ *, X1, Y1
      PRINT *, 'Enter X,Y location for second point'
      READ *, X2, Y2
      DIST = SQRT ( ( X2 - X1 ) ** 2 + ( Y2 - Y1 ) ** 2 )
      PRINT *, 'Distance between the points is ', DIST
      END
```


Once the values of (X_1, Y_1) and (X_2, Y_2) have been entered, they are used in the equation to determine the distance d by the formula $d = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$. Note that we have used the Fortran function SQRT to perform this calculation.

It is permissible to place one function within another. In fact, many times common sense and defensive programming practices will require it. For example, if you are going to take the square root of a number, you may have to make sure that the number is positive before you can attempt the square root. Otherwise, you may be asking the computer to perform an illegal operation.

EXAMPLE 2.28

Here is the program of Example 2.27 that has been modified to take the absolute value of a number before attempting to take the square root of a number:

```
C Distance Between Two Points (X1,Y1) and (X2,Y2)
C
      PRINT *, 'Enter X,Y location for first point'
      READ *, X1, Y1
      PRINT *, 'Enter X,Y location for second point'
      READ *, X2, Y2
      DIST=SQRT(ABS((X2-X1)**2 + (Y2-Y1)**2))
      PRINT *, 'Distance between the points is ', DIST
      END
```

In this situation, it made no difference that we added the ABS function, since the argument $(x_1-x_2)^2+(y_1-y_2)^2$ is always positive (or zero). So taking the absolute value makes no difference. But there will be many occasions when this might be needed.

2.8 DEBUGGING TIPS

Debugging is the process of removing errors from your program. For the types of programs presented in this chapter, two error types are most likely to occur; improper use of integer/mixed-mode arithmetic and simple typographical errors.

You may also have programs that contain *run-time* and *logic* errors. Run-time errors are those that occur while the program is running, and can usually be traced to illegal mathematical operations such as the logarithm of a negative number. Logic errors are those where the program executes to completion, but gives you the wrong answer. You simply gave the computer a wrong series of instructions to execute. Both of these types of errors are most easily solved by tracing.

Typographical errors (typos) occur when you accidentally mistype the name of a variable or command. Since Fortran utilizes implicit typing, new variables can be created when you make a simple typo error. You can detect such mistakes by:

- Including the statement IMPLICIT NONE at the beginning of your program.
- Declaring all the variables that you intend to use in the program by explicit typing.

By disabling the implicit typing feature, variables that are not declared will result in a compiler error. This is one way of locating variables created by "typos."

EXAMPLE 2.29

Here is an example of how to use the IMPLICIT NONE statement to help locate typographical errors. Note that if we turn off the implicit typing feature by using the IMPLICIT NONE statement, we will then have to declare every variable with the explicit typing.

```
C The IMPLICIT NONE statement is placed first in the program
C We must then explicitly name all the variables in a type
C declaration statement(s)
      IMPLICIT NONE
      REAL X1, Y1, X2, Y2, LENGTH
      READ *, X1, Y1, X2, Y2
      LENGHT=SQRT((X2-X1)**2 + (Y2-Y1)**2)
      PRINT *, 'Length is ', LENGTH
      END
```

On line 7 LENGTH is misspelt as LENGHT. Because implicit typing is turned off, the compiler will generate an error due to an undeclared variable. If the IMPLICIT NONE statement is omitted, the computer would accept both spellings as different variables and will report no error.

Errors due to mixed-mode arithmetic can often be located by adding PRINT statements before and after each expression. Before an expression, print out the variables being used in the calculation. After an expression, PRINT the result. Trace the program and see if the values being printed out at each step of the program agree with what you expect.

EXAMPLE 2.30

This program always returns the result "Length is 1.000000," no matter what values are entered. Find the error.

```
IMPLICIT NONE
REAL X1, Y1, X2, Y2, LENGTH
READ *, X1, Y1, X2, Y2
PRINT *, X2, X1, Y2, Y1
LENGTH=((X2-X1)**2 + (Y2-Y1)**2)**(1/2)
PRINT *, 'Length is ', LENGTH
END
```

We have added PRINT statements *before* and *after* the calculation of LENGTH. The computer will always report "LENGTH=1.000000," but when you trace through by hand you should get a different result. By adding the PRINT statements, we find that the program line that calculates LENGTH is in error. The problem is the improper use of integer arithmetic in the exponentiation (** (1/2)). Because 1/2 involves integer division, the result is 0. This problem can be fixed by simply adding decimal points, producing (**(1./2.)).

Solved Problems

2.1 The following examples illustrate various types of literal constants. Some examples are valid, while other are invalid. Where appropriate, we indicate the default data type (Real, Integer, etc). Explanations for the invalid examples are provided.

a)	1.00	*(Real)*
b)	123	*(Integer)*
c)	+8	*(Integer)*
d)	13 7/8	*(Invalid: Fraction is not allowed)*
e)	$ 78.24	*(Invalid: $ is not allowed)*
f)	−0.123E7.1	*(Invalid: Exponent must be an integer)*
g)	'She''s happy'	*(Character)*
h)	(1.2, 3.4)	*(Complex)*
i)	(1E0, 3E0)	*(Complex)*
j)	1,234	*(Invalid: commas are not allowed.)*
k)	.False	*(Invalid: must have trailing "." to be logical)*

2.2 The following are examples of valid and invalid variables. Reasons for invalid examples are provided.

a)	X	*(Valid)*
b)	Height	*(Valid)*
c)	R P M	*(Valid; Spaces ignored)*
d)	1station	*(Invalid: 1st character must be letter)*
e)	.TEST	*(Invalid: "." not allowed)*

2.3 The following examples demonstrate implicit data typing based on the first letter of the variable name.

a)	X	*(Real)*
b)	Volume	*(Real — note that lower case letters are OK)*
c)	KOUNT	*(Integer)*
d)	InDia	*(Integer — mixing lower/upper case letters OK)*

2.4 The following examples demonstrate explicit data typing.

a)	REAL X, Y, Z	*(Valid, defines three real variables)*
b)	INTEGER COUNT	*(Valid, defines one integer variable)*
c)	CHARACTER NAME*20	*(Valid, defines one variable 20 characters long)*
d)	LOGICAL OK	*(Valid, defines one logical variable)*
e)	DOUBLE PRECISION VOL	*(Valid, defines one double precision variable)*