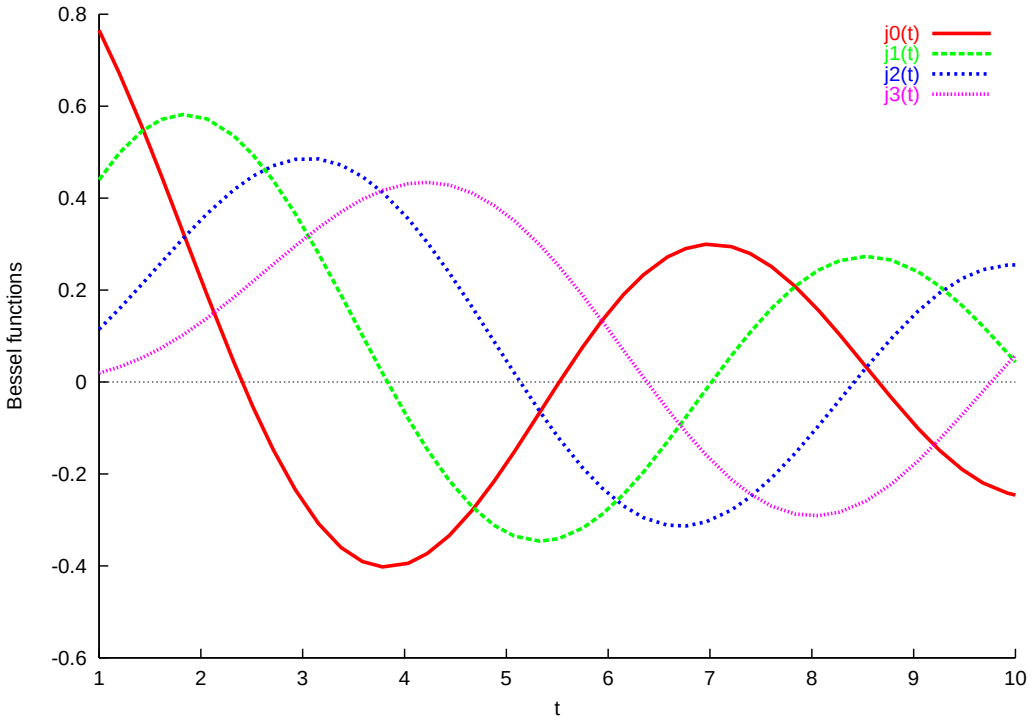
C for Engineers and Scientists

— An Interpretive Approach

Harry H. Cheng
University of California, Davis



Chapter 24

‡Introduction to Fortran

Originally developed in the mid-1950s for scientific computing, FORTRAN was the first general-purpose high-level computer programming language. Its name was derived from *FORmula TRANSlation*, meaning that it was intended for translating scientific formulas into computer codes. Although the original FORTRAN language was limited and contained many flaws, further developments and updates have improved it dramatically. The last major update was Fortran 90, which incorporated all the facilities of FORTRAN 77 along with a few other extensions. The spelling of the language was also changed in Fortran 90.

Many features in Ch are added to bridge the gap between C and Fortran for scientific numerical computing. References, complex numbers, generic functions, and variable length arrays in Ch are similar to those in Fortran. For example, linguistic features of references in Ch are closely related to equivalence statements, arguments in subroutines and functions in Fortran. Many programming features such as array syntax and assumed-shape arrays of Fortran 90 have been incorporated into Ch. Users with prior Fortran experience can easily adapt to the Ch programming paradigm. This chapter gives a brief introduction to Fortran 90 and discusses some issues related to port Fortran code to Ch.

24.1 Getting Started

A Fortran program can be formatted in the *fixed-source form* or *free source form*. FORTRAN 77 uses the fixed-source form. Fortran 90 and newer versions support both fix-source form and free source form.

In the fixed-source form, columns 1 through 5 are reserved for statement labels. A letter 'C' or an asterisk `*` in column 1 specifies that it is a comment line. Column 6 is normally blank. If any character other than a blank or zeo is placed in that column, the statement is interpreted as a continuation of the previous statement. Columns 7 to 72 contains the Fortran instructions. Columns 73 to 80 are ignored by the compiler.

Similar to C code, the free source format has no restriction. Any characters, placed after an exclamation point `!`, are considered as comments which do not affect the code of a Fortran program. An ampersand `&` at the end of a line is used to separate a single statement into multiple lines. In this book, Fortran code is written in the free source form.

As an introduction to programming with Fortran, consider Program 24.1. It is similar to the "Hello, World!" program in C. The first line of the program is comment line, indicated by '!' on the 1st column. Program 24.1 simply displays the string "`Hello, World!`" on the terminal screen. The program block is enclosed between the `PROGRAM` and `END PROGRAM` statements. Fortran is case insensitive. The `WRITE` statement is used to display the "Hello, World!" message, which will be discussed in detail in later sections.

```
! File name: helloworld.f90

PROGRAM helloworld
! Display the output
WRITE (*,*) 'Hello, World!'
END PROGRAM helloworld
```

Program 24.1: "Hello, World!" program in Fortran.

24.2 Constants and Variables

A Fortran *constant* is defined as a data object whose value does not change throughout the execution of a program. In contrast, a Fortran *variable* is a data object whose value can change during the execution of a program. Similar to C, the value of a variable may or may not be initialized when it is declared. A variable name may have up to 31 characters consisting of any combination of alphanumeric characters and the underscore `'_'` character. Note, however, that the first character of a variable name must be alphabetic. Also note that Fortran is case insensitive. That is, uppercase letters are equivalent to lowercase letters. For example, `A` is equivalent to `a`, and `FORTRAN` is equivalent to `Fortran`. In this book, names for keywords and generic functions of Fortran use capital letters.

There are five built-in data types in Fortran: **INTEGER**, **REAL**, **COMPLEX**, **LOGICAL**, and **CHARACTER**. Types **INTEGER**, **REAL**, and **COMPLEX** are for numerical values; type **LOGICAL** is for logical values; and **CHARACTER** is for characters and strings. Data types **INTEGER**, **REAL**, **LOGICAL**, and **CHARACTER** will be discussed in this section, whereas data type **COMPLEX** will be described in a later section. The formats for declaring data types are as follows,

```
type :: name
type :: name = val
```

where the variable `name` has the data type corresponding to `type`. Unlike C, there is no semi-colon required at the end of a statement.

Additionally, Fortran supports user-defined data types similar to **struct** in C. However, this is beyond the scope of this chapter and will not be covered.

24.2.1 Constants

An *integer constant* is a whole number that can be positive, negative, or zero. A *real constant* is any floating-point number. Like integer constants, real constants can be positive, negative, or zero. Additionally, real constants can be written with or without an exponent. No commas may be embedded in either integer or real constants. Examples of integer and real constants are shown below.

```
0  -999  +111  5678           ! integer constants
-10.0  35.  1.2345  6.78E+1    ! real constants
```

Note that contents after the exclamation point `!` are treated as comments.

There are only two possible *logical constant* values: `.true.` and `.false.`, which correspond to logical true and false, respectively. The two periods before and after the two logical values are required to distinguish them from variable names.

A *character constant* is defined as a string of characters enclosed in either single ' or double " quotes. The minimum number of characters allowed in a string is 1, whereas the maximum number of characters is implementation defined. Examples of character constants are as follows.

```
'This is a test string.'
"This is also a test string."
'1.23456'
" "                          ! single blank space
```

Names can also be attributed to constants by using the **PARAMETER** attribute of a type declaration statement in the following format,

```
type, PARAMETER :: name = val [, name2 = val2, ...]
```

where `type` corresponds to the data type of the constant and `name` is the constant name assigned to constant value specified by `val`. This is similar to defining a macro in C. Note that multiple constant names may be declared on a single line. Below are examples of declaring constant names for π, the gravitational constant 9.81, and an age limit of 21 years old.

```
REAL, PARAMETER :: pi = 3.14159, g = 9.81
INTEGER, PARAMETER :: age_limit = 21
```

24.2.2 Integer variables

Integer variables are variables that contain values of the **integer** data type. The number of bytes required to store an integer variable in memory is implementation defined, but is typically 2 or 4 bytes. Examples of declaring integer variables are shown below.

```
INTEGER :: time, distance
INTEGER :: age = 10
```

Aside from explicitly declaring integer variables, any variables beginning with the letters i through n are assumed to be of type **INTEGER** by default. Thus, variable `n` is assumed to be of **INTEGER** type by default. It can be used without declaration inside a program.

24.2.3 Real variables

Real variables are those that contain values of **REAL** data type, which is equivalent to **float** in C. Real numbers are usually stored in 4 or 8 bytes of memory. Declaring variables of **REAL** types is similar to declaring integer variables. For example,

```
REAL :: sum, quotient
REAL :: init_val = 24.9
```

Also like integer variables, any variable name beginning with a letter other than i, j, k, l, m, and n are assumed to be of type **REAL** by default. For example, variable `size` is assumed to be of **REAL** type if it was not explicitly declared otherwise.

Double Precision

By default, floating-point variables declared by the **real** type qualifier are only *single precision* variables. However, there are some cases when a variable with *double precision*, equivalent to **double** in C, is required to handle a programming problem. Although the definition of single and double precision floating-point numbers are implementation dependent, it will be assumed that the size of a single precision variable will correspond to 32-bits, or 4-bytes, of memory, and the size of a double precision variable will correspond to 64-bits, 8-bytes, of memory. In order to specify double precision variables, the **kind** type parameter can be used. The kind of a real variable can be specified within a pair of parentheses after the **REAL** type qualifier, such as

```
REAL(KIND=4) :: var1        ! single precision
REAL(KIND=8) :: var2        ! double precision
```

where the phrase `"KIND="` is optional. Note that `KIND=4` refers to a 4-byte, single precision variable, and `KIND=8` corresponds to a 8-byte, double precision variable.

24.2.4 Logical variables

As the name describes, a *logical variable* is one that consists of a logical value. They are commonly used along with logical expressions to control program execution. Logical variables may be declared as follows.

```
LOGICAL :: condition
LOGICAL :: test = .true.
```

24.2.5 Character variables

Character variables are variables of the **CHARACTER** data type, equivalent to **char** in C, which are used to store characters and strings. As stated earlier, characters and strings are typically enclosed in a pair of single or double quotes. The beginning quote of a string must match its ending quote. That is, a string beginning with a single quote `'` cannot be ended by a double quote `"`. For example, the following character string is not valid.

```
'This is invalid."
```

However, if a character string must include an apostrophe inside a pair of single quotes, then the apostrophe can be represented by two consecutive single quotes. Otherwise, if the character string is enclosed inside a pair of double quotes, a single quote can be used without any special handling, which is similar to C. Likewise, double quotes may be used in a character string enclosed by a pair of single quotes. For example, the following are valid Fortran character strings.

```
'It''s done.'
"It's done."
'The show "MASH" is good.'
```

Below is the extended format for declaring character variables in Fortran,

```
CHARACTER(len=<len>) :: var1, var2, ...
```

where `(len=<len>)` is optional and `<len>` is an integer value specifying the number of characters in the variable(s). If `(len=<len>)` is not included, then the variables of the declaration statement have a character length of 1. Another way to specify the length of a character variable is to just simply include an integer value. For example, the following are all valid character variable declarations.

```
CHARACTER(len=10) :: first_name, last_name
CHARACTER :: middle_initial
CHARACTER(15) :: hometown
```

Note that declaring character variables is similar to using the **char** data type and arrays to declare characters and strings in C. The equivalent form of the above declarations in C is as follows.

```
char first_name[10], last_name[10];
char middle_initial;
char hometown[15];
```

24.3 Arrays and Matrices

Similar to C/Ch, Fortran supports the use of arrays and matrices. An array in Fortran can be declared by using the **DIMENSION** attribute, which specifies the extent(s) of an array's dimension(s). An array may be of the four data types: **INTEGER**, **REAL**, **LOGICAL**, **CHARACTER**, or **COMPLEX**. Examples of array declarations are shown below.

```
INTEGER, DIMENSION(5) :: a
REAL, DIMENSION(2,3) :: b
```

In the above example, `a` is declared as a one-dimensional 5-element array, whereas `b` is a 2×3 dimensional array. When declaring a two-dimensional array, the first subscript refers to the number of rows, and the second refers to the number columns.

Like in C, Fortran arrays can be initialized when they are declared. This is done by using the delimiters `(/` and `/)` to specify an *array constructor*. For example, the following statement declares array `a` and initializes its 5-elements.

```
INTEGER, DIMENSION(5) :: a = (/ 1, 2, 3, 4, 5 /)
```

Furthermore, arrays can be initialized in their declaration statements with an *implied do-loop*. For example, the following statement initializes the 10-element array `a` with values 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

```
INTEGER, DIMENSION(10) :: a = (/ (i, i=1,10) /)
```

Thus, `a(1) = 1`, `a(2) = 2`, `a(3) = 3`, etc.

The limitation of the array constructor is that it can only represent one-dimensional arrays. Thus, initializing a multiple dimensional array requires the use of the **RESHAPE()** function, which changes the shape of an array without changing the number of elements. It has the following form,

```
output = RESHAPE(array1, array2)
```

where `array1` contains the data to reshape and `array2` is a one-dimensional array specifying the new shape. The **RESHAPE()** function returns the newly reshaped array. For example,

```
INTEGER, DIMENSION(2,3) :: b(2,3) = &
         RESHAPE((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /))
```

Note that the ampersand & is used to separate a single statement into multiple lines in Fortran. Arrays in Fortran are assigned and stored *column-wise* as shown in Figure 10.13. This contrasts C arrays, which are associated *row-wise* as shown in Figure 10.2. Thus, array b in the above declaration has the following form.

$$b = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Different elements of an array can be accessed through subscripts in Fortran. In contrast to C, which uses a pair of square brackets [] to specify array subscripts, Fortran utilizes a pair of parentheses. For example, a(i) refers to the ith element of array a in Fortran. Another difference between C and Fortran arrays is that, by default, the first element of a Fortran array has a subscript of 1, whereas the first element of a C array has a subscript of 0. Similar to Ch, however, the lower and upper bounds of a Fortran array can be specified explicitly. For example,

```
REAL, DIMENSION(0:5) a       ! equivalent to float a[0:5] in Ch
REAL, DIMENSION(0:5,10:15) b ! equivalent to float b[0:5][10:15]
                             ! in Ch
```

24.4 Complex Numbers

Fortran supports the use of complex numbers similar to Ch. Complex numbers can be declared in Fortran by using the **COMPLEX** type declarator. For example,

```
COMPLEX :: z1
COMPLEX, DIMENSION(10) :: z2     ! an array of complex numbers
```

Furthermore, variables of **COMPLEX** types can be initialized similar to other Fortran data types. Complex numbers are initialized with a complex constant, which consists of two numbers separated by a comma and enclosed in a pair of parentheses. For example, the following statement initializes complex variable z with a value of $1 + i2$.

```
COMPLEX :: z = (1, 2)    ! equivalent to complex z = complex(1, 2)
                         ! in Ch
```

Fortran also includes some functions specifically for complex numbers. Similar to **complex()** in Ch, function **CMPLX()** with the form

```
CMPLX(a, b)
```

converts to real or integer values into a complex number with real and imaginary parts equal to a and b, respectively. Also, functions **REAL()**, or **INT()**, and **AIMAG()** can be used to obtain the real and imaginary parts of a complex number, respectively. The **INT()** function returns an integer value of the real part. To obtain the magnitude of a complex number, function **cabs()** can be used. These functions are similar to the **real()**, **imag()**, and **cabs()** functions in C.

Double Precision

Similar to a double precision floating-point variable, a double precision complex variable can be declared in the same manner, since a variable of complex type is actually the combination of two real numbers. For example, a single and double precision complex variables can be declared as follows.

```
COMPLEX(KIND=4) :: complex_var1         ! single precision
COMPLEX(KIND=8) :: complex_var2         ! double precision
```

The double precision **REAL** and **COMPLEX** data types in Fortran are equivalent to the **double** and **double complex** data types in C.

24.5 Operators

Aside from the assignment operator `'='` and unary plus `'+'` and minus `'-'` operators, Fortran also have operators for arithmetic, relational, and logical operations. Tables 24.1 - 24.3 lists the arithmetic, relational, and logical operations available in Fortran along with their C equivalents. Note that C does not have equivalent forms of the `**` and `.NEQV.` operators. However, function **pow()** in Ch has the same functionality as the exponential operator `**`. For example, 2^3 is equal to `2**3` in Fortran and `pow(2, 3)` in Ch. Similarly, the C relational operator can be used to perform the same test is Fortran's logical equivalence operator `.EQV.`, which returns true only if both operands are false or both are true. For example, `logic_op1 .EQV. logic_op2` in Fortran is equivalent to `logic_op1 == logic_op2` in C. Fortran's logical non-equivalence operator `.NEQV.` is similar to the exclusive-or operator `^^` in Ch. The operation results in true only if one operand is true, while the other is false.

Table 24.1: Fortran Arithmetic Operators

Operator	Fortran Syntax	C Syntax
addition	+	+
subtraction	-	-
multiplication	*	*
division	/	/
exponential	**	pow(x,y)

Table 24.2: Fortran Relational Operators

Operator	Fortran Syntax	C Syntax
equal to	==	==
not equal to	/=	!=
greater than	>	>
greater or equal	>=	>=
less than	<	<
less or equal	<=	<=

Table 24.3: Fortran Logical Operators

Operator	Fortran Syntax	Ch Syntax
logical AND	.AND.	&&
logical OR	.OR.	||
logical equivalence	.EQV.	==
logical non-equivalence	.NEQV.	^^
logical NOT	.NOT.	!

24.6 Control Flow

Table 24.4 lists the syntax for the various control flows for both Fortran and C. Syntax for the **IF**, **IF-ELSE**, and **IF-ELSE IF-ELSE** constructs are similar to those of the C programming language. Also, the **SELECT** construct is identical to the **switch** statement in C. Note that Fortran applies the **DO** construct to perform the same functionalities as the for- and while-loops on C. One noticeable difference between the syntax for the two programming languages is that C uses a pair of brackets `{}` to specify the beginning and end of a control flow, while Fortran uses **END IF**, **END SELECT**, and **END DO** statements.

Fortran also offers the **CYCLE** and **EXIT** statements to manipulate the operations within the DO-loop. Similar to the **continue** statement in C, executing the **CYCLE** statement in the body of a loop will stop execution of the body and return control to the beginning of the loop. Note, however, the loop index (`i` in DO-loop of Table 24.4) will be incremented, and execution will only resume if the exit condition has not been met. The **EXIT** statement, however, will completely stop execution of the loop body and transfer control to next executable statement following the loop, which is similar to the **break** statement in C.

Table 24.4: Control flow and selection comparisons between Fortran and C

Description	Fortran Syntax	C Syntax

IF	`IF (expr1) THEN`
	`  statements`
	`END IF`
		`if(expr1) {`
		`  statements`
		`}`

IF-ELSE	`IF (expr) THEN`
	`  statements1`
	
	`ELSE`
	`  statements2`
	`END IF`
		`if(expr1) {`
		`  statements1`
		`}`
		`else {`
		`  statements2`
		`}`

IF-ELSE IF-ELSE	`IF (expr1) THEN`
	`  statements1`
	
	`ELSE IF (expr2) THEN`
	`  statements2`
	
	`ELSE`
	`  statements3`
	`END IF`
		`if(expr1) {`
		`  statements1`
		`}`
		`else if(expr2) {`
		`  statements2`
		`}`
		`else {`
		`  statements3`
		`}`

CASE	`SELECT CASE (expression)`
	`CASE (const-expr1)`
	`  statements1`
	
	`CASE (const-expr2)`
	`  statements2`
	
	`CASE DEFAULT`
	`  statements3`
	
	`END SELECT`
		`switch(expression) {`
		`  case const-expr1:`
		`  statements1`
		`  break;`
		`  case const-expr2:`
		`  statements2`
		`  break;`
		`default:`
		`  statements3`
		`  break;`
		`}`

DO-loop	`DO 100 i=n1,n2`
	`  statements`
	`100 CONTINUE`
	
	`DO i=n1,n2,n3`
	`  statements`
	`END DO`
	
	`DO`
	`  IF(expr) THEN`
	`  statements`
	`  ELSE`
	`  EXIT`
	`  END IF`
	`END DO`
		`for(i=n1; i<=n2; i++)) {`
		`  statements`
		`}`
		
		`for(i=n1; i<=n2; i+=n3)) {`
		`  statements`
		`}`
		
		`while(expr) {`
		
		`  statements`
		
		`}`

24.7 Formatted Input and Output

24.7.1 The WRITE statement

The **WRITE** statement is used to print out data to a computer terminal. The default form of the **WRITE** statement is as follows.

```
WRITE (*,*) output_list
```

The contents of `output_list` are characters and strings and/or values of the integer, real, complex, and logical data types. For example, the following code fragment

```
INTEGER :: i
i = 5
WRITE (*, *) "i =", i
```

will print out `"i = 5"`.

Like C, Fortran allows the user to specifically define the format of the output. All that needs to be done is to replace the second asterisk inside the pair of parentheses following the **WRITE** statement with an integer constant that is typically 3 digits long. For example, the statements below indicate that the **WRITE** statement will have the format corresponding to the **FORMAT** statement associated with label `100` with the format specifier listed in Tables 24.5 and 24.6.

```
REAL :: x
x = 1.23456789
WRITE (*,100) x
100 FORMAT ("The value of x is ", F6.3)
```

The output of the above code fragment is as follows.

```
The value of x is  1.234
```

Similar to the format specifier `"%f6.3"` in C, format specifier `F6.3` in Fortran indicates that a floating-point value should be printed out with a maximum width of 6 characters, and that only three digits should be printed to the right of the decimal point.

Table 24.5 lists and describes the format descriptors available in Fortran. The meanings of the symbols used in this table are explained in Table 24.6.

Table 24.5: Format specifiers in Fortran.

Specifier	Description
*r*I*w* (or *r*I*w.m*)	output an integer
*r*F*w.d*	output a floating-point number
*r*E*w.d*	output a floating-point number in exponential notation
*r*ES*w.d*	output a floating-point number in scientific notation
*r*L*w*	output logical value
*r*A (or *r*A*w*)	output character or string
*n*X	insert horizontal spacing
T*c*	insert horizontal tab
/	newline (equivalent to `'\n'` in C)

Table 24.6: Symbols used in Table 24.5.

Symbol	Description
c	column number
d	number of digits to the right of the decimal point
m	minimum number of digits to be displayed
n	number of spaces to skip
r	number of times to use a descriptor or group of descriptors
w	field width

24.7.2 The READ statement

The **READ** statement is used to obtain data from the input buffer associated with an input device, which is typically the computer keyboard. The **READ** statement has the default form shown below.

```
READ (*,*) input_list
```

Input values have to correspond to the data types of the variables in the `input_list`. For example, the following statement reads in 3 integer values and stores them into integer variables `i1`, `i2`, and `i3`.

```
READ (*,*) i1, i2, i3
```

Just like the **WRITE** statement, it is also possible to format a **READ** statement. Format specifiers used for the **READ** statement is the same as those used for the **WRITE** statement. For example, the following code fragment can be used to read three integers with a field width of 6 characters.

```
READ (*,100) i1, i2, i3
100 FORMAT (3I6)
```

24.8 File Processing

File processing in Fortran involves the use of the **OPEN** and **CLOSE** statements for opening and closing a file, respectively. The general form of the **OPEN** statement is as follows,

```
OPEN(UNIT=io_num; FILE="filename"; STATUS="stat";
     ACTION="permission"; IOSTAT=status)
```

where `io_num` corresponds to the I/O unit number associated with a file, `filename` is the file name of file to be opened, `stat` is the status of the file, `permission` corresponds to the open permission of the file, and `status` is for debugging purposes used to determine whether opening the file was successful or not. The I/O unit number equivalent to the file pointer `FILE *` in C. The `status` of the file can be either `'OLD'` for an already existing file, `'NEW'` for specifying a new file, `'REPLACE'` for specifying a new file to be created or overwrites a file if it already exists, `scratch` for creating a temporary file similar to the **tmpfile()** in C, and `UNKNOWN` for an implementation dependent status. The value of `permission` can be `'READ'` for read permission, `'WRITE'` for write permission, and `'READWRITE'` for both read and write permission. As stated before, the last specification `IOSTAT=status` is for error handling, where the value of `status` is an integer. If the **open** statement is successful, a zero will be assigned to `status`; if it is not successful, a positive number corresponding to a system error message will be returned.

The **CLOSE** statement can be used to close a file and release the I/O unit number associated with it. Its general form is as follows,

```
CLOSE(UNIT=io_num)
```

where `io_num` corresponds to the I/O unit number of the file.

After a file is successfully opened for reading and/or writing, the **READ** and **WRITE** statements can be used to read from or write to the file, respectively. Recall that the general form of the **READ** and **WRITE** statements are as follows,

```
READ (*,*)
WRITE (*,*)
```

where the the second asterisk can be replaced to customize the format of the **READ** and **WRITE** statement. Replacing the first asterisk with a file's I/O unit number will associate the corresponding statement to the file. For example, the following code fragment

```
OPEN(UNIT=4, FILE='info.dat', STATUS='old', IOSTAT=status)
READ (4, *) i1, i2, i3
```

opens file `info.dat` and reads three integers from it. In contrast, the code fragment below creates a new file `info2.dat` to write the values of three floating point numbers to the file.

```
OPEN(UNIT=8, FILE='info2.dat', STATUS='new', IOSTAT=status)
WRITE (8, 100) f1, f2, f3
100 FORMAT ('f1 = ', F10.2, ' f2 = ', F10.2, ' f3 = ', F10.2)
```

Fortran also provides two statements for moving around a file. They are the **BACKSPACE** and the **REWIND** statements. The **BACKSPACE** and **REWIND** statements has the following forms,

```
BACKSPACE(UNIT=io_num)
REWIND(UNIT=io_num)
```

where `io_num` corresponds to the I/O unit number associated with a file. The **BACKSPACE** statement moves the file position back one position. On the other hand, the **REWIND** statement places the file position back at the beginning of the file, similar to function **rewind()** in C.

24.9 Functions and Subroutines

Like C and other programming languages, Fortran provides a library of generic functions as well as offers users the capability to define their own subroutines or functions. These features allow programs to be more efficient and portable. This section will discuss generic functions and how to define subroutines and functions in Fortran as well as comparing functions in Ch and Fortran.

24.9.1 Function Definition

User-defined functions are split up into two categories in Fortran: *subroutines* and *functions*. A Fortran subroutine is similar to a Ch function using arguments of reference to pass more than one value. It is invoked through its name by a **CALL** statement. The input and passed variables of a subroutine are contained within an argument list. Its general form is as follows.

```
SUBROUTINE subroutine_name(arg_list)
  (declaration statements)
  (execution statements)
  RETURN
END [SUBROUTINE subroutine_name]    ! [...] is optional
```

Every subroutine begins with the **SUBROUTINE** statement and ends with the **END** or **END SUBROUTINE** statement. A subroutine is an independent programming unit in Fortran. This means that it not only can be invoked by a Fortran program, but also by another subroutine.

Similar to a Fortran program, a Fortran subroutine consists of a declaration section and an executable section. In the declaration section, variables of the argument list are declared to be either input or output arguments with the optional **INTENT** attribute. For example, consider the following subroutine.

```
SUBROUTINE my_subroutine(arg1, arg2, arg3)

  ! Declarations.
  REAL, INTENT(in)    :: arg1
  REAL, INTENT(out)   :: arg2
  REAL, INTENT(inout) :: arg3    ! REAL, INTENT(in out) :: arg3

  ...
  return
end subroutine
```

The above subroutine definition specifies that `arg1` is an input argument, `arg2` is an output argument, and `arg3` is used as both an input and output argument. Using the **INTENT** attribute to specify whether an argument in a subroutine argument list is an input argument, an output argument, or both an input/output argument is good for debugging purposes as well as provides clearer coding for reading of a program. Calling subroutine `my_subroutine` can be done as follows.

```
CALL my_subroutine(arg1, arg2, arg3)
```

Unlike subroutines, user-defined Fortran functions will return one output value. Its general form is shown below,

```
! FUNCTION func_name(arg_list)
! data_type :: func_name
! or
data_type FUNCTION func_name(arg_list)
  (declaration statements)
  (execution statements)
  func_name = retval
  RETURN
END [FUNCTION func_name]
```

where `[FUNCTION func_name]` indicates that the descriptor **FUNCTION** and function name may be added to the end of the **END** statement for documenting purposes. Function definitions must begin with the **FUNCTION** statement and end with the **END** or **END FUNCTION** statement. Fortran functions are invoked by using their names in an executable statement. An example of a user-defined function is shown below.

```
INTEGER FUNCTION addition(a, b)
  IMPLICIT NONE

  INTEGER :: a, b

  addition = a + b

  RETURN
END FUNCTION addition
```

The above code defines a function call `addition()`, which returns the sum of its two integer arguments `a` and `b`. This Fortran function is equivalent to the C function `addition()` in Program 6.1 in Chapter 6. Note that the **IMPLICIT NONE** statement indicates that default data type for variables will be turned off and all variables have to be explicitly declared before they are used.

24.9.2 Recursive Subroutines and Functions

Like C, Fortran 90 allows a subroutine to call itself or a function to use itself inside the function definition. To specify a recursive subroutine or function, the keyword **RECURSIVE** needs to be prefixed to the subprogram. For example, the recursive C function for calculating a factorial of an integer in Program 6.15 can be written in Fortran as follows.

```
RECURSIVE INTEGER FUNCTION factorial(n)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: n

  IF (n <= 1) THEN
    factorial = 1
  ELSE
    factorial = n*factorial(n-1)
  END IF

  RETURN
END FUNCTION factorial
```

24.9.3 Generic Functions

Like C, Fortran contains a library of commonly used generic function. They are invoked inside Fortran programs and subroutines similar to the user-defined functions. Some of these functions are listed below.

ABS() SIN() COS() TAN() ASIN() ACOS() ATAN() ATAN2() CEILING() EXP() FLOOR() LOG() LOG10() MAX() MIN() MOD() SQRT() SUM() TRANSPOSE() ...

```
! File: function.f90

SUBROUTINE func(a, m, n, r)
    INTEGER :: m, n
    COMPLEX, DIMENSION(m,n) :: a
    COMPLEX :: r
    INTEGER :: i, j
    DO i = 1, m
        DO j = 1, n
            a(i,j) = r*r + 3
            a(i,j) = SIN(a(i,j)/r)
        END DO
    END DO
END SUBROUTINE

PROGRAM function
COMPLEX, DIMENSION(5,6) :: za
COMPLEX :: z
z = CMPLX(1,2)
CALL func(za, 5, 6, z)
END
```

Program 24.2: A Fortran program

These functions have the same functionalities as those of the Ch equivalent functions. They can be used to handle arguments of different data types.

24.10 Call-by-Reference in Ch and Fortran

In Fortran, arguments are passed by reference. Functions in Ch can be called either by value or by reference. In Fortran, the data types of actual arguments of a function should be the same as those of the formal arguments of the function. In Ch, the data types of actual arguments of functions can be different from those of the corresponding formal arguments of references in functions. In Fortran, when an argument of a subroutine is used as an lvalue inside a subroutine, the actual argument in the calling subroutine must be a variable. A reference variable in Ch can be used as an lvalue inside a function even if the actual argument is not an lvalue, whereas Fortran cannot.

Note that when arrays are passed to a function through its arguments, the memory space for arrays in the calling function will be used in the called functions. In other words, arrays in Ch are passed by reference. Since Ch encapsulates all the programming capabilities of Fortran 90, porting subroutines and functions in Fortran to functions in Ch is not very difficult. For example, the Fortran program given in Program 24.2 can be ported, without changing the functionality of the program, to a Ch program shown in Program 24.3. In Program 24.3, the shape 5x6 for the complex array `a` inside the function `func()` is assumed from its actual argument of the complex array `za`. Integral values 5 and 6 are passed to arguments of references `m` and `n`. The complex variable `z` is passed to the function by reference. Block-structured nested do-loops are used inside the function. Like many other mathematical functions, the function `sin()` in Ch is a polymorphic function that calculates the result according to the data type of the input argument. In this example, a

```
/* File: function.c */

void func(complex a[:][:], int &m, int &n, complex &r) {
  int i, j;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++) {
      a[i][j] = r*r + 3;
      a[i][j] = sin(a[i][j]/r);
    }
  r = 50;
}

complex za[5][6], z;
z = complex(1,2);
func(za, 5, 6, z);
```

Program 24.3: A Ch program ported from a Fortran program.

complex value will be produced by the function `sin()`.

For Program 24.2, the subroutine `func()` is defined similar to function `func()` in Program 24.3 in C. The Fortran equivalent of the C **main()** function is the main program block enclosed inside the **PROGRAM** and **END** statements. A Fortran 90 program typically uses a file extension `.f90`. For example, Program 24.2 is named `function.f90`. A parameter following the keyword **PROGRAM** is required. The file name without the file extension is typically used. For example, `function` is used in Program 24.2.

An Application Example

Program 24.4 is the Fortran equivalent of Program 6.3 for the application example in Chapter 6. It defines a function `force()` for calculating the magnitude of the applied load, which is invoked within the Fortran program. Note that function `force()` is declared as a **REAL** data type in the main program `accelfun`. This is due to the fact that function `force()` is defined after the main program, which is similar to function prototyping in C. Note also that the words `"PROGRAM accelfun"` following the **END** statement of the main program are optional similar to those in subroutines and functions. After the acceleration of the system is calculated, the result is printed out with a generic **WRITE** statement. The output of Program 24.4 is the same as that for Program 6.3.

Different commands are typically used for different Fortran compilers that conform to different Fortran standards. For example, commands `f77, f90, f95, f03` in Unix are used for Fortran compilers conforming to FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003 language standards, respectively. At the same time, a Fortran program conforming to different Fortran language standards may use different file extensions. A FORTRAN program conforming to the FORTRAN 77 language standard has the file extension `.f` or `.for`, whereas a Fortran program with file extension `.f90, f95`, or `.f03` conforms to Fortran 90, Fortran 95, or Fortran 2003, respectively.

To compile a Fortran program in Unix using the Fortran 95 compiler, simply type `f95` followed by its file name. Program 24.4 with file extension `.f90` conforms to the Fortran 90 standard. It can be compiled by a compiler for Fortran 90, Fortran 95, or Fortran 2003. The command below

```
> f95 accelfun.f90
```

```
! File: accelfun.f90
PROGRAM accelfun

IMPLICIT NONE

! Function declaration.
REAL :: force

! Variables declaration.
REAL, PARAMETER :: g = 9.81
REAL :: a, mu, m, p, t

! Calculate acceleration.
mu = 0.2
m = 5
t = 2
p = force(t)
a = (p-mu*m*g)/m

! Display the result.
WRITE (*,*) 'Acceleration a = ', a, '(m/s^2)'

END PROGRAM accelfun


REAL FUNCTION force(t)
    IMPLICIT NONE

    REAL :: t
    REAL :: p

    p = 4*(t-3)+20
    force = p

    RETURN
END FUNCTION force
```

Program 24.4: The equivalent Fortran program for `accelfun.c`.

compiles Program 24.4 using a Fortran 95 compiler. If no compilation error occurs, an executable file `a.out` will be created. File `a.out` can be run by typing its name on the command prompt, such as

```
> a.out
Acceleration a =    1.238000      (m/s^2)
```

For a Windows operating system, Fortran compilers typically incorporates a graphical-user-interface (GUI) for the user to interactively work with. Depending on the compiler, there is usually a **build** menu bar for the user to select the option to compile and run a Fortran program.