SCIENTIFIC PROGRAMMING IN FORTRAN 2003

A tutorial Including Object-Oriented Programming

Katherine Holcomb
University of Virginia

1 Forward

This tutorial covers the basics of Fortran 2003, along with a few 2008 features. It is intended to be an introduction to the language, not a complete reference. It does not discuss features of Fortran 77 that are deprecated in Fortran 2003, nor does it cover all aspects of Fortran 2003; it is presented merely with the hope that it will be useful to beginners learning the language. Knowledge of basic programming concepts is assumed but otherwise the information should be self-contained.

Note that the correct spelling is Fortran, not FORTRAN. This has been the case at least since Fortran 77.

In older Fortran the layout was rigid and the column position of a character could have meaning. This is referred to as *fixed format* code and reflects Fortran's beginnings on 80-column punched cards. Beginning with Fortran 90 *free format* was introduced, in which column position does not matter. This tutorial covers only free format. However, fixed or free format does not determine whether the code is Fortran 77 or Fortran 90+; it is possible, though unusual, to write Fortran 2003 in fixed format. Most compilers now support Fortran 77 fixed format for backwards compatibility and there is no distinction between a "Fortran 77" and a "Fortran 90/95/2003" compiler anymore. In general the programmer must specify to the compiler in advance whether the source file contains fixed or free format. This is accomplished by some compiler-specific means, usually by the suffix on the file name or sometimes by a compiler option. For most Unix-based compilers a suffix of `.f90` indicates free format (regardless of whether the code is actually 2003 or even 2008 Fortran) and `.f` indicates fixed format.

2 Fundamentals

2.1 Program Units

A Fortran program consists of one or more **program units**.

The unit containing the `PROGRAM` attribute is often called the *main program* or just *main*. The main program should begin with the `PROGRAM` keyword. Unlike some other languages, this keyword for the beginning of the main program unit is not required in Fortran, but its use is highly recommended.

Example

```
PROGRAM myprog

    non-executable statements

    executable statements

END PROGRAM myprog
```

Unlike many other programming languages, Fortran is *not* case-sensitive; that is,

```
program myprog

Program Myprog

PROGRAM myprog
```

and so forth, are all the same to the compiler. This is true for any name or statement in the program. In this document, we will tend to write keywords in upper case, but this is for clarity only and is not required in actual code.

Other types of program unit include **functions**, **subroutines**, and **modules**. These will be covered in detail later. All program units must terminate with an `END` keyword. The unit type and name may optionally follow the `END`.

```
END PROGRAM myprog
```

Program units other than modules may also contain one or more `STOP` statements. This statement causes execution to cease. The `STOP` keyword may be followed by a message to indicate why or where the termination took place.

```
STOP

STOP "Divide by zero attempted. Execution terminated in subroutine DIV."
```

As a matter of programming style, the `STOP` statement should be used only for abnormal terminations. Much old code still uses it routinely in the main program unit but it is not required unless execution must be ceased before the next `END` statement is reached.

2.2 Literals and Variables

Variables are the fundamental building blocks of any program. In Fortran, a variable name may consist of up to 31 alphanumeric characters, of which the first character must be a letter. Underscores are allowed but not spaces or special characters.

2.2.1 Variable Types

Computers divide the entities on which they operate into **types**. The internal representation of different types is quite distinct and with some exceptions they are not interchangeable. Most compiled programming languages require that the type of each variable be specified. The fundamental generic types are integer, floating point, character or string, and sometimes bit or byte and/or Boolean (also known as logical).

Loosely speaking, an integer variable is represented by a bit indicating whether it is positive or negative, plus some number of bits (usually 31) that represents the number in base 2, i.e. the binary number system.

The basis of floating-point numbers is scientific notation. The internal representation of floating-point numbers consists of a sign bit, a specific number of bits denoting the exponent, and the remaining bits devoted to the *mantissa* (fractional part) of the number. The mantissa is normalized to be greater than or equal to zero and less than one. The precision of the number is a function of the number of bits used for the exponent (in base 2) and the mantissa. Essentially all modern computers use a convention called the IEEE 754 standard, which assumes a 4-byte (32-bit) *word* in single precision and a 64-bit, two-word representation in double precision. Since the number of available bits is finite, the set of floating-point numbers is in turn finite and thus is an imperfect representation of the infinite set of mathematical real numbers. In the IEEE 754 standard, single precision represents numbers in the range of approximately 10^{-45} to 10^{38} with 7 to 8 decimal digits of accuracy, whereas double precision ranges from about 10^{-323} to 10^{308} with approximately 14 decimal digits representable. On most newer computers the word size is 64 bits but single precision is still 32 bits and double precision is 64 bits; in this case single precision is half a word and double precision occupies one word.

Operations that are mathematically illegal, such as division by zero, result in a *floating-point exception*. In IEEE 754, the result of an illegal operation is represented by `NaN`, which stands for Not a Number. The standard defines that any operation on a `NaN` produces another `NaN`. With each such operation throwing an exception, generation of `NaN`s causes the program to run slowly, as well as producing nonsensical results. Therefore, illegal operations should be avoided. Some languages provide facilities to *trap* floating-point exceptions, making it possible to handle them or at least to exit gracefully, but unfortunately the Fortran does not. Some compilers offer options that will cause the execution to cease if an exception is encountered, however. Fortran 2003 specifies a means by which the programmer may insert code to check for floating-point exceptions manually but it is only a partial implementation of the IEEE exception handling. We will cover this feature after we have discussed modules.

The values of characters are generally assumed to be represented by the ASCII system, in which one byte is used per character. This covers 127 characters. Some programming languages (including recent Fortran) can use other systems, such as Unicode, but ASCII is the base representation.

Operations on the different internal types are quite distinct and for this reason, the types should never be mixed without explicitly converting one to another. For example, because integer types cannot represent fractions, division of one integer by another drops the remainder. This can result in results that are surprising to the novice programmer, such as the fact that in computer arithmetic, `2/3 = 0`.

Fortran defines several distinct types corresponding to the generic types described above. These include

`INTEGER`

`REAL`

`DOUBLE PRECISION` (this name is obsolescent but is still widely used)

`COMPLEX`

`CHARACTER`

`LOGICAL`

The `REAL` type is a single-precision floating-point number. The `COMPLEX` type consists of two reals (most compilers also provide a `DOUBLE COMPLEX` type). Most arithmetical and mathematical operations that are defined for reals and doubles are also defined for complex numbers. The `LOGICAL` type is equivalent to the `Boolean` of some other languages; logical variables can assume only the two values `.true.` and `.false.` and are used primarily in conditionals. Only a few restricted operations are possible on logicals and the programmer does not need to be concerned with their internal representations, which can vary.

2.2.2 Explicit and Implicit Typing

For historical reasons, Fortran is capable of *implicit typing* of variables. When this is used, variables with names beginning with the letters `I` through `N` are integers; all others are real. If not told otherwise, the compiler will assume implicit typing. Implicit typing can lead to bugs that are difficult to find and for this reason its use is not recommended. To cancel implicit typing and force all variables to be explicitly declared, use the statement `IMPLICIT NONE` in each unit. This statement must precede all declarations. In older code other types of `IMPLICIT` statements are often seen; for example, to declare all variables except default integers to be double precision, the statement used was

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

However, new code should always disable implicit typing since that will allow the compiler to catch most typographical errors in variable names.

2.2.3 Literal Constants

Constants are literal representations of a type that are specified by the programmer. In Fortran, different constant formats are required for each type.

Integer

Integers are specified by a number without a decimal point.

Examples

```
137

-56894

42

900123
```

Commas should not be inserted.

Real

Reals (single precision) may be specified either by adding a decimal point, or by using scientific notation with the letter *e* indicating the exponent.

Examples

```
19.

3498.12

6.0221415e23

-13.01
```

Double Precision

Double precision constants must be specified in exponent notation with the letter *d* indicating the exponent. In the absence of the *d* exponent, the constant is real *no matter* how many decimal positions are specified, and it will be truncated if the decimals are more than a real variable can represent. It is important for the programmer to understand that in most programming languages, the default floating-point literal is double precision whereas in Fortran it is single precision. Numerical errors can result when literals are not specified with the `d` exponent.

Examples

```
15d00
```

```
9.375d-11

27.8734567807d3
```

Complex

Complex constants consist of two single-precision constants enclosed in parentheses. The first constant is the (mathematically) real part; the second is the imaginary part.

Examples

```
(1.0,0.0)

(-270.,15.5)

(0.0,-10.)
```

For those systems that support `double complex`, the floating-point constants must use the `d` notation.

```
(1.0d0,0.0d0)
```

Character

Character constants are specified by enclosing them in single quotes.

Examples

```
'This is a character constant.'

'The answer is 42.'

'15.3'
```

Note that a character constant representing a number cannot be stored into a numerical variable; it must be cast appropriately. This will be discussed in the section on input and output.

If an apostrophe is to be part of the constant, it should be represented by a double quote.

```
'All the world"s a stage.'
```

Logical

Logical constants can take only the values `.true.` or `.false.` with the periods part of the constant.

2.2.4 Variable Declarations

Most compiled languages are *statically typed*; a variable is brought into existence with a particular type and that type cannot be changed subsequently in the program. Hence variables must be *declared* before they can be used. To declare a variable, preface its name by its type. A double colon may (and sometimes must) follow the type. This is the new form and its use is recommended for all declarations, whether or not it is required. In

Fortran variable declarations must precede all executable statements; that is, a variable may not be declared immediately before it is used.

Numeric Variables

Examples

```
INTEGER  ::  i, j

REAL     ::  r, s, t

COMPLEX  ::  z, w
```

A variable can be assigned a value at its declaration:

```
INTEGER  ::  i = 2, j
```

In the above example the value of `i` is loaded into the variable at compile time and can be changed. If the keyword `parameter` is included, the assigned value *cannot* be changed subsequently (an attempt to do so will generate an error):

```
INTEGER, PARAMETER  ::  i = 2
```

Character

Character variables are declared with the `character` type; the length is supplied via the keyword `len`. The length is the maximum number of letters or digits that will be stored in the character variable. In Fortran, all character variables must be of fixed length, although they may be padded with blanks. (There are a few exceptions to this rule which we shall ignore for the time being.)

```
CHARACTER(LEN=16)    :: a
```

Logical

Logical variables are declared with the `logical` keyword.

```
LOGICAL              :: l, flag = .false.
```

Initialization with Intrinsics

In Fortran 2003 variables may be initialized with intrinsic functions, e.g.

```
REAL, PARAMETER      ::  pi=atan(1.0)
```

Even some compilers that still do not implement much of the 2003 standard will support this functionality as an extension to Fortran 95.

2.2.5 Variable Kind

The current preferred method of specifying the precision of a variable is to use the `KIND` representation. With `KIND`, the programmer specifies the number of decimal digits and the exponent range required, and the system matches the request to its underlying machine representation as closely as possible. Since nearly all current systems use the IEEE 754 convention, in practice only two kinds are widely used.

Compilers still recognize the `REAL` and `DOUBLE PRECISION` keywords. The advantage to the kind convention is that it can be made very simple to switch between real and double precision.

For most purposes, it is only necessary to use the `SELECTED_REAL_KIND` intrinsic and the programmer need never worry about what is going on within the machine. To declare single precision, use

```
INTEGER, PARAMETER  ::  rk = SELECTED_REAL_KIND(6,37)
```

Double is

```
INTEGER, PARAMETER  :: rk = SELECTED_REAL_KIND(15,307)
```

Floating-point variables can now be declared with statements such as

```
REAL (rk)           ::  r, s, t
```

and the precision will be determined by which `selected_real_kind` statement precedes the declarations.

2.2.6 Variable Operations

Fortran defines a number of operations on each type of variable.

The basic operators, defined for numeric types, are the arithmetic operators (addition, subtraction, multiplication, and division) and the negation operator. Unlike many other languages, Fortran also has an exponentiation operator (x raised to the y power). It should be noted that arithmetic operations on the set of floating-point numbers do not possess the mathematical properties of the same operations on the abstract real numbers; in particular, floating-point arithmetic is not commutative ($a+b \neq b+a$ in general).

Integer Operators

Unary

`-` : negation

Binary

`Arithmetic operations (+ - * /)` : (addition, subtraction, multiplication, division). The result of these binary arithmetic operations is another integer; in the case of integer division, only the quotient is returned.

`N**M` : `N` raised to the `M`th power.

Floating Point Operators

Unary

- : negation.

Binary

Arithmetic (**+ - * /**) : return another floating-point number of the same type.

xm** : x raised to the mth power, where m is an integer.

xy** : x raised to the yth power. This is normally evaluated using logarithms, and is slower than raising a floating-point number to an integer. Some compilers recognize expressions like x**2.0 as raising a real number to an integer power, but most do not. Illegal operations, such as raising zero to a negative number, will throw a floating-point exception.

Character Operators

// : concatenation (joins two character strings)

Substrings can be extracted from character variables by a colon operator along with a specification of the bounds:

```
words(i:j)
```

This expression references all the characters in words from the ith position to the jth position. A single character within a longer string must be referenced with

```
words(i:i)
```

If the lower bound is omitted the beginning of the string is assumed; similarly if the upper bound is not specified the end of the character variable is assumed.

```
words(:j)
words(i:)
```

Logical Operators

Unary

.not. : negation.

Binary

.and. : intersection.

.or. : union (evaluates to .true. if either operand is true).

.eqv. : logical equivalence.

.neqv. : exclusive or (evaluates to .true. only if exactly one operand is true).

Relational Operators

In addition to operators on specific types, there is a set of binary *relational operators* that take any two numeric or character variables of the same type and kind and return a logical result. These operators are used in *conditionals* to construct tests.

`.eq.` or **`==`** : equality.

`.neq.` or **`/=`** : non-equality.

`.lt.` or **`<`** : less than.

`.gt.` or **`>`** : greater than.

`.le.` or **`<=`** : less than or equal to.

`.ge.` or **`>=`** : greater than or equal to.

Only `.eq.` and `.ne.` are valid for complex types. When applied to character variables, the comparison is taken based on the ASCII collating sequence.

Operator Precedence

Expressions are evaluated by strict progression from left to right. If an ambiguity occurs, a predefined ordering of the precedence of operators is followed. Exponentiation is first, followed by multiplication/division, then negation, and finally addition/subtraction. (Logical operators, used in control statements, are beneath numerical operations and have their own ranking.) The ordering may be changed by parentheses; programmers should make liberal use of parentheses to be sure their intentions are carried out by the compiler. For example, the expression

```
I*J+K
```

is potentially ambiguous. The compiler will evaluate this left to right and with `*` taking precedence over + so that the expression is `I` multiplied by `J` and then `K` is added. This ordering can be changed to `I` times the sum of `J` plus `K` by the use of parentheses:

```
I*(J+K)
```

Logical operators also have a well-defined precedence; in order, they are `.not.`, `.and.`, `.or.`. Last are `.eqv.` and `.neqv.` which have equal precedence, meaning that an expression with both operators and no parentheses will be evaluated left to right.

All relational operators have the same precedence.

2.2.7 Expressions, Statements, and Assignments

An **expression** is a combination of symbols that forms a valid unit for evaluation or computation.

Examples

```
x + 1.0
97.4d0
sin(y)
x*aimag(cos(z+w))
```

A **statement** is a complete instruction. Statements may be classed into two types, *executable* and *non-executable*. Non-executable statements are those that the compiler uses to determine various fixed parameters, such as module `use` statements, variable declarations, function interfaces (prototypes), and data loaded at compile time. Executable statements are those which are executed at runtime. With a few exceptions such as the `format` statement, all non-executable statements must occur before the first executable statement.

A statement is normally terminated by the end-of-line marker. If a statement is too long for a line, it may be continued by ending the line with an ampersand (`&`). The standard line length before a continuation is required is 80 characters. Many compilers accept line lengths of up to 132 characters, although sometimes a compiler option is required for longer lines.

Multiple statements may be written on the same physical line if they are separated by semicolons. The final statement should not end in a semicolon in this case.

Examples

```
z = sin(y)

a = b+c/d

do i=1,10
var1 = qa * ( qb**3 * ( az + bz ) ) + qc * ( qd**2 * ( ax - bx ) ) &
   + qe * ( wa + cos(wb) )

A = 0; B = 0; C = 0
```

Note that spaces ("whitespace") are ignored, except for what is needed to separate keywords. Some keywords can be written with or without spaces between tokens; specifically, `ENDDO` and `END DO` are the same, `ENDIF` and `END IF` are the same, and so forth.Programmers are encouraged to make liberal use of whitespace to increase program clarity. The tab is not a standard Fortran character and its use can lead to problems in moving source files from one editor or development environment to another, so it should be avoided although most compilers accept it.

Assignment statements assign an expression to a quantity. An assignment statement uses the equals sign (=), but this symbol does not have the same meaning as it has in mathematics. In Fortran, the statement

x = x + 1.0

means that the value `x` is to be incremented by 1 and the result stored into the memory location reserved for the quantity represented by `x`. Most computer programming languages use this convention, with true equality being represented by a different symbol.

In general, a quantity that may appear on the left-hand side of an assignment statement is called an *lvalue*. Not every expression may be an lvalue; the statement

```
x + 1.0 = y
```

is *not* a valid assignment in Fortran.

2.2.8 Comments

Comments are strings inserted into the code that are ignored by the compiler but may be of use to the programmer and others using the code. In free-form Fortran the comment is indicated by the `!` symbol. All characters following an exclamation point are ignored.

Programmers are encouraged to make liberal use of comments to explain what the code is doing, to note special cases, to indicate the meaning of default values, and so forth.

Examples

```
!*********************************************************
! This program computes a solution to the Lapace equation *
! Author: I. Programmer                                   *
! Date: 2012/08/20                                        *
!                                                         *
! Important variables                                     *
!  A : the solution                                       *
!  x : the x variable                                     *
!  y : the y variable                                     *
!                                                         *
!*********************************************************

INTEGER  ::  i = 10 ! Initialize loop index to 10 at compile time
```

2.2.9 Variable Type Conversions

Due to the differences in internal representation of the primitive variable types, operations on variables must take the correct type for each operand and must return a result of the same type. However, it is frequently the case that a numeric variable of one type, for example an integer, must be used in an operation with a variable of another type; this is called a *mixed* expression. For the operation to occur, a type conversion must occur. Conversion of one variable type to another is often called *casting*. In many cases, the compiler will perform an implicit cast; a variable of lower-ranked type is *promoted* to the highest-rank type in the expression. For example, in the expression `2*3.1416`, most compilers will convert the integer to a floating-point number automatically. Correct form calls for the programmer to cast types explicitly, using one of the intrinsic functions described in the next section. In this example, the programmer should write `real(2)*3.1416`. Of course, more typically the casts will be performed on variables and not on literals, since literals can be easily entered with the correct type.

The ranks of numeric variable type, from lowest to highest, are integer, real, double precision, and complex. Logicals cannot be cast to any other type. Characters can be converted to numeric types, and vice versa, through the use of internal reads and writes, which will be covered as part of the discussion of input and output.

2.3 Statement Labels and Names

A statement may carry a **label** that uniquely identifies it by a number. The number must consist of one to five digits, at least one of which must be nonzero. Leading zeros are ignored. The label must occupy the first field of the statement and at least one blank must separate it from the body of the statement. Any statement may be labeled, but in practice the most common labeled statement is the `CONTINUE` statement. This statement is a placeholder, i.e. a "no op" (no operation) statement. It does nothing but tell the compiler to continue on to the next statement. However, it is very useful for clarity whenever a jump or break occurs.

`FORMAT` statements *must* be labeled. These statements are described in the section on input and output.

Certain statements may be *named*. The name may be anything that would be valid as a Fortran variable name. The name occupies the first field and a colon separates it from the body of the statement.

Examples

```
100  continue

2100 format(6i)

outer: do i=1,20
   end do outer
```

2.4 Intrinsics

Many operations are implemented in the form of *intrinsic functions*. These intrinsics are functions that are defined by the compiler; true to its role as a numerical language, Fortran provides a wide range of mathematical functions. Most of these intrinsics are *overloaded*; that is, if they take numeric types as arguments then they accept all floating-point types and sometimes integer as well. Most numerical intrinsics also accept complex arguments provided it makes mathematical sense to do so.

The following examples give only a sampling of the large number of intrinsic functions available. Please see a language reference for a complete list. Several conversion intrinsics can take an optional integer kind argument but this is usually omitted below.

Integer Intrinsics

`mod(i,j)` : returns the remainder of the division i/j

`real(i)` : returns the real version of the integer argument (casts integer to real)

`real(i,k)` : returns real kind k of integer

`abs(i)` : absolute value of `i`

`max(i,j)` : returns the larger of `i` and `j` (can take a variable number of arguments)

`min(i,j)` : returns the smaller of `i` and `j` (can take a variable number of arguments)

Floating Point Intrinsics

int(a) : returns integer part of floating-point number (cast real to integer)

dble(a) : casts real to double precision (equivalent to `real(a,k)` with appropriate `k`)

real(d) : casts (truncates) double precision to single precision

abs(a) : absolute value

ceiling(a) : returns the smallest integer greater than or equal to its argument

floor(a) : returns the largest integer less than or equal to its argument

max(a,b [,c…]) : returns the maximum of the list of arguments

min(a,b [,c…]) : returns the minimum of the list of arguments

mod(a,b) : returns the remainder of the division `a/b`

exp(a) : exponential (base e) of `a`

log(a) : natural logarithm (base e) of `a`

log10(a) : common logarithm (base 10) of `a`

sqrt(a) : square root

sin(a) : sine

cos(a) : cosine

tan(a) : tangent

asin(a) : arcsine

acos(a) : arccos

atan(a) : arctangent

atan2(a,b) : returns the principle value of the arctangent for the complex number `(b,a)`

sinh(a) : hyperbolic sine (not defined for complex argument)

cosh(a) : hyperbolic cosine (not defined for complex argument)

tanh(a) : hyperbolic tangent (not defined for complex argument)

cmplx(a) : converts a real to a complex number with zero imaginary part

cmplx(a1,a2): converts a real to a complex number `a1+ia2`

aimag(z) : imaginary part of complex number z

real(z) : real part of complex number z (casts complex to real)

conjg(z) : conjugate of complex number z

Character Intrinsics

len(c) : length

len_trim(c) : length of `c` if it were trimmed

lge(s1,s2) : returns `.true.` if s1 follows or is equal to s2 in lexical order, otherwise returns `.false.`

lgt(s1,s2) : returns `.true.` if s1 follows s2 in lexical order

lle(s1,s2) : returns `.true.` if s2 follows or is equal to s1 in lexical order

llt(s1,s2) : returns `.true.` if s2 follows s1 in lexical order

adjustl(s) : returns string with leading blanks removed and the same number of trailing blanks added

adjustr(s) : returns string with trailing blanks removed and the same number of leading blanks added

repeat(s,n) : concatenates string s to itself `n` times. Return variable should be declared large enough.

scan(s,c) : returns the integer starting position of string c within string s or zero if it is not found

trim(c) : trim trailing blanks from c (returns another character string)

DATA statements

A `DATA` statement may be used to initialize a variable or group of variables. It causes the compiler to load the initial values into the variables at compile time; it is thus a nonexecutable statement. It takes the general form

```
DATA varlist /vallist/ [, varlist /vallist/]
```

Examples

```
DATA a,b,c / 1.,2.,3./

DATA a,b,c / 1.,2.,3. /, X,Y,Z / 4.,5.,6. /

DATA ra / 100*1.0 /
```

The last form illustrated above initializes a 100-element array to one at compile time. Individual elements or sections may also be initialized in a similar manner. In most cases, the `data` statement can be eliminated by initializing variables in their declarations.

```
REAL       ::  a=1.,b=2.,c=3.

REAL, DIMENSION(100)  ::  ra=(/100*1.0)/
```

Exercises

1. Write a program that adds two integers together. Be sure to use correct style (program name, no implicit typing, correct declarations).

2. Write a program that adds a real number to an integer. Follow correct style and be sure to cast one of the numbers appropriately.

3. Write a program that declares three character variables. Initialize two of them to any desired constant strings. Concatenate those two and store the result into the third.

4. Write a program that declares some real and complex variables. Initialize the complex variables. Store the real part of one complex variable into a real variable and its imaginary part into another real variable. Add two complex variables and store the result into another. Multiply one complex variable by the conjugate of another.

3 Arrays

Fortran is primarily a language for numerical computing. In most mathematical formulations, arrays are essential. An **array** is a variable that consists of more than one **element,** each of which is of the same type. (A single-valued variable is called a **scalar**.) An array has at least one **dimension**. The *extent* of the dimension is the number of elements of the specified type in that dimension; the *size* of the array is the total number of elements. Thus the size of an `NxM` array is `N` times `M`. The **shape** of the array is an ordered list of its extents; i.e. the shape of an `NxM` array is `(N,M)`. The number of dimensions of an array is called its *rank*. Thus a two-dimensional array has rank 2, a four-dimensional array has rank 4, and so forth. The Fortran 2003 standard permits arrays of rank up to 7; some compilers permit higher ranks as an extension and the 2008 standard permits up to rank 15, with some 2008 compilers permitting up to 31 dimensions. A rank-1 array is usually called a *vector* and a rank-2 array a *matrix*, in analogy to the similar mathematical entities.

Static arrays have a fixed extent for each dimension that is declared along with the name of the array. Other types of arrays are possible in Fortran 2003; these will be described in the section on memory management.

3.1 Array Declarations

Arrays may be of any type supported by the compiler. They are declared with the `DIMENSION` keyword. This keyword is followed by the integer extent in each dimension, enclosed within parentheses and separated by commas.

Examples

```
INTEGER, DIMENSION(10)     ::  iarr

REAL (rk), DIMENSION(200,200) ::  rarr, darr

CHARACTER(LEN=12), DIMENSION(4) ::  carr

COMPLEX, DIMENSION(10,5,3)  ::  zarr

LOGICAL, DIMENSION(200)    ::  larr
```

The total memory space required by the array will be the number of elements multiplied by the number of bytes occupied by each element. For example, the total memory occupied by a 100x100 double-precision array, assuming an 8-byte double as in IEEE 754, is 100x100x8 bytes. Defining a kilobyte as 1024 bytes, the memory size of this array is about 78 kilobytes.

Fortran permits any dimension of an array, or all dimensions, to be of zero extent. This is mainly to simplify coding, by eliminating special cases. An array with zero extent is not conformable to any other array and cannot be an argument to array operations.

By default, the lower bound of an array dimension in Fortran is 1 (not 0). If no lower bound is indicated, the default will always be used. However, the programmer may specify a lower bound by including it in the declaration:

```
real (rk), dimension(0:3)     :: E
integer  , dimension(-1:12, -3:15) :: Nv
```

In the above examples, the extent of the vector `E` is 4; the shape of the array `Nv` is `14x19`. The bounds must be integers and the lower bound must be less than the upper bound.

Elements of an array are denoted by *indices*. For a "matrix" the first index indicates the row and the second the column. For higher-dimensional arrays, subsequent indices generalize row and column in the obvious way. Thus a single element of a rank-3 array `A` can be denoted `A(i,j,k)`.

3.2 Storage Order

In the computer's memory, an array is actually a linear list of elements, regardless of the actual declaration in the program. Therefore, there must be a mapping between the indices of the multi-dimensional array in the program and the index used by the compiler to point to an element in memory. In order for this mapping to be established, a convention must exist on the ordering of multi-dimensional arrays. Fortran uses what is known as *column-major ordering*. In this convention, the *innermost* index varies fastest. For example, for a `2x3` array the layout in memory would be

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

That is, a two-dimensional array is stored by column. This is easy to visualize for matrices but the principle extends to arrays of higher rank.

Most (newer) computer languages use row-major ordering, which is more intuitive for matrices; however, Fortran's convention is more natural for functions, in which the dependent variable varies fastest and is represented first.

3.3 Array Constructors

A vector or array can be initialized or assigned to explicit values by enclosing a comma-separated list between forward slashes and parentheses. For example, assigning a vector `V` to the first nine integers would be accomplished with the following statement:

```
V = (/1., 2., 3., 4., 5., 6., 7., 8., 9./)
```

A form of implied `DO` may also be used:

```
IV = (/i, i=1,9/)
```

To construct a multidimensional array by this means, the `RESHAPE` intrinsic must be used. An example is shown in Section 3.5.1 when this intrinsic is described.

3.4 Subarrays

A subarray is a *section* of a larger array. It is defined by indicating the range of indices with the colon notation.

If we declare

```
real, dimension(10,10) :: A
```

then a subsection of `A` that selects row elements from 2 to 5 and column elements from 4 to 9 is denoted as `A(2:5,4:9)`.

Entire array sections may be referenced in Fortran by replacing the indices with a colon. For example, if `A` is a `4x3` array, the subarray `A(:,3)` indicates a vector of extent 4 that is formed by taking the elements of column 3. Similarly, if `A` is `4x5x6`, the section `A(:,2,:)` is a rank-2 array with extent `4x6`. Note that both the lower and upper bounds are included within the slice.

3.5 Array Operations

Most arithmetic and mathematical operations in Fortran can be applied to entire arrays simply by using the name of the variable. Such operations are performed elementwise on the array or arrays. Note that if more than one array participates in the operation, the shapes must *conform*, i.e. must obey the rules for the operation. For example, it is only possible to perform elementwise arithmetic operations on arrays with the same shape, whereas matrix multiplication requires matrices of dimension `NxM` and `MxK`.

Examples

```
real (rk), dimension(200,200)  ::  rarr, sarr
…
rarr = 3.0
sarr = rarr + 8.0
```

The first statement above causes the value 3.0 to be assigned to each element of array `rarr`. The second statement adds the value 8.0 to each element of `rarr` and assigns it to the corresponding element of `sarr`. In order for this to be a valid assignment, `rarr` and `sarr` must agree in shape and kind.

3.5.1 Array Intrinsics

Fortran 2003 offers a large number of intrinsic functions that operate on arrays. Most of the mathematical intrinsic functions can take array arguments.

Examples

```
B = cos(A)

X = aimag(Z)
```

As is true for arithmetical operators, the arrays must conform, i.e. must agree in shape and kind. It is possible to cast arrays element-by-element in order to obtain agreement in kind.

Intrinsic functions are available that compute standard mathematical matrix transformations.

`dot_product(A,B)` : Computes the dot product, defined as the sum of the elementwise multiplication of vectors (one-dimensional arrays) `A` and `B`. If `A` is complex it returns the sum of the multiplication of the conjugate of A with B.

`matmul(A,B)` : Returns the product of one- or two-dimensional arrays `A` and `B`, where `A` must be `MxN` and `B` must be declared `NxK`. The result is an array of shape `MxK`. `A` may be a vector of size `M`, in which case `B` must be `MxK` and the result is a vector of size `K`. Note that Fortran does not make a distinction between "column" and "row" matrices and it is rarely necessary to declare an array with an extent of 1.

`transpose(A)` : Returns the transpose of array `A`. The array `A` must be of rank two (i.e. a matrix).

Several array reduction intrinsics are provided. In the list below, the argument `Dim` in square brackets is optional, and indicates that the reduction is to be taken only along one dimension rather than over the entire array.

It is important to note how the dimension argument behaves in these intrinsics; it indicates the section to be taken. Mathematically, the vectors that span through the `Dim` dimension are used in the operation; the rank of the result is one lower than that of the input and the shape is determined by the dimensions *other than* `Dim`. For example, for the intrinsic `maxval`, which determines the maximum value for the array or a section thereof, the result is as follows:

```
integer, dimension(3,3) ::  B
```

Assume B is assigned

```
   3. 8. 1.
   7. 6. 5.
   4. 2. 9.
```

`maxval(B)` is 9.

`maxval(B,1)` is (7.,8.,9.)

`maxval(B,2)` is (8.,7.,9.)

Similarly, if `A` is `LxMxN`, `SUM(A,DIM=2)` is equivalent to `SUM(A(i,:,j))` and the result is of shape `LxN`.

Reduction Intrinsics

`All(Mask [,Dim])` : This function takes a logical array argument and returns a logical value. It returns `.true.` if all elements of `Mask` are true; otherwise it returns `.false.`

`Any(Mask [,Dim])` : This function takes a logical array argument and returns a logical value. It returns `.true.` if any element of `Mask` is true; otherwise it returns `.false.`

Count(Mask [,Dim]) : This function takes a logical array argument and returns an integer value. The return value is the number of elements of Mask that are .true.

Maxval(A [,Dim] [,Mask]) : This function takes an integer or real array argument and returns an integer or real value. The return value is the scalar maximum value in the array, optionally only among those elements for which Mask is .true., if Dim is absent. If Dim is present, the result is an array of rank one less than A that specifies the maximum value for each vector spanning the requested dimension.

Maxloc(A [,Dim] [,Mask]) : This function takes an integer or real array argument and returns a rank-one integer array containing the locations of the maximum values in the array. If Mask is present, only elements for which the mask is .true. are considered. The dimension argument behaves similarly to maxval.

Minval(A [,Dim]) : This function takes an integer or real array argument and returns an integer or real value. The return value is the minimum value in the array.

Minloc(A [,Dim] [,Mask]) : This function takes an integer or real array argument and returns a rank-one integer array containing the locations of the minimum values in the array. If Mask is present, only elements for which the mask is .true. are considered. The dimension argument behaves similarly to minval.

Product(A [,Dim]) : This function takes an integer, real, or complex array argument and returns an integer, real, or complex value. The return value is the total product of all the elements of the array.

Sum(A [,Dim]) : This function takes an integer, real, or complex array argument and returns an integer, real, or complex value. The return value is the total sum of all the elements of the array.

Inquiry functions return properties of an array.

Allocated(A) : For an allocatable array, returns .true. if it has been allocated, or .false. if it has not. Allocatable arrays will be discussed in the section on memory management.

Lbound(A [,Dim]) : Without Dim, returns a rank-one array containing the lower bounds. With Dim, returns a scalar integer that is the lower bound along dimension Dim.

Shape(A) : Returns a two-element array containing the dimensions of the array A.

Size(A, [Dim]) : Without Dim, returns an integer that is the total size of A; i.e. LxMx …N. With Dim returns the size along the specified dimension.

Ubound(A [,Dim]) : Without Dim, returns a rank-one array containing the upper bounds. With Dim, returns a scalar integer that is the upper bound along dimension Dim.

Array Manipulation

Merge(A, B, Mask) : Merges two arrays based on the logical array Mask. Returns an array in which a given element is from A if the corresponding element of Mask is .true. or from B if Mask is .false. All three arrays must have the same shape unless either A or B is a scalar; in that case it is broadcast to all the elements of the array according to Mask.

Pack(A, Mask [,Vector]) : Without Vector, returns a one-dimensional array containing the elements of A corresponding to .true. values of Mask, laid out in the standard column-major order. Mask may consist of

the scalar value `.true.`, in which case the entire array is packed. That is, `Pack` converts a multi-dimensional array into a linear one-dimensional array. If `Vector` is present, its elements are inserted into the result after all elements of `A` have been selected by `Mask`. In other words, if not all elements of `A` are packed, the rest of the one-dimensional array is filled with the values of `Vector`.

`Unpack(V, Mask [,Field])` : Unpacks the one-dimensional array `V` into an array of the size and shape of the logical array `Mask`. When `Field` is present, it must have the same shape and size as `Mask` or it must be a single scalar. The value of a particular element of the unpacked array is the corresponding element of `V` where `Mask` is `.true.`, and where `Mask` is `.false.` the corresponding value of `Field` (or, if scalar, the single value of `Field`) is inserted.

Array Reshaping and Shifting Intrinsics.

`Reshape(A, Shape [,Pad] [,Order])` : Returns an array with shape given by the two-element vector `Shape`. `Pad`, if present, is an array of the same type as `A`. `Order`, if present, specifies the ordering of the reshaped array.

Example:

```
B = reshape((/1, 2, 3, 4, 5, 6, 7, 8, 9/), (/3,3/))
```

Then B is

```
1 4 7
2 5 8
3 6 9
```

Note the column-major ordering. For row-major ordering, use

```
B = reshape((/1, 2, 3, 4, 5, 6, 7, 8, 9/), (/3,3/), Order=(/1,2/))
```

which results in

```
1 2 3
4 5 6
7 8 9
```

`Spread(A, Dim, Ncopies)`: Returns an array of the same type as `A` with its rank increased by one. `A` may be a scalar or an array. Along dimension `Dim`, which is a required parameter, the elements of the result are the same as the elements of the corresponding source `A`.

Example:

```
real, dimension(3) :: A = (/1.,2.,3./)
real, dimension(3,3) :: B

B = spread(A, 1, 3)
```

This gives

```
1. 2. 3.
```

```
1. 2. 3.
1. 2. 3.
```

For

```
B = spread(A, 2, 3)}
```

The result is

```
1. 1. 1.
2. 2. 2.
3. 3. 3.
```

`Spread` is mostly used to populate arrays with a vector.

`Cshift(A, Shift [,Dim])` : Circular shift. Shift the elements of `A` by `Shift` times circularly. `Shift` must be a scalar if `A` is a vector; otherwise it may be either scalar or array. The optional argument `Dim` specifies along which dimension to shift, if it is present.

`Eoshift(A, Shift [,Boundary] [,Dim])` : End-off shift. Shift the elements of `A` by `Shift` times, dropping any elements shifted off the end. `Shift` must be a scalar if `A` is a vector; otherwise it may be either scalar or array. The optional argument `Boundary` provides values to be inserted into the locations formerly occupied by the dropped values; if it is omitted, the value zero is inserted. The second optional argument `Dim` specifies along which dimension to shift, if it is present.

Exercises

1. Write a program that declares two real and one integer arrays, each of size (100,100).

   a. Initialize one real array to all zeroes and the other to all threes. Initialize the integer array to all six. Add the integer array to the array whose elements take the value of three and store into the third array. Remember to cast appropriately.
   b. Change your program to add an integer parameter initialized to 100. Change the declaration of the arrays so that their size is determined by this parameter.

2. Write a program that declares three arrays, multiplies two of them and stores the result into the third. The arrays may be of any appropriate size and initialized in any manner.
3. Write a program that finds the sum of a rank-3 array with the argument `dim` set to 2.

4 Input and Output

No program is useful unless varying data can be entered into it and the results communicated to the outside world. This is accomplished through *input/output* routines. Input and output are accomplished by operations on **files**. In order to be written to or read from, a file must be *open*. A *closed* file is protected. Files are often called **units** in Fortran parlance.

In some computing environments such as Unix, some files are automatically open for every process. In Unix these are standard input, standard output, and standard error. Normally these files are attached to the console, i.e. direct entry or output, but they may be redirected. Similar files exist under environments such as Windows, but they work with the console window and not with the more typical graphical user interfaces.

Files are identified by some form of *file handle*. In Fortran the file handles are extremely simple – a file is associated with an integer, called the **unit number**.

Under Unix, by default unit 5 is associated with standard input (input from the console) and unit 6 is assigned to standard output (output to the console). These unit numbers should also work for the Windows equivalents. Unix also has a standard error console file that is usually associated with Fortran unit 2.

4.1 File Operations

A file may be opened with the statement

```
OPEN([UNIT=]un, FILE=fname [,options])
```

When placed first in the list, `unit=un` may be replaced by the unit number *un* alone. The list of *options* includes several that can be used to specify the type of file, error handling conditions, and so forth. The `FILE` specification is optional only if the type of file is specified as `scratch`. The result for a scratch file is system-dependent and this type is not widely used; thus we will not discuss it further in this short outline.

Options for the `open` statement include:

`IOSTAT=ios` This option returns an integer `ios`; its value is zero if the statement executed without error, and nonzero if an error occurred.

`ERR=label` *Label* is the label of a statement in the same program unit. In the event of an error, execution is transferred to this labeled statement.

`STATUS=stat` This option indicates the type of file to be opened. Possible values are `OLD`, `NEW`, `REPLACE`, `SCRATCH`, or `UNKNOWN`. If the file is `OLD` the file must exist under the name given by the `FILE` parameter. If it is `NEW` it will be created under the `FILE` name. For status `REPLACE` the file will be created if it does not exist, but if it does it will be deleted and a new one created under the same name. For `UNKNOWN` the status of the file is dependent on the system. In most cases, an `UNKNOWN` file is created if it does not exist, and if it does exist it is opened without further processing. `UNKNOWN` is the default and is usually sufficient for most purposes.

`ACCESS=acc` This option describes the way in which the file is organized. Permitted values are `SEQUENTIAL`, `DIRECT`, and `STREAM`. Data on sequential files must be accessed in a linear manner.

Direct access means that the (binary) data is indexed and can be accessed without reading all the data before it. Stream access is a binary format that corresponds closely to the C standard. It does not have a header or footer and must be positioned explicitly if the programmer does not wish to start at the beginning of the file. The default is SEQUENTIAL.

FORM=format This option specifies whether the data written to or read from the file are `FORMATTED` (human-readable text) or `UNFORMATTED` (binary, platform-dependent). The default is `FORMATTED` if access is unspecified. If access is direct the default is `UNFORMATTED`. Stream data must be specified as `UNFORMATTED`.

The `OPEN` statement has many other options that are rarely used or are seldom needed by beginners. Please consult a reference for their descriptions.

Examples

```
open(12, file="data.in")
open(unit=33,file="text.txt",status="old")
open(19,file="unit.dat",iostat=ios)
open(28,file="input.dat",status="unknown",err=100)
```

The status of a file may be tested at any point in a program by means of the `INQUIRE` statement.

```
INQUIRE([UNIT=]un, options)
```

or

```
INQUIRE(FILE=fname, options)
```

At least one option must be specified. The options include:

IOSTAT=ios Like the same option for the `OPEN` statement.

EXIST=lex Returns whether the file exists in the logical variable `lex`.

OPENED=lop Returns whether the file is open in the logical variable `lop`.

NUMBER=num Returns the unit number associated with the file, or -1 if no number is assigned to it. Generally used with the `FILE` form of the `INQUIRE` statement.

NAMED=isnamed Returns whether the file has a name. Generally used with the `UNIT` form of the `INQUIRE` statement.

NAME=fname Returns the name of the file in the character variable `fname`. Used in conjunction with the `NAMED` option. If `NAMED` returns `.true.` the name of the file will be returned in `fname`. If `FILE` is used the returned name need not agree with that value, but it will always be the correct name that should be used in any subsequent `open` statement.

READ=rd Returns a string `YES`, `NO`, or `UNKNOWN` to the character variable `rd` depending upon whether the file is readable. If this status cannot be determined, it returns `UNKNOWN`.

WRITE=wrt Returns a string YES or NO to the character variable wrt depending upon whether the file is writeable. If this status cannot be determined, it returns UNKNOWN.

READWRITE=rdwrt Returns a string YES, NO, or UNKNOWN to the character variable rdwrt depending upon whether the file is both readable and writeable or whether this is unknown.

The INQUIRE statement has several other options; see a reference for details.

When a file is no longer needed, it may be closed with the CLOSE statement.

```
CLOSE([unit=]un [,IOSTAT=ios] [,ERR=lerr] [,STATUS=stat])
```

The IOSTAT and ERR options are like the corresponding options for OPEN. STATUS is different; however. It may take the character values KEEP or DELETE. The default is KEEP unless the file was opened with status SCRATCH, in which case the default (and only permissible value) is DELETE.

Once a file has been closed, it is no longer available for reading or writing. A unit number may be closed, then reassigned to a *different* file name by a new OPEN statement. A single file may be closed and later reopened under a different unit number provided that its status was not set to DELETE.

4.2 Reading and Writing Data

The unit number is used in input-output statements to reference the file.

The WRITE statement is used to write to a file. Viewed as a function, WRITE takes two arguments. The first argument is the unit number of the file to which the write is to occur. If the first argument is an asterisk, the file is standard output. Standard output cannot be opened explicitly, but all other files must be associated with a unit number via an OPEN statement.

The syntax of the WRITE statement is

```
WRITE(UNIT=iu, options) varlist
```

The *varlist* is the list of variables to be written. Any particular member of the list can be an expression, as long as the expression can be evaluated when the statement is executed.

The most common options for WRITE are:

FMT=ifmt A format statement label specifier.

IOSTAT=ios Returns an integer indicating success or failure; its value is zero if the statement executed without error, and nonzero if an error occurred.

ERR=label The label is a statement label to which the program should jump if an error occurs.

The READ statement is used to read from a file. Its syntax is very similar to that of WRITE.

```
READ(UNIT=iu, options) varlist
```

The first argument is the unit number of the file from which the read is to occur. If the first argument is an asterisk, the file is standard input. Like standard output, standard input cannot be opened explicitly. All other files must be associated with a unit number via an `OPEN` statement just as for writing.

Unlike the `WRITE` statement, expressions may not appear in the variable list of a `READ`.

The `READ` statement has a number of options:

UNIT=iun The unit number

FMT=ifmt A format specifier

IOSTAT=ios Returns an integer indicating success or failure; its value is zero if the statement executed without error, and nonzero if an error occurred.

ERR=label The label is a statement label to which the program should jump if an error occurs.

END=label The label is a statement label to which the program should jump if the end of file is detected.

4.2.1 List-Directed I/O

The simplest method of getting data into and out of a program is *list-directed I/O*. In this approach, data are read or written as a stream into or from specified variables. The variables may be read from or written to a file or a standard console unit.

Print to Standard Output

```
print *, varlist
```

This prints the specified variables to standard output. (In the early days of computing, the "standard output" unit was a line printer.) Varlist is a comma-separated list of variables, array names, constants, or expressions. Expressions are evaluated before printing. The variables are printed as a stream and the elements of arrays are printed as a linear list in column-major order. The variables and elements are separated by spaces (the choice of how many is up to the compiler) and as many lines as needed will be used.

The `print *` statement is extremely useful for debugging, especially in a Unix environment.

List-Directed Output to a File

In list-directed output the second argument to `WRITE` is an asterisk. The list of variables and/or expressions follows on the same line.

```
WRITE(un,*) varlist
```

Examples

This statement writes to unit 10:

```
WRITE(10,*) a, b, c, arr
```

This statement is equivalent to `print *`:

```
WRITE(*,*) a, b, c, arr
```

Another method for writing to standard output, provided its unit number has not been reassigned to a different file, is (for Unix):

```
WRITE(6,*) a, b, c, arr
```

Note that *unlike* many other programming languages, Fortran always writes an end-of-line marker at the end of the list of items for any `print` or `write` statement. Printing a long line with many variables may thus require continuations. This behavior can be suppressed, but formatted I/O is then required.

List-Directed Input from a File

This statement reads from unit 10:

```
READ(10,*) a, b, c, arr
```

This statement reads from standard input:

```
READ(*,*) a, b, c, arr
```

An alternative form for the above READ echoes the form for PRINT.

```
READ *, a, b, c, arr
```

Another method for reading from standard input, provided its unit number has not been reassigned to a different file, is (for Unix):

```
READ(5,*) a, b, c, arr
```

When reading from standard input, the program will wait for a response from the console. It will not print any prompts for the user unless explicitly told to do so. Therefore, a set of statements such as the following is needed:

```
PRINT *, `Please enter the value of the variable inp:'
READ(*,*) inp
```

4.2.2 Formatted Input/Output

In many cases it is desirable to have more control over the format of the data to be read or written. List-directed I/O makes many assumptions and does not always print the results in a particularly readable form.

Formatted input/output requires that the programmer control the layout of the data. Not only the type, but also the number of characters that each element may occupy must be specified.

4.2.2.1 Edit Descriptors

A formatted data description must adhere to the generic form

```
nCw.d
```

Where `n` is an integer constant that specifies the number of repetitions (assumed to be 1 if omitted), `C` is a letter indicating the type of the item(s) to be written, `w` is the *total* number of spaces allocated to this item, and the `d` following the decimal point is the number of spaces to be allocated to the fractional part of the item. The decimal point and `d` designator are not used for integers, characters, or logical data items. Collectively, these designators are called **edit descriptors**. The space occupied by an item of data is called the *field*.

Integer Formats

Integer formats take the form

```
nIw
```

Here `n` is the repeat count. The `I` edit descriptor specifies integer type. The `w` is the total number of spaces, including any leading or trailing blanks, that are allocated to the individual integer(s).

Examples

```
I10
```

```
3I12
```

Floating-Point Formats

There are a number of options for formatting floating-point variables. The simplest edit descriptor for these types is

```
nFw.d
```

As usual, `n` is the number of repeats. `F` indicates floating-point type, `w` is the *total* number of spaces to be used, and `d` is the number of decimal places allocated to the fractional part of the number. Exponential notation is *not* used.

It is important to understand that `w` must allow room for the entire number; it should allow space for a negative sign, the number of digits to be devoted to the integer part, the spaces for the decimal part, the decimal point itself if it is present, *and* any leading or trailing blanks.

Examples

```
F15.5
```

```
6F8.2
```

The interpretation of this edit descriptor is slightly different on input and on output. On input, the actual number may appear anywhere in the field specified by the field-width edit descriptor (`w` in the above notation). The decimal point may be omitted from the number in the file to be read. If it is absent, the program uses the decimal edit decriptor to determine its location. If present, it overrides any specification of the length of the decimal part.

In the days of punched cards, data often appeared in a form such as

```
178564867391637690037728
```

If this string were read with the edit descriptor

```
3F8.4
```

the processor would first separate the input into 3 strings:

```
1785.6486  7391.6377 9003.7728
```

Punched-card data often consisted of such densely-packed numbers in order to save paper and time in punching. In the world of electronic text files, data are almost never run together in this manner. However, it is important to understand this example since it illustrates how the edit descriptors work.

If the program attempts to output these numbers with the same edit descriptor, a surprise may ensue:

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

Unlike input, on output the edit descriptors must allow space for the decimal point, so the specified field is not large enough for the numbers to be printed. When this occurs the Fortran processor prints asterisks to fill the field that cannot be accommodated. Also, on output the default is that each item is right-justified in its field and the field is padded with blanks. The justification can be changed but we refer the reader to reference manuals for details.

If the output edit descriptor is changed to

```
3F9.4
```

the result is

```
1785.64867391.63779003.7725
```

It is now possible to print the numbers, but because no space for blanks was allocated in the edit descriptor, they are run together. To separate the numbers requires at least

```
3F10.4
```

This yields

```
1785.6486 7391.6377 9003.7725
```

Numeric Conversions

The output of the final example may be surprising. The final digits do not agree with our input strings. This is a consequence of the fact that numbers in base 10 must be converted into the binary representation used by the computer. Most decimal number s cannot be exactly represented in binary within the number of bits devoted to a floating-point number.

The Fortran processor parses the data input initially as characters. The characters must then be converted to a number, and in particular a binary (or hexadecimal) number. On output the reverse conversion is performed.

Exponential Edit Descriptors

The `F` descriptor is inadequate for many, perhaps most, numbers used in numerical computing. The `E` edit descriptor is available for numbers that are best represented by scientific notation. It is used for both single and double precision.

An `E` format must allow space for leading and trailing blanks, the numbers, the decimal point if present, a possible negative sign, *and* four spaces for the exponent (this allows a negative sign to be part of the exponent). One reasonable format for the `E` edit descriptor for single-precision floating-point numbers is thus

```
E15.7
```

This descriptor normalizes the integer part to an absolute value less than one.

For exponents containing more than three digits the format string

```
Ew.de
```

n

should be used. For example, if numbers like 2.3×10^{1021} must be expressed, the appropriate format string would be

```
E15.7e4
```

There are some variants of the `E` edit descriptor that alter the ranges of the reported mantissa and exponent. The most useful variant is the `ES` edit descriptor, which prints the result in standard scientific notation, with the integer part normalized to have an absolute value greater than or equal to 1 and less than 10.

```
ES15.7
```

For more detailed control of output, including zero padding, printing initial plus signs as well as minus signs, etc., the interested reader should consult a reference. The same differences in behavior between input and output apply to the `E` edit descriptor as to the `F` descriptor.

Complex Formats

For complex variables, use two appropriate floating-point edit descriptors.

Character Formats

The character format is indicated by the A edit descriptor. As for numbers, the repeat count precedes the descriptor, and the field-width indicator defines the number of characters to be printed. That is, the generic form is

```
nAw
```

Examples

```
A12
```

```
10A8
```

If the field-width descriptor w is omitted, the full character variable as declared is taken. If the width descriptor is larger than the character length, the character is right justified and the field is padded with blanks. If the width descriptor is smaller than the length, the character is truncated.

On input, if the width descriptor is less than the length of the character string that is to be read, the string will be truncated. If this string is then stored into a character variable whose length is greater than the width descriptor, the portion of the string that was read in will be left justified in the variable and the remaining spaces will be padded with blanks.

Logical Formats

Logical variables are formatted with the L edit descriptor.

```
nLw
```

Generic Format

The G edit descriptor can be used when the quantity to be printed is not well known in advance. Its format is

```
nGw.d
```

For floating point numbers with small magnitudes, it is identical to the F descriptor. For all other floating-point variables, it is equivalent to the E descriptor. For any non-numeric type the d is ignored and it follows the rules for Iw, Aw, or Lw as appropriate.

Control Descriptors

These descriptors alter the input or output by adding blanks, new lines, and so forth. The / causes an end-of-record character to be written. Under Unix this results in the format skipping to a new line; the behavior under other operating systems may be system-dependent but most likely will have the same result. The X descriptor writes a blank. Both these descriptors may be preceded by a repeat count. Therefore, the statement

```
PRINT '(20X,I5,20X,I5/21X,I4,21X,I4)', N, M, I, J
```

causes 20 spaces to be skipped, an integer written, another 20 spaces skipped and another integer written; then on the next line 21 spaces are skipped and an integer written, and finally 21 spaces and an integer are output.

There are other control descriptors, but these two are the most commonly used.

Carriage Control When Fortran was developed, the only output devices were line printers. These devices had physical carriages to advance to new lines, and so Fortran provided *carriage control* for output (`print` and `write`) statements. For such devices, the first character of each format was not printed, but was used to control the printer. An initial blank was interpreted as a command to print on the current line. Some programmers make a practice of starting all output formats with a blank (e.g. `X`) to avoid having the first character interpreted for carriage control; however, most modern systems ignore carriage control on standard output devices such as a file or terminal.

4.2.3 Non-Advancing I/O

The default behavior is that each I/O statement reads or writes one *record* and appends an EOL (end of line) character. It is possible to suppress the EOL so that multiple statements can read from or written to the same line. This is indicated with the `advance` specifier to the read or write statement. Non-advancing I/O *must* be formatted even if the format is just to "read all characters."

```
READ(*,'(a)',advance='no',err=100) 'Please enter the name of the file:'
READ(*,'(a)') fname
```

4.2.4 Format Statements

Edit descriptors must be used in conjunction with a `PRINT`, `WRITE`, or `READ` statement. In many cases, it suffices to enclose the format in *single* quotes and parentheses and provide it as an argument to the keyword (it takes the place of the asterisk used for list-directed output).

Examples

```
print '(I5,5F15.5)', i, a, b, c, d, e

write(10,'(8E20.7)') arr

read(in_unit,'(2a)') c1, c2
```

If the same format is to be used repeatedly or it is complicated, the `FORMAT` statement can be used. This takes the form

```
label FORMAT(formlist)
```

The `FORMAT` statement *must* be labeled.

Examples

```
   10 FORMAT(I5,5F15.5)

3100 FORMAT(8E20.7)
```

If a FORMAT statement is used, the label is used in the input/output statement to reference it. That is, we use

```
PRINT label, varlist
WRITE(un, label) varlist
READ(un, label) varlist
```

Examples

```
   PRINT 10, i, a, b, c, d, e
10 FORMAT(I5, 5F15.5)

  WRITE(35, 3100) arr
3100 FORMAT(8E20.7)

  READ(22, 100) c1, c2
50 FORMAT(2a)
```

Format statements are *non-executable* but one may appear anywhere within the unit in which it is referenced. Traditionally either the statement immediately follows the I/O statement that references it, or else all FORMATs are placed together before the END statement of the unit.

4.2.5 Binary I/O

Formatted and list-directed I/O write text (ASCII or related character) data. Fortran can also read and write *binary* data, that is, raw data that is not converted from its machine format to character format or vice versa upon input or output. This is called **unformatted** or **stream data** in Fortran. Because it is in machine format, unformatted data can be reliably read only on the same platform (hardware plus operating system), and sometimes only with the same compiler, that wrote it. The major advantages to binary data are that it is already in machine format and does not require conversion, and it generally occupies less space than formatted data. Non-advancing I/O is not permitted for this type of data.

4.2.5.1 Unformatted Access

Input and output of unformatted data is fairly simple, but the type must be declared in the OPEN statement.

```
OPEN(20, file='mydat.dat', form='unformatted')
READ(20) a, b, arr
```

Note that no format of any type is specified. Unformatted data contains extra metadata, usually as a header and footer of 4 or 8 bytes, that indicates the number of *records* in the data. Each write statement writes one record and each read statement reads one record. The iostat, err, and end specifiers can be used as for formatted files.

4.2.5.2 Stream Access

Stream access is a new feature in Fortran 2003. The unit of access is nearly always bytes. There is no header or footer. This type of file access is similar to C-style binary files and usually interoperates easily with C binary files on the same platform. However, it is not possible to skip records; the location of an item must be computed from the number of bytes past the beginning of the file, starting at 1; this can be indicated via the `pos` specifier.

```
OPEN(iunit,file='bin.dat', access='stream', form='unformatted')
READ(iunit) my_array
OPEN(ounit,file='out.dat', access='stream', form='unformatted')
WRITE(ounit,pos=next_val) x
```

It is also possible to overwrite individual items by means of the `pos` specifier, but this is beyond the scope of this introduction.

4.2.6 Reading from the Command Line

Prior to the 2003 standard, Fortran did not have a standard method of reading arguments from the command line used to run the executable, though many compilers provided functions as an extension. The standard now contains two methods for reading command-line arguments. In Unix the typical paradigm is to obtain the number of arguments and then to use this number to loop through a procedure that reads each argument, where an "argument" is assumed to be a string separated from other strings by whitespace. For this method we first invoke the function `command_argument_count()` and then `call get_command_argument(n_arg,value [,length] [,status]`). In particular, we can parse a command line with the following code (the syntax will be clearer once the reader has gone through the section on loops):

```
nargs=command_argument_count()
do n=1,nargs
   call get_command_argument(n,str)
   !Process string str to determine what this argument is requesting
enddo
```

The argument values are always read as strings and the programmer must subsequently convert the value to a number if necessary, using internal reads as described in 4.3 below.

The alternative method is to use

```
call get_command(cmd [,length] [,status])
```

In this case the entire command line is returned and it is the responsibility of the programmer to parse it. The character string `cmd` must be of sufficient length to accommodate the maximum length of the command line.

4.2.7 Namelist

Many, possibly most, scientific codes have a large number of input parameters to control the setup of the problem to be solved. Remembering which parameter is which can require frequent reference to documentation and tedious counting of fields. Fortran provides **namelist** input to simplify this situation. In a `namelist`, parameters are specified by name and value and can appear in any order.

The `NAMELIST` is declared as a non-executable statement in the main program and the variables that can be specified in it are listed. The input file must follow a particular format; it begins with an ampersand followed by the name of the namelist and ends with a slash (`/`). The variables are specified by their names with an equals sign (=) between the name and its value. Only static objects may be part of a namelist; i.e. dynamically allocated arrays, pointers, and the like are not permitted.

```
NAMELIST /name/ varlist
```

Namelists are read with a special form of the `READ` statement:

```
READ(un,[nml=]name)
```

For example, in the main program a namelist could be declared as follows:

```
NAMELIST /fluid/ rho, eps, x0, dx, time0, dt
```

The corresponding input file would take the form

```
&fluid
rho  = 1.3
eps  = 1.e-7
dt   = 0.01
time0  = 0.0
/
```

Note that the parameters may appear in any order and may be omitted if they are not needed for a particular run or are used only to override default values.

The entire namelist would be read from a single statement

```
READ(20,fluid, ERR=999, END=999)
```

or

```
READ(20, nml=fluid)
```

The values of a namelist may be written with a similar statement:

```
WRITE(21, nml=fluid)
```

Like everything in Fortran, namelist names and variable names are not case-sensitive. Blanks are not permitted to be part of the namelist designator. Arrays may be namelist variables, but all the values of the array must be listed after the equals sign following its name. If any variable name is repeated, the final value is taken.

4.3 Internal Read and Write

Internal reads and writes are something of an oddity in Fortran. They are included here with input/output statements because they involve the `READ` and `WRITE` statements, but they are not truly input/output operations; they are the way in which Fortran casts from numeric to character types and vice versa. The character variable functions as an *internal file* in this context. An **internal write** converts from numeric to character type, while an **internal read** converts from character to numeric. That is, to cast from a numeric quantity to a character, the program *writes* the number(s) into the character buffer, whereas to convert in the opposite direction the program *reads* from the character buffer into the numeric variable(s). An appropriate format is required to specify the desired result.

Examples

Convert an integer to a character

```
CHARACTER(len=10) :: num
INTEGER     :: inum
WRITE(NUM,'(A10)') INUM
```

Convert a character to an integer

```
CHARACTER(len=10) :: num = "435"
INTEGER     :: inum

READ(NUM,'(i4)') inum
```

Convert an integer, such as a timestamp, to a character and append it to a filename.

```
    write(cycle,900) ncycle
900 format(i4)
  write(fname,910) filen, cycle
910 format(a10,'.',a4)
```

The above example has a disadvantage; if ncycle is less than four digits, the character into which it is written will be padded with blanks. A *non-standard* method of dealing with this is to use an integer decimal edit descriptor:

```
    write(cycle,900) ncycle
900 format(i4.4)
  write(fname,910) filen, cycle
910 format(a10,'.',a4)
```

This edit descriptor left-fills the internal file with zeroes. Although not standard, it is widely supported. A standard-conforming solution requires loops, a topic that has not yet been covered.

Exercises

1. Take any program from a previous exercise and change it to print its results.
    a. Use list-directed output to print to a console.
    b. Use list-directed output to print to a file. Remember to open the file appropriately.
    c. Use formatted output to print to a file.
    d. Use formatted output, print to a file, and add code for handling potential error conditions in the output statement.
2. Take one of your earlier programs that performs some mathematical operation on at least two scalars. Change the program so that those numbers are input from a file, rather than being set in the program itself.
    a. Use list-directed input from a file.
    b. Use formatted input from a file. Include instructions to handle error conditions.
    c. Use a namelist to input the values.

Read several numbers (some real and some integer) into a program. Convert some of the numbers into characters. Print the results in some informative manner.

5 Loops and Conditionals

5.1 Conditionals

Programs that always follow the same path are seldom very useful. Like all programming languages, Fortran provides a variety of constructs to handle decisions.

5.1.1 IF

Conditionals are used to determine whether one path or another should be taken. The `IF` and `IF-THEN-ELSE` statements implement this logic.

```
IF ( condition ) statement

IF ( condition ) THEN
  statement1
  statement2
  etc.
ENDIF
```

The *condition* is any expression that evaluates to a logical result (i.e. is true or false). The actions following the `IF` statement are performed if the expression evalutes to `.true.` Nothing is done if the conditional evaluates to `.false.` and the program continues with the next statement after the `IF` or `ENDIF`.

Only a single statement may follow an `IF` without a `THEN`. Any number of statements (including none) may appear between a `THEN` and an `ENDIF`.

Conditionals frequently involve the *relational operators* discussed previously.

If other actions can be taken when the conditional evaluates to `.false.` then the form of the `IF` becomes

```
IF ( condition ) THEN
  actions if true
ELSE
  actions if false
ENDIF
```

Examples

```
IF ( a .lt. b ) THEN
 z = (a, b)
ELSE
 z = (b, a)
ENDIF

IF ( x == y ) THEN
 w = x**2
ELSE
```

```
 w = x*y
ENDIF
```

`ENDIF` may also be written `END IF`.

One important but easily overlooked pitfall is comparisons of floating-point numbers. Since the set of floats is not complete, operations that might mathematically equal a certain exact value, most often zero, may not yield the expected result when carried out under floating-point arithmetic. Conversely, numbers that should mathematically be greater than zero may be smaller than the smallest value representable in the "number model," e.g. real or double precision. This condition is called *underflow*. Similarly, numbers that are too large to be represented in the number model represent *overflow*. Fortran provides some intrinsics to use in conditionals to determine whether a floating-point number is effectively zero, is at the limit of precision of the number model, is the largest that can be represented, and so forth. These numeric inquiry intrinsics have uses other than conditionals, of course, but this is where they commonly occur so it seems appropriate to introduce them at this point. The most useful of these intrinsics for conditionals are listed below:

`EPSILON(X)` : returns a number that is effectively negligible for the number model represented by `X`. `X` may be a constant such as `1.0` or `1.d0`.

`HUGE(X)` : returns the largest number that can be represented in the number model corresponding to `X`.

`TINY(X)` : returns the smallest number that can be represented in the number model corresponding to `X`.

Examples

```
IF ( x <= TINY(1.d0) ) THEN
! Possible divide by zero
  y = 0.0
ELSE
  y = w/x
ENDIF

IF ( y <= EPSILON(1.0) ) then
! y is effectively zero
  y = 0.0
ENDIF
```

`IF` statements may be nested with the `ELSE IF` statement:

```
IF ( ij .le. 0 ) then
   x(index) = z(index)
ELSE IF ( ij .gt. 0 .and. ij .le. im )
   x(index) = w(index)
ELSE
   x(index) = r*s
ENDIF
```

Note the compound conditional in the `ELSE IF` statement; `and` and `or` occur frequently in tests. Keep in mind the operator precedence rules and use parentheses freely if there is a chance of ambiguity.

In compound conditionals, most Fortran compilers *short circuit*; i.e. if they can determine the truth or falsehood of the conditional from the value of the first clause, they do not evaluate the second clause at all. Therefore, the second clause of a compound conditional should never be used to set the value of any expression.

5.1.2 CASE

Multiply nested `IF` statements can become confusing. The `CASE` construct can be used as a substitute for `IF`, but it is mainly used in circumstances in which several alternatives, not just one or a few, are permitted. It has the general syntax

```
SELECT CASE ( expression )
  CASE ( value1 )
    code
  CASE ( value2 )
    code
  CASE ( value3 )
    code
  CASE DEFAULT
    code
END SELECT
```

The `CASE DEFAULT` statement is optional. It provides a means for code to be executed if none of the conditions specified are met.

Example

```
SELECT CASE ( ij )
 CASE ( 0 )
    x = 0.
 CASE ( 1 )
    x = 12.
 CASE ( 10 )
    x = 99.
 CASE DEFAULT
    x = 1000.
END SELECT CASE
```

5.1.3 WHERE

Standard `if` statements may be used with arrays, but Fortran provides a conditional specially tailored for a common test on arrays. A *mask* must be defined, either as a conformable logical array or as a conditional that can be evaluated for all elements of the array.

```
WHERE ( condition1 )
  code
ELSE WHERE ( condition2 )
```

```
  code
END WHERE
```

Examples

```
WHERE ( A < 0.0 ) A=0.0
```

```
WHERE ( I > 2 )
 A = 3.
ELSE WHERE ( I < 0 )
 A = -4.
END WHERE
```

In the second example, we assume an integer array I is defined and has the same shape as the array `A`. `ELSE WHERE` and `END WHERE` may also be written as `ELSEWHERE` and `ENDWHERE`.

5.2 Loops

Much of programming consists of performing a group of operations many times on different data, i.e. iterating over the data. The primary loop construct in Fortran is `DO`.

5.2.1 DO loop

```
DO lv = lexpr, uexpr, sexpr
 code
END DO
```

or

```
name: DO lv = lexpr, uexpr, sexpr
        code
  END DO name
```

The *loop variable* `lv` must be a scalar integer variable. The variable `lexpr` can be any valid expression that evaluates to a scalar integer. It represents the value at which `lv` should begin. The variable `uexpr` similarly must evaluate to a scalar integer, and it represents the stopping point. The *stride* `sexpr` gives the integer to be added to the value of `lv` at each iteration. Evaluation stops when `lv` exceeds `uexpr`. The stride is optional; if absent it is assumed to be `1`. It may be negative (i.e. a loop may be "walked" backwards) but it should not be zero. For a negative stride the `lexpr` should be larger than `uexpr`.

In modern Fortran, the test that compares the loop variable to the stated maximum occurs at the *top* of the loop. Therefore, it is possible that a loop may not be executed at all. The actual increment or decrement of the loop variable for the next iteration occurs at the bottom of the loop.

Important: Fortran uses column-major ordering. Efficiency in looping over multidimensional arrays requires that the loop order follow the storage order. That is, loops should be arranged so that the outermost loop iterator corresponds to the rightmost array index, with the order then moving right to left. For example,

```
do k = 1, kmax
 do j = 1, jmax
  do i = 1, imax
   a(i,j,k) = b(i,j,k)
  enddo
 enddo
enddo
```

Similarly to other constructs, `END DO` may also be written `ENDDO`.

5.2.2 Implied DO

The *implied do* is a construct that is used in input/output statements to give more control over the output of an array. It can also be used to initialize an array to constant values. It takes the general form

```
(var, iterator=lexpr, uexpr, sexpr)
```

The parentheses are part of the statement and are required. Implied `do` statements may be nested; the loop closest to the variable is the "inner" loop.

Examples

```
WRITE(15,*) (A(I), I=1,10)

READ(10,100) ((R(I,J), J=1,10), I=1,20)

PRINT *, ((S(I,J), I=1,20), J=1,10)

DATA (((A(I,J,K), K=1,10), J=1,10), I=1,10) / 50. /
```

The second example reads an array in row-major order. The third example prints an array in column-major order. The last statement initializes a three-dimensional array as `DATA`.

5.2.3 DO WHILE

The `DO WHILE` loop executes for as long as a condition is `.true.`

```
DO WHILE ( condition )
 code
END DO
```

It is critical that the code within the loop provide a stopping condition for the loop; otherwise an *infinite loop* will result. The stopping condition may occur because the condition changes to `.false.` or else because of an exit statement (Section 5.3).

5.2.4 FORALL

The `FORALL` statement is a variant of the `DO` statement that was originally intended for autoparallelizing compilers. A `DO` statement requires that steps in the loop be carried out strictly sequentially; this is not true for a `FORALL`. The operations can be carried out in any order or, in the case of shared-memory processors, simultaneously.

```
FORALL(I=L:U)
```

where `I` is an integer variable and `L` and `U` are integer expressions. Multiple loop variables may be specified as long as they are separated by commas.

```
FORALL(I=L1:U1, J=L2:U2)
 array statement
 array statement
 etc.
END FORALL
```

A mask may be included within the limits to restrict the operation to certain locations.

Examples

```
FORALL ( I = M:N )  A(I) = B(I)

FORALL ( I = M:N, A(I) /=0.0 )  C(I)=B(I)/A(I)

FORALL ( I = 1:N, J=2:N-1 )
  A(I,J) = B(I, J+1) + 0.5*B(I, J-1)
  B(I,J) = A(I,J)
END FORALL
```

5.3 Breaking from Loops

It is frequently desirable to be able to exit from a loop if a certain condition is encountered. In Fortran the `EXIT` statement is provided for this purpose.

```
DO I = 1, 20

 IF (x(i) < tiny(1.0) ) then
    exit
 ENDIF

ENDDO
```

It may also be desirable to skip over the remainder of the loop under certain circumstances. The `CYCLE` statement causes the program to drop to the `END DO` statement immediately.

```
DO I = 1, IMAX
! For whatever reason, we don't want z to be negative
 IF ( x(i) > y(i) ) CYCLE
 z(i) = y(i) – x(i)
END DO
```

Sometimes it is desirable to construct an infinite loop whose end is controlled by an `EXIT` statement. This may be accomplished in two ways.

```
DO
 if ( condition ) exit
ENDDO
```

The absence of the loop variables in the `DO` statement indicates an infinite loop. The other way uses `DO WHILE`.

```
DO WHILE (.true.)
  if ( condition ) exit
END DO
```

With conditionals and loops we may now begin to write programs that do interesting things.

Exercises

1. Write a program that declares a real array of some particular size, using a `parameter` statement. Initialize the array to zero. Read in an integer variable from a file. Test that the value is no greater than the upper bound of the array. If it is, exit with an error message. If it is not, loop from `1` to the input value, setting the I*th* array element to the cosine of the loop variable (remember to cast appropriately). Print out every 3rd element of the array so constructed.

2. Use the Leibnitz formula to compute π

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

   Compute until your result agrees to four decimal digits with the correct value. This series converges very slowly so this will still take a lot of iterations.

3. (Advanced.) Write a program that solves Laplace's Equation $\nabla^2 = 0$ using a relaxation method. (If you are not familiar with relaxation methods, consult any elementary text on numerical methods.)

   a. Declare a two-dimensional `NxN` array, where `N` is some reasonably large value (at least 100x100). Declare a real variable `dx=0.1`. Set column 1 to 100 and all other values to 0. Use array and subarray operations for this purpose. Add a `DO` loop over `I` and `J` such that `A(I,J) = 0.25*(A(I,J-1) + A(I,J+1) + A(I-1,J) + A(I+1,J))`
   b. Be sure to pay attention to the loop bounds (that is, do not try to access an element of the array that is outside of its declared size).
   c. Add an outer loop that performs the inner loop repeatedly. Test whether for all elements of A, the value at the end of the loop is the same as (to the limits of floating-point precision) the value at the beginning of the loop. If this is true, exit from the loop. (Hint: you can introduce another array.) Print the result to a file in row-major order.
   d. Change the `DO` loop to a `FORALL`. Check that your result is the same as before.

If you have access to a software package that can plot text files, such as Python with Matplotlib, plot your solution to the Laplace equation. Bear in mind that such packages generally use the end-of-line marker to determine the end of a row, so you will have to use formatted output.

6 Subroutines and Functions

In any nontrivial program, different groups of operations are clearly defined that obtain particular results. These may be cast into the form of **subroutines** and **functions**. Structuring of programs into well-thought-out subroutines and functions is an important part of good software development practices.

The major difference between functions and subroutines is that functions take zero or more *arguments* and return exactly one result, whereas subroutines take zero or more *input arguments* and can return more than one result in the form of *output arguments*. Functions and subroutines are often collectively called **procedures**.

Like the `PROGRAM` unit, functions and subroutines constitute units. This is important for purposes of variable definitions, as will be explained later. Variables are *passed* to procedures. The `IMPLICIT NONE` statement that is recommended must appear in every program unit, including all subroutines and functions.

6.1 Functions

A `FUNCTION` is declared

```
<type> FUNCTION name(arglist)
```

Just like variables, a function is *typed*. The type corresponds to the *return value*. Remember that a function may return only one variable. The return variable is the name of the function itself.

Examples

```
real function myfunc(x,y)

integer function index(a,n)

logical function is_imag(z,w)
```

The function type may also be declared inside the function. This is the required form if more than one attribute is needed, such as array size.

```
function xy(w)
real, dimension(100) :: xy
```

A function ends with an `END FUNCTION` statement. Before the `END` it must assign some value(s) to its name; this is the way in which the result is returned to the calling unit.

Example

```
real function myfunc(x)
real, intent(in)   :: x
```

```
 myfunc = x**2

end function myfunc
```

Functions are invoked as if they were variables; that is, they are part of an expression.

Examples

```
y = myfunc(x)

if ( is_imag(x,y) ) then

C = matmul(A, Bfunc(x,N) )
```

A function may return its value under a name other than its own with a `RESULT` clause.

```
   REAL FUNCTION myfunc(x) RESULT y
```

It is still the case that only one variable may be returned. The variable corresponding to the `result` must be declared and must be assigned within the function.

Like all program units, functions end with an `END` statement. As usual, this statement should take the form

```
END FUNCTION name
```

in order to make clear to which statement the `END` corresponds.

6.2 Subroutines

Subroutines are by far the most common means of packaging routines. Subroutines are declared as

```
SUBROUTINE name(varlist)
```

The list of variables is called the **parameter list**. For example,

```
SUBROUTINE bvals(a,b,n)
```

Subroutines may return multiple values and thus do not take a type. They are invoked with the `CALL` statement.

```
CALL name(varlist)
```

Example

```
CALL SUBROUTINE mysub(a, b, n)
```

The variables in the `CALL` parameter list are called the *actual arguments*. The variables in the parameter list of the subroutine definition are called the *dummy arguments*. The names of the actual and dummy arguments need not be the same, but they **must** match in number and type; that is, the number of arguments in both lists must be the same and each corresponding argument must be of the same type.

Subroutines must end with an `END` statement.

```
END SUBROUTINE name
```

If it is desired to exit from the subroutine before the `END` statement has been reached, the `RETURN` statement may be used. `RETURN` causes control to return to the calling unit immediately. The `RETURN` may be labeled and it is executable.

```
RETURN
```

A `STOP` statement may appear in a subroutine, but a `RETURN` may not appear in a main program.

6.2.1 Subroutine Variables

Just as in the `program` unit, the variables must be declared appropriately. In the case of subroutines, those variables may be for input, output, or both. This may be indicated by the `INTENT` attribute. The intent may be `IN`, `OUT`, or `INOUT`.

Example

```
SUBROUTINE mysub(a, b, n, m, c, d)
REAL, DIMENSION(N), INTENT (IN)  :: a, b
INTEGER,      INTENT (IN)  :: n, m
REAL, DIMENSION(M), INTENT (INOUT) :: c
REAL, DIMENSION(M,N), INTENT (OUT) :: d
```

Variables declared `INTENT(IN)` may not be changed within the subroutine.

The `INTENT` attribute is optional, but its use is recommended.

In Fortran, variables are *passed by reference*. This means that at the compiler level, a pointer to the memory location occupied by the variable is passed to the subroutine. Therefore, changes to the variable that occur within

the subroutine *are* passed back to the calling unit. If this is unwanted or accidental, it is said to be a *side effect*. One purpose of the `INTENT(IN)` clause is to prevent side effects–if any changes are made to a variable so declared, the compiler will flag it as an error at compile time.

The Fortran 2003 standard does not require that a subroutine or function retain the values of local variables from one invocation to another. However, it is frequently useful to keep these values. In order to ensure that this occurs, the `SAVE` attribute must be applied to the variables. It may be applied either as part of the declaration or in a separate statement:

```
REAL, SAVE :: x,y
```

is equivalent to

```
REAL :: x,y

SAVE x,y
```

The `SAVE` statement alone, with no variable names specified, saves *all* local variables declared in the procedure. However, this can cause problems, so unless all the values really are needed, blanket `SAVEs` should be avoided.

Automatic arrays (discussed in the next chapter) cannot be saved.

6.2.2 Optional and Keyword Arguments

In many cases, some of the arguments to a subroutine can be handled by default values most of the time, with only an occasional need to change their values. In Fortran 2003, these can be given default values within the subroutine and passed only if needed. To do this it is only necessary to add the `OPTIONAL` attribute in the dummy variable's declaration and initialize it.

Example

```
SUBROUTINE mysub(a, b, c, d)
REAL     :: a, b
REAL, OPTIONAL :: c, d
```

In the above example, `a` and `b` are non-optional and `c` and `d` are optional. If none of the optional arguments need to be given values, then this procedure may be called with actual arguments as indicated below:

```
CALL mysub(aa, bb)
```

If all the optional arguments are to be changed, then the subroutine may be called in the usual way.

```
CALL mysub(aa, bb, cc, dd)
```

If argument `c` is to be changed but `d` is not, it may once again be called normally but with `d` omitted.

```
CALL mysub(aa, bb, cc)
```

What happens if it is desired to change `d` but not `c`? In such a case it is necessary to use a **keyword argument**. The name of the *dummy* argument is given, followed by an equals sign, then the name of the actual argument.

A procedure with optional arguments must usually test whether the caller has reset any of them. The intrinsic `PRESENT` returns a logical value indicating whether the optional argument is present in the argument list.

```
CALL mysub(aa, bb, d=dd)
REAL     :: a, b
REAL, OPTIONAL :: c, d

  IF ( PRESENT(C) ) THEN
   code
  ELSE
   code
  ENDIF

  IF ( PRESENT(D) ) THEN
   code
  ENDIF
```

Keyword arguments are not confined to use with optional arguments but may be used with any argument. They find their greatest use with optionals, but are also extremely helpful when argument lists are long. The important thing to remember is that the name of the *dummy* comes *first*.

Keyword arguments may appear in any order in the argument list. As indicated above, they may also be mixed with ordinary positional arguments. Positional arguments must always match their corresponding dummies by position even when keyword arguments are present, so any positional arguments should form the first group in the list.

6.3 Internal and External Procedures, Interfaces

Internal procedures are those defined within the same file as the unit that calls them. Internal subroutines are optionally denoted by a `CONTAINS` statement.

Example

```
PROGRAM myprog

statements (all of main program)

CONTAINS

  SUBROUTINE mysub1(a,b,c)
   statements
  END SUBROUTINE mysub1

  SUBROUTINE mysub2(a,b,c)
  statements
```

```
  END SUBROUTINE mysub2

END PROGRAM myprog
```

Conversely, external procedures are those resident in a different file. They may have an **explicit interface** declared within the calling unit. The use of explicit interfaces is strongly recommended, because if present they will cause the compiler to check that the number and type of the actual arguments agrees with the dummy arguments. Without an interface this checking will not happen, which can lead to runtime errors that can be very difficult to find.

An interface consists of the keyword `INTERFACE` followed by the declarations of one or more functions and/or subroutines, ending with `END INTERFACE`.

The interface must include the subroutine declaration (name and parameter list), the `IMPLICIT NONE`, the type of each dummy argument, any `USE` statements for modules, and the `END` statement for each procedure.

Example

```
INTERFACE

 SUBROUTINE mysub1(a, b, n)
  REAL, DIMENSION(N), INTENT(INOUT)  :: a,b
  INTEGER,    INTENT(IN)   :: n
 END SUBROUTINE mysub1

 SUBROUTINE mysub2(x, y, z)
   USE parms
   REAL (rk), DIMENSION(:), INTENT(INOUT)  :: x, y, z
 END SUBROUTINE mysub2

END INTERFACE
```

Interfaces serve a purpose similar to the function prototypes of other languages. In some circumstances they are optional; there are a few instances, mostly having to do with pointers and modules, in which they are required. The interface block is non-executable. It must appear with other declarations, before any executable statements but after any `IMPLICIT` or `USE` statements.

6.4 Passing Procedures as Arguments

Procedure names may be passed as arguments to subroutines. This is necessary if the subroutine performs some operation on a generic user-supplied function. Examples might include an optimization routine, a subroutine performing numerical quadrature (integration), and so forth. The procedure to be passed must have an interface in the calling unit. (If it is not possible to provide an explicit interface, an `EXTERNAL` declaration can be used, but this construct is a leftover from Fortran 77 and we will not discuss it further, preferring to emphasize newer constructs.)

Example

An integrating routine takes the function to be integrated as a parameter.

```
PROGRAM myprog
IMPLICIT NONE

REAL  :: a, b

INTERFACE
 REAL FUNCTION func(x)
 IMPLICIT NONE
  REAL, INTENT(IN)  :: x
 END FUNCTION func
END INTERFACE

code

CALL integrate(func, a, b)

code

STOP
END PROGRAM myprog

SUBROUTINE integrate(func, a, b)
 IMPLICIT NONE
 REAL, INTENT(IN)  :: a, b
 INTERFACE
  IMPLICIT NONE
  REAL FUNCTION func(x)
   REAL, INTENT(IN)  :: x
  END FUNCTION func
 END INTERFACE

   code

END SUBROUTINE integrate

REAL FUNCTION func(x)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x

  func = x**2

END FUNCTION func
```

It would be desirable for the procedure that actually calls the dummy procedure to also contain an interface, as indicated in the example, although it is not required. Note that no `CONTAINS` statement is made in the above example; this is due to the *scoping rules* to be discussed in Section 6.6.

6.5 Procedure Overloading

Procedure overloading enables a single function or subroutine to be written for arguments of different types, but be called by a single generic name. With operator overloading this can be extended to the basic operators; the operators can be defined upon user-defined types.

Interfaces are used to accomplish the overloading.

Example

```
INTERFACE func
  FUNCTION ffunc(x)
  real :: x
  END FUNCTION ffunc

  FUNCTION dfunc(x)
  double precision :: x
  END FUNCTION dfunc
END INTERFACE
```

To overload operators, a similar set of statements is used:

```
INTERFACE OPERATOR(+)
  FUNCTION addtype(x,y)
  type(mytype), INTENT IN :: x,y
  END FUNCTION addtype
END INTERFACE
```

In the latter example, two variables of type `mytype` could then be added with the usual + operator. The number of input variables must agree with the number of operands required by the operator. The overloading procedure must be a function that returns the result of the operation.

It is also possible to overload the assignment operator, in a manner similar to an operator overload. The overloading procedure must be a subroutine for which the first argument must be `INTENT OUT` and the second `INTENT IN`.

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE equaltype(y,x)
  type(mytype), INTENT OUT :: y
  type(mytype), INTENT IN  :: x
  END SUBROUTINE equaltype
END INTERFACE
```

The overloaded procedures or operators are accessible only from the program unit in which they are defined.

6.6 Variable Scope

In general, variables and some other entities, such as labels, are only defined within the particular program units in which they occur. The unit or units in which a given variable has a well-defined value is said to be the **scope** of the variable. In such a unit, the variable is said to be *in scope*. Outside of its *scoping unit*, the entity may be undefined, or its name may be used to refer to a completely different entity. A scoping unit may be one of the following program units:

- The definition of a derived type.
- A subroutine or function interface body.
- A program unit or subprogram (subroutine or function) unit.

If one of the above units is contained within the other, for example if an interface body appears within a subroutine, the scope of the exterior unit does *not* apply within the inner unit.

Example

```
SUBROUTINE mysub(x, y)
IMPLICIT NONE
REAL   ::  x, y

INTERFACE
 FUNCTION myfunc(x)
 REAL   :: x
 END FUNCTION myfunc
END INTERFACE

code

CALL integrate(y,myfunc)

more code

END SUBROUTINE mysub
```

The variable `x` in `mysub` is *not* the same as the variable of the same name in the interface block.

A variable within its scoping unit is said to be **local** to that unit. Variables defined within all units are **global**. In Fortran 2003, global variables must be declared in a module that is then `used` in all units. Entities declared within the module cannot be local to the unit that uses the module. However, the entities within the module are accessible to the unit by *use association*. Similarly, if a unit includes a `CONTAINS` statement, variables in the outer scope (the "host" of the contained subprograms) are available to the inner scopes by *host association* provided that no local variable has the same name.

Example

```
SUBROUTINE outer_scope
REAL  ::  x, y

x = 10.

print *, x

CONTAINS

  SUBROUTINE inner_scope
   REAL  ::  x, z
   x = 20.
   z = x + y
   print *, z
  END SUBROUTINE inner_scope

END SUBROUTINE outer_scope
```

Because of the `CONTAINS`, the variable `y` from `outer_scope` is accessible to `inner_scope`; without the `CONTAINS` statement, `y` would be undefined within `inner_scope`. However, the two `x` variables are different.

6.7 Recursion

Procedures may be recursive; that is, they can call themselves. Recursion is a natural way to express certain algorithms. The most popular example is the Fibonacci sequence. We start the sequence with

$$F_0 = 0, F_1 = 1.$$

Thereafter,

$$F_{n+1} = F_{n-1} + F_n$$

We may express this algorithm in pseudocode as

```
function Fib(N)

     if N=1 then

          Fib=0

     else if N=2 then

          Fib=1

     else
```

```
          Fib=Fib(N-1)+Fib(N-2)
     endif
end function Fib
```

To write the actual code we must inform the compiler that the function will call itself and we cannot use the name of the function as the return value. The syntax is

RECURSIVE FUNCTION X(VARS) RESULT(Y)

Thus our Fibonacci function would be

```
recursive function fib(n)result(f)
integer     :: f
integer, intent(in) :: n
     if (n==1) then
          f=0
     else if (n==2) then
          f=1
     else
          f=fib(n-1)+fib(n-2)
     endif
end function fib
```

It is very important that any recursive function have a stopping condition. Also, recursive functions should be used judiciously since they can result in a "stack explosion," since each call to the function results in a copy being made of its code.

Subroutines may also be recursive. In that case no `result` attribute is used.

6.8 Pure and Elemental Procedures

6.8.1 Pure Procedures

Because Fortran passes variables by reference (pointer to memory locations) it is possible for functions to have side effects. This is nearly always undesirable. Fortran 2003 allows the programmer to declare that a procedure has no possibility of side effects. This is accomplished with the pure attribute

```
PURE FUNCTION MYFUNC(X,Y)
```

A pure function does not change its input parameters. Both pure functions and pure subroutines must not alter any variables that are global to them from outside units, nor can any local variables be `saved`. Moreover, pure procedures cannot perform any input/output operations and cannot contain a `stop` statement.

In order to enforce these rules the compiler will require that a procedure declared `pure` must declare all dummy arguments with the intent attribute (unless the argument is a procedure name or a pointer). For a function, all dummy arguments must be declared `intent(in)`. There are also rules severely limiting targets and pointers that are beyond our scope here.

Any routine called by or contained in a pure procedure must also be pure. All intrinsics are pure so they may be called within programmers' pure procedures.

6.8.2 Elemental Procedures

We have seen that many intrinsic functions can operate element by element on arrays. Programmers can define their own elemental procedures. These procedures are called with one or more array arguments but the dimension of the arrays are *not* specified in the variable declarations within the procedure. If there is more than one array argument they must have the same shape. A type must be declared, however, so if the elemental procedure is to work for multiple types it must be overloaded. Character variables must have a fixed length.

An elemental function returns another array of the same shape as the argument(s). Arrays of types (Chapter 8) are permitted.

The function must be declared with the `elemental` keyword:

```
ELEMENTAL FUNCTION MYFUNC(A)
```

Example

```
elemental function rescale(A,B)
real      :: rescale
real, intent(in)  :: A, B
   if (max(A,B) /= 0) then
    rescale=(A-B)/max(A,B)
 else
    rescale=0.0
 endif
 end function rescale
```

The arguments of elemental subroutines should take at least one array and return at least one array, which will automatically be of the same shape. In general, if any argument is an array then all `intent(inout)` or `intent(out)` arrays must be arrays.

Elemental procedures are automatically `pure` and must conform to the requirements for pure procedures. They require an explicit interface and the `elemental` keyword must be supplied.

Exercises

1. Write a function that evaluates the hyperbolic cosine of a complex variable `z`. Use the formula

$$\cosh(z) = \frac{1}{2}(e^z + e^{-z})$$

Do not worry about numerical accuracy. Do not use the name `cosh` for your function, since that is the name of an intrinsic function.

2. Write a calling program for your hyperbolic cosine function. Test for both real and complex parameters. For real parameters, you can use the `cosh` intrinsic to check that your function is correct.

    a. Write a function that takes two scalar variables as input and returns their sum.
    b. Change your sum function into a subroutine.
    c. Write a subroutine that takes two scalar variables as input and returns a vector with its first element the sum of the input, its second their difference, its third their product, and its fourth their quotient. Be sure to check for error conditions and return with a message if an illegal operation would be attempted, specifically a division by a number that is effectively zero.

3. Change your program that solves Laplace's equation into a program with at least one subroutine. Pass the initial conditions to the subroutine.

4. Change this program to allow it to solve Poisson's equation

$$\nabla^2 A = -4\pi\rho$$

Note that if $\rho = 0$, Poisson's equation reduces to the Laplace equation. Pass both the initial conditions and the source function to the solver subroutine. Be sure to use an interface for the source function.

5. Write a subroutine with two mandatory and two optional arguments. Assign a default value to one of the optional arguments. Test whether the optional arguments are passed and perform some operations if they are. Write a calling program that exercises all possibilities for calling this subroutine.

6. Write a calling program for the `outer_scope` and `inner_scope` block in the text above. Add a variable `x` to the calling unit and set its value to `30`. Print all `x` variables. What does this illustrate about variable scope? Print `z` in `inner_scope`.

7 Memory Management

7.1 Allocatable Arrays

So far we have discussed only statically-allocated arrays. The size of these arrays is determined at compile time and is fixed throughout the run of the program. Fortran 2003 also permits *dynamically allocatable* arrays. The sizes of these arrays are determined at runtime, although their ranks must be declared to the compiler. These arrays are denoted by the keyword `ALLOCATABLE` and their declaration follows the form

```
type, DIMENSION(: [,:] [,:] [,:] [,:] [,:]), ALLOCATABLE :: A
```

Each colon corresponds to a dimension. For example, to declare a real rank-2 array `A`, we would write

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A
```

The declaration of an allocatable array merely informs the compiler that such a variable exists. To be brought into being it must be *allocated*:

```
ALLOCATE(A(sizelist))
```

For example, to allocate a two-dimensional array, the statement would be

```
ALLOCATE(A(M,N))
```

where `M` and `N` are determined dynamically by some means, such as by being read in to the program. Their values must be known when the `allocate` statement is issued.

Two important uses for allocatable arrays are arrays whose size may change from one run to another, and arrays that are needed only temporarily. The former case has many applications. For example, a code may be written to solve a system of linear equations. The methods used do not depend upon the size of the system, so "hard-coding" sizes makes no sense. The relevant arrays can be made allocatable and allocated to the desired size for a given problem at runtime.

The second case can be a means of conserving memory. Arrays such as temporary work variables can be allocated, then deallocated when not needed. If their creation is appropriately staggered and they do not persist when not needed, this can dramatically reduce the memory "footprint" of the program while it is running. Deallocating an allocatable array is done simply with

```
DEALLOCATE(A)
```

Note that the dimensions of the array are not needed to deallocate it.

More than one array may be allocated or deallocated by a single call:

```
ALLOCATE ( A(M,N), B(N), C(N,M) )
DEALLOCATE (A, B, C)
```

An attempt to allocate an array that is already allocated is an error. In order to avoid this, the intrinsic function `ALLOCATED` is available to test the status of the array. It takes an array argument and returns `.true.` if it is allocated, `.false.` if it is not.

Example

```
IF ( .NOT. ALLOCATED(A)) THEN
 ALLOCATE(A(M,N))
ENDIF
```

This is particularly useful for work arrays that are regularly allocated and deallocated.

In Fortran 95 and later, arrays allocated within a program unit are automatically deallocated upon exit from the unit; it is not necessary to deallocate them explicitly. This feature was introduced to minimize the possibility of *memory leaks*, which occur when arrays are allocated without the previous memory ever being freed.

7.2 Assumed-Shape Arrays

The dummy argument defined within a subprogram may be assumed to be the same shape as the corresponding actual argument. Only the number of dimensions must be specified in the subprogram. For example, if in the calling program `A` is declared

```
REAL, DIMENSION(10,10)  :: A
```

then the dummy array, `AA` for this example, could be declared in the subprogram as

```
REAL, DIMENSION(:,:)  :: AA
```

Note that if the lower bound of the actual argument is *not* the default of 1, the same lower bound must be declared for the dummy in order for the elements to align in the same manner. So if in our calling program we declare

```
REAL, DIMENSION(-1:10,2:10)  :: A
```

the corresponding dummy array would be

```
REAL, DIMENSION(-1:,2:)  :: AA
```

7.3 Automatic Arrays

Arrays local to subroutines often need to be of variable size, depending upon parameters set within the calling unit. An alternative to an allocatable array is the *automatic array*. This is a local array whose size is passed via the parameter list.

Example

```
SUBROUTINE mysub(a,b,n)
  REAL, DIMENSION(n,n), INTENT(INOUT)  :: a, b
  INTEGER,      INTENT(IN)   :: n

  REAL, DIMENSION(n,n) :: temp
```

It is possible to combine automatic and assumed-shape arrays in the same subprogram with the help of the `SIZE` intrinsic.

Example

```
SUBROUTINE mysub(a,b)
  REAL, DIMENSION(:,:), INTENT(INOUT)  :: a, b

  REAL, DIMENSION(SIZE(a,1),SIZE(a,2)) :: temp
```

The standard sets no requirements on the implementation, but most compilers automatically create automatic arrays upon entry into the subprogram and destroy them upon exit.

7.4 Pointers

In general, a **pointer** is a variable that exists for the purpose of referring to some other variable. Pointers in Fortran have some similarities to pointers in other languages such as C, but they are not the same and their unique properties must be kept in mind. In Fortran, the object to which a pointer points is called a **target**.

Pointers are declared with the `POINTER` attribute. If a pointer is to point to an array, the rank must be specified but the bounds will be taken to be the same as those of the target.

Example

```
REAL, POINTER       ::  P
REAL, POINTER, DIMENSION(:,:) ::  PA
```

In order to be useful, a pointer must be *associated* with a target. The target must be declared with the `TARGET` attribute.

```
REAL, TARGET      :: X
REAL, TARGET, DIMENSION(:,:)  :: A
```

A pointer can be associated with a target by a special assignment operator =>:

```
P  => X
PA => A
```

If pointers are *equated* then something quite different happens; the target to which the pointer on the right-hand side is pointing is *copied*. For example, if

```
PA => TA
PB => TB
```

then

```
PB = PA
```

is equivalent to

```
TB = TA
```

that is, `TB` is overwritten.

Pointers may be initialized to a defined but unassociated state by means of the intrinsic function `NULL()`. This function takes no arguments and returns a disassociated pointer. It is a good idea to initialize all pointers with this intrinsic, since then none will ever be undefined.

```
REAL, POINTER        ::  P  => NULL()
REAL, POINTER, DIMENSION(:,:) ::  PA => NULL()
```

A pointer may be given its own storage by an `ALLOCATE` statement, much like an allocatable array.

```
ALLOCATE(P, PA(N,N))
```

Just as an allocatable array may be tested to determine whether it has been allocated, a pointer may be tested for association. The intrinsic function for testing association is `ASSOCIATED`; it takes a pointer and returns a logical.

```
IF ( ASSOCIATED(P) ) THEN
  code
ELSE
  P => X
ENDIF
```

The ASSOCIATED intrinsic can also check whether a pointer is associated with some *particular* target, if the target is included as a second optional argument.

```
IF ( ASSOCIATED(P, X) ) THEN
  code
ELSE
  P => X
ENDIF
```

When pointers are no longer needed, they may be released. If the pointer was allocated, it may be deallocated like an allocatable array.

```
DEALLOCATE(PA(N,N))
```

The programmer should be careful not to deallocate a pointer that is still associated with some target. This leads to a *dangling pointer*. Any attempt to use a dangling pointer is an error.

A pointer may be disassociated from its target with the NULLIFY intrinsic.

```
NULLIFY(P)
```

Nullification does *not* deallocate the pointer, it merely dissociates it from its target. If the pointer was allocated and is no longer needed, it must also be deallocated.

Pointers that refer to memory that is no longer accessible through any pointer (e.g. because the pointer was nullified) can lead to memory leaks. Programmers should take care that pointers are deallocated properly when they are no longer needed.

Most of the conditions under which pointers were required in Fortran 90/95 are now handled by allocatables. Allocatables are preferable to pointers under nearly all circumstances; however, pointers are still needed for certain data structures such as linked lists (Section 8.2.1) and for C interoperability, which is part of the 2003 standard but is beyond the scope of this tutorial.

Exercise

Rewrite your Poisson solver to use allocatable arrays whose size is determined by an input parameter at runtime.

8 Derived Types

Fortran provides a number of intrinsic types (real, integer, logical, complex, character)) that suffice for most numerical programming. However, even in scientific and engineering codes, not all considerations are numerical, and it is useful for programmers to be able to define their own types. Fortran 2003 provides derived types for this purpose. Derived types are like generalized arrays; in arrays, all elements must be of the same intrinsic type, whereas derived types permit different types to be grouped together and referenced by a single name. Alternatively, multiple subunits of the same intrinsic type may be collected into a single derived type.

Derived types are defined via the keyword `TYPE`. The layout must be specified with

```
TYPE mytype
  elements
END TYPE mytype
```

Specific variables corresponding to this type are declared with the same keyword, followed by the name of the type in parentheses:

```
TYPE(mytype) Atype
```

An example that might be relevant to numerical coding is a grid. Grids typically are laid out with three Cartesian coordinates which we shall denote as `x`, `y`, and `z`. The grid might also have associated with it the number of zones in each dimension. An example of how such a type might be constructed follows.

Example

```
TYPE grid
 INTEGER          :: nx, ny, nz
 REAL, ALLOCATABLE, DIMENSION(:) :: x, y, z
END TYPE grid
```

Please note that some older compilers will not permit allocatable arrays in derived types since it is a Fortran 2003 feature. For these older compilers the workaround is to use pointers instead.

```
TYPE grid
 INTEGER        :: nx, ny, nz
 REAL, DIMENSION(:), POINTER :: x, y, z
END TYPE grid
```

Functionally these pointers will behave exactly like allocatable arrays in that they can be allocated when a representative of the type is constructed.

It should also be noted that characters with undefined lengths (`char(len=*)`) are not permitted in derived types.

The variables that make up a derived type are called *elements* or *fields* or sometimes *members*, though the latter name is usually intended for elements of classes. The elements describe the *attributes* of the type. A variable of the type itself is often called an *instance* of the type.

In addition to containing arrays, derived types may contain other derived types. Arrays of derived types are also permitted.

To declare a variable of type `grid`, we would write

```
TYPE(grid) agrid
```

Components of the type may be addressed individually by means of the `%` operator. (Some compilers also allow a period as the field separator, as in C/C++.) Thus if we wish to manipulate the `x` component of our example grid type, we use the expression

```
agrid%x
```

or for an individual element of `x`

```
agrid%x(i)
```

For example, we can construct the three coordinate arrays for an object of type `grid` with an `ALLOCATE` statement in the usual way.

```
ALLOCATE(agrid%x(nxmax),agrid%y(nymax),agrid%z(nzmax))
```

Suppose we were to declare an array of grids for some reason:

```
TYPE(grid), DIMENSION(5) :: thegrids
```

Each "element" of `thegrids` is itself an object of type `agrid`. Thus we may refer to the *i*th element of the third grid with a statement such as

```
zb = thegrids(3)%z(i)
```

8.1 Type Extension

A derived type can be created by extending an existing type. The new type automatically contains all the elements of its parent. Any elements defined in the new type are in addition to the elements of the parent.

Example

```
type species
  character(len=20) :: genus_name
  character(len=20) :: species_name
  integer     :: max_clutch
```

```
   integer     :: incubation_time
   integer      :: time_to_fledging
end type species
type, extends(species) :: bird
  character(len=1) :: gender
  integer    :: age
end type bird
```

The type bird inherits all the characteristics of its species, plus it has the attributes that apply to an individual bird.

8.2 Example Applications

8.2.1 Linked Lists

Linked lists are a staple of non-numerical computer programming, but are often useful in scientific code as well. Construction of linked lists requires a pointer element that points to its own type.

Example

```
TYPE employee
 CHARACTER(LEN=80)   :: name
 INTEGER       :: ID
 TYPE(employee), POINTER :: next
END TYPE employee
```

We could work with a list of employees with the following snippet of code:

```
TYPE(employee) ::  first, current

ALLOCATE(first)

! read first employee entry
first%name = "My Name"
first%ID = 12345

! continue to process
IF ( no_more_data ) THEN
  NULLIFY(first%next)
!  handle end of data situation
ELSE
  ALLOCATE(first%next)
ENDIF

current => first
```

```
!process employees

DO WHILE ( not_done )
 current%name = "Name"
 current%ID = the_id
 IF ( more_data)
   ALLOCATE(current%next)
   current => current%next
 ENDIF
END DO
```

This is just a code stub, of course; a number of checks must be added and real code filled in for vague placeholders such as “no_more_data”, but the basic outline is similar for all linked lists; initialize the positioning pointer (`current`) to the first entry, process the entry, then set the “current” pointer to the “next” pointer.

8.2.2 Type with Array and Array of Type

Suppose we were developing an ecological model of bird nesting. We might want a type to represent a region containing different habitat types. Each cell of a grid could be assigned to a particular coded habitat type and a list of the resident birds could be constructed, based on migration into and out of the cell.

```
type plot

  integer  ::  habitat_code

  real   :: area

  type(bird), dimension(:),  allocatable :: bird_list

end type plot

…..

type(plot), dimension(:,:), allocatable  ::  field

…..

read(lun,fname) m,n, MAX_BIRDS

allocate(field(m,n))

do j=1,n

  do i=1,m

  read(fun,datafile) field(m,n)%habitat_code, field(m,n)%area

  enddo

enddo

! No birds yet, but allocate list to maximum in a cell
```

```
do j=1,n
  do i=1,m
  allocate(field(m,n)%bird_list(MAX_BIRDS)
  do k=1, MAX_BIRDS
    field(m,n)%bird_list(k)%genus_name=' '
    field(m,n)%bird_list(k)%species_name=' '
    field(m,n)%bird_list(k)%gender='F'
    field(m,n)%bird_list(k)%age=0
  enddo
 enddo
 enddo
```

Note that allocating an array of types *does not* generally create any instances of that type; it merely informs the compiler that there will be an array of a specified type. It is up to the programmer to fill each instance and, if appropriate, to allocate any dynamic arrays that are elements of the type. Typically we will write a procedure, often called a **constructor**, to perform this operation.

Exercises

1. Relativistic mechanics makes use of *four-vectors* that are generalizations of vectors to four-dimensional spacetime. They consist of a quantity that is a vector in three-dimensional space (e.g. momentum) and a fourth quantity that is a scalar (e.g. energy). One way of handling four-vectors in a program might be to make the lower bound zero; this would correspond to the ``scalar'' element. However, the zeroth element might have different properties from elements one through three; thus it might be desirable to create a derived type. Define a derived type for the *energy-momentum* four-vector, which contains the scalar three-energy E plus a three-dimensional vector $\vec{p}$, which is the ordinary momentum. Make the momentum element allocatable.

2. Define a derived type that would describe a species, with the aim of creating a simple database for the use of a code that models an ecosystem. Elements are up to the programmer's discretion but might include entries such as type of species (e.g. plant, animal, bacterium, etc.), genus name, species name, habitat type, observed population, and so forth. Make at least one member of the type be an integer vector to hold characteristics of the species such as habitat type, population, and other descriptors that might be useful to an ecology model. Include a pointer to the type so that a list can be developed.

3. Use the derived type in the previous Exercise to create a linked list of species for the ecological model. Add the checks that were omitted from the code stub in the text (check for unassigned pointers, nullify all pointers not in use, etc.).

9 Modules

Modules are one of the most important innovations of Fortran 90/95 and they have only increased in importance with later revisions of the standard. Modules permit encapsulation of related variables and procedures, provide a mechanism for controlling which procedures outside the module may access which data, and enable the creation of objects that can include function and operator overloading.

A module is a program unit and cannot stand alone; it must be included into other program units in order for anything in the module to be accessed. It also must be compiled before any program unit that uses it is compiled. Finally, the object file created when the module is compiled must be linked with the other program units to construct the executable. It is not necessary for a module to be in its own file but this is the normal practice unless the module is extremely short; regardless, the compiler must have seen the module before any other program unit can `use` it.

Modules begin with the keyword `MODULE`

```
MODULE modname
```

and end with an `END MODULE`

```
END MODULE modname
```

Within another program unit, a module is invoked via the `USE` statement.

```
USE modname
```

A module may use another module, but it may not use itself either directly or indirectly.

A module may contain both variables and functions/subroutines, only variables, or only functions and subroutines. All functions and subroutines must follow a `CONTAINS` statement. Each subroutine must end with

```
END SUBROUTINE sub
```

while the functions must end with

```
END FUNCTION func
```

Any variables declared prior to the `CONTAINS` statement are *global* in the module, and those variables that are generally accessible are global to all program units that use the module.

One very useful application for global variables is defining kinds. A module named `precs` could be defined in a similar manner to:

```
module precs

! Single precision
integer, parameter :: sp = selected_real_kind(6,37)

! Double precision
integer, parameter :: dp = selected_real_kind(15,307)

end module precs
```

If this module is used in every program unit, real variables could then be declared as

```
use precs

real(sp) :: x, y, z
real(dp) :: temp
```

Modules containing only global variables can also take the place of the deprecated `COMMON` statement of Fortran 77 and earlier; that is, variables common to more than one routine can be defined within the module and invoked by all those needing those variables. This application should be used sparingly, however.

One caveat must be noted in the discussion of variables in modules. As we have seen, the Fortran 95 standard does not permit allocatable arrays in derived types, even those that are defined within a module; a pointer must be used instead. This restriction has been removed in the Fortran 2003 standard. However, nearly all "Fortran 95" compilers now available support allocatables in derived types even if they do not support the entire Fortran 2003 standard.

Procedures defined in modules have implicit interfaces; that is, type, kind, and number of parameters are automatically checked by the compiler. Thus it is erroneous to define explicit interfaces for procedures in modules. Not all compilers will catch this error, however.

9.1 Data Accessibility

One of the purposes of modules is to create a program unit whose interface to external units can be controlled. This reduces the chances for *side effects*, unintended changes to variables. Data (variables and subroutines) within modules may be kept private to the module, which means they cannot be accessed or changed by external units, or they may be public, accessible to all units that use the module.

Data permissions are declared with the attributes `PUBLIC` and `PRIVATE`. Data declared public is accessible to any unit using the module; private data is accessible only to procedures defined within the module. The default is `PUBLIC`.

Permissions may be set either when the variable is declared or as a separate statement.

Example

```
REAL, PUBLIC  :: x, y, z
REAL, PRIVATE :: a, b, c
```

This is equivalent to

```
REAL :: x, y, z, a, b, c

PUBLIC  :: x, y, z
PRIVATE :: a, b, c
```

If the PUBLIC or PRIVATE attribute is specified without a list of data symbols, it sets the default for that module. For example, PRIVATE alone specifies that all symbols are private unless they are explicitly declared public; this is the opposite of the default but would often be useful, especially for large modules with many internal symbols. As a general rule, modules that define classes (Section 11.1) should declare a default of PRIVATE.

9.2 Procedure and Operator Overloading

Modules permit *overloading* of functions, subroutines, and operators. Procedure overloading means that versions of a single function or subroutine can be written that can accept different types in the parameter list, but the procedure can be called by a single generic name. Operator overloading means that the basic operators can be defined for new data types, typically a derived type, and the operator can be invoked to perform the indicated operation.

Overloading is accomplished by defining a MODULE PROCEDURE. An interface is defined for the generic procedure name, but since interfaces are automatic within a module, no actual interfaces are described; instead the specific names are given after the MODULE PROCEDURE keywords.

Example

The method is probably most easily explained by an example. Suppose it was desired to define a generic function diag that would take real, double precision, or integer parameters.

```
module isosceles
IMPLICIT NONE

public :: diagonal

private:: fdiagonal, ddiagonal, idiagonal

INTERFACE diagonal
 MODULE PROCEDURE fdiagonal, ddiagonal, idiagonal
END INTERFACE
```

```
CONTAINS

FUNCTION fdiagonal(x)
real :: fdiagonal
real :: x
fdiagonal = x*sqrt(2.0)
END FUNCTION fdiagonal

FUNCTION ddiagonal(x)
double precision :: ddiagonal
double precision :: x
ddiagonal = x*sqrt(2.d0)
END FUNCTION ddiagonal

FUNCTION idiagonal(i)
real :: idiagonal
integer :: i
idiagonal = real(i)*sqrt(2.0)
END FUNCTION idiagonal

END MODULE isosceles
```

Operator overloading can be used to extend operators or assignments to derived types. Within a module, this is accomplished in a similar way to overloading procedures.

```
INTERFACE OPERATOR(+)
  MODULE PROCEDURE addtype
END INTERFACE

INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE equaltype
END INTERFACE
```

The same rules described previously in Section 6.5 for overloading operators and assignment within explicit interfaces apply within modules: for operators the procedures must be functions with the correct number of arguments, while for assignment the procedure must be a subroutine with two arguments. For overloading arithmetic operators, the parameters are the values on which the operator works, they must be declared `INTENT(IN)`, and the function must return the result. In the case of assignment the first argument must be the value to be on the left side of the assignment operator and it must be `INTENT(OUT)` or `INTENT(INOUT)` while the second argument must pass the quantity to be assigned and it must be declared `INTENT(IN)`. The interfaces need not be declared within the module in which the procedures are defined (but this module must be used in the program unit that defines the interfaces); however, under nearly all circumstances it makes the most sense to place the overloading interfaces into the module that contains the procedures.

9.3 USE with options

The `USE` statement may be followed by options that specify which procedures are to be loaded from the module, under which names.

A module entity may be given a different name in the using program unit with the redirection operator =>

```
USE mymod, modprod=>prod
```

Within the unit that uses `mymod`, the routine called `prod` will be called `modprod`. Note that the name within the using unit "points" to the name within the module. Renaming can prevent *namespace collisions*, when more than one program unit contains an entity with the same name. For example, if the program unit uses more than one module but each has an entity named `prod`, one of these entities must be renamed if either is used. Only public entities may be renamed, since they are the only module entities that are accessible to the using program unit.

Sometimes not all public entities within a module are required by the unit that uses it. In this case, the `USE` statement may contain an `ONLY` attribute, followed by a list of the names to be made available to the using unit.

```
USE module statmod, ONLY : mean, sstdev=>stdev
```

In this example, `mean` is not renamed but `stdev` is.

There is no `USE EXCEPT` or similar statement. If all but a few entities are needed, and those unneeded names would collide with names within the using unit, the `USE` statement should rename those unwanted entities.

9.4 Summary

Modules are an innovation in Fortran 90/95 that fill many roles. They provide a mechanism for sharing variables among different program units. They can be used to overload procedures and operators. However, their most important function is to package related code into a self-contained unit with a well-defined interface. This enables them to function much like libraries; any new libraries written in Fortran should be contained within modules. Modules also implement the core ideas of object-oriented programming. Modules alone provide many of the benefits of OOP but greater capabilities are realized in Fortran 2003 and up with the introduction of **classes** (Chapter 11).

Exercises

1. Write a module that computes the mean and standard deviation of the values in an input vector. Only the return values should be public; all internal variables should be declared private. Use the module in a program.

2. Write another module that computes the standard deviation and the sample kurtosis of an input vector. Write a program that uses both modules, but loads the standard-deviation function from this module under a different name. Compute the mean, standard deviation, and sample kurtosis for a vector obtained using the random_number intrinsic.

3. Change one of the modules from the preceding exercise so that the functions will accept either real or double-precision input values.

4. Write a module in which a type is defined consisting of two reals and an array of undetermined size. Define the + operator for this type. Write a main program that uses the module, creates two of this type, and adds them.

10 Exception Handling

Many programming languages provide some means of testing for error conditions and handling them before they cause the program to terminate abnormally. We have seen in Chapter 4 how to deal with errors in input/output operations by use of the `iostat` and/or `err` flags to the `open`, `read`, and `write` operations. Another major category of exceptions occurs due to illegal mathematical operations or to operations that violate the ability of the floating-point model to represent the result; these are generally called *floating-point exceptions* (FPEs). Until the 2003 standard Fortran had no mechanism to handle these exceptions, though many compilers offered options to terminate the program upon encountering an FPE. Floating-point exceptions are not the only kind that can occur but for numerical code, which is the main area of application in which Fortran programmers have always worked, FPEs are by far the most important. Thus Fortran 2003 introduced a standard for a set of modules to handle these exceptions.

The IEEE standard specifies five flags for floating-point exceptions: `overflow`, `divide_by_zero`, `invalid`, `underflow`, and `inexact`. Overflow and underflow occur because the floating-point numbers, whether single or double precision, have a finite extent and can represent only a finite number of values. Overflow means that the number is too large to be represented by the type. Underflow means that it cannot be distinguished from zero to the specified precision; typically the hardware substitutes zero for the number. Divide by zero means an attempt to divide by zero (resulting in a mathematical infinity). Invalid represents a mathematically invalid operation, such as the square root of a real number. Inexact means that the number cannot be represented by the specified type without rounding. It is rarely necessary to test for underflow or inexact.

Fortran provides access to exception handling via several intrinsic modules. Only three are commonly required: `ieee_exceptions`, `ieee_arithmetic`, and `ieee_features`. The `ieee_arithmetic` module will also bring in the `ieee_exceptions` module.

10.1.1 IEEE Features

The `ieee_features` module provides only a set of named constants. These determine whether support is to be included for a particular IEEE feature. For example, if we do not wish the hardware to substitute zero for an underflow but to support an exception for at least one type of real, we would include

```
use, intrinsic :: ieee_features, only: ieee_underflow_flag
```

Note the `intrinsic` keyword; this declares that we wish to use the intrinsic module rather than one we might have written.

10.1.2 IEEE Arithmetic

The `ieee_arithmetic` module provides inquiry functions as well as some functions that return arithmetic values or set modes. Some of the more useful functions are

ieee_is_nan(x) : returns .true. if x is NaN, .false. otherwise.

ieee_is_finite(x) : returns .true. if x is finite, .false. if it is Inf or −Inf.

ieee_is_normal(x) : returns .true. if x is not denormalized (contains leading zeros in the mantissa).

call ieee_get_rounding_mode(value) : returns a type ieee_round_type that represents the rounding mode. If the hardware rounds with one of the IEEE modes the value will be ieee_nearest, ieee_to_zero, ieee_up, or ieee_down. If an IEEE rounding mode is not in use the value will be ieee_other.

call ieee_set_rounding_mode(value): specifies the rounding mode to be used henceforth.

10.1.3 IEEE Exceptions

The ieee_exceptions module is probably the most useful. It allows the programmer to control whether an exception will be handled by the code or by the default system exception handler. This module also contains several inquiry functions but also some routines to set handlers. Please note that many compilers still do not implement the IEEE exception functionality.

ieee_support_flag(flag,x) : returns .true. if the flag exception is supported for the type of x.

ieee_support_halting(flag) : returns .true. if the processor supports a user-initiated change in the halting mode.

call ieee_get_halting_mode(flag,halting) : flag is type ieee_flag_type and takes one of the values ieee_invalid, ieee_divide_by_zero, ieee_underflow, ieee_inexact and halting is .true. or .false.

call ieee_set_halting_mode(flag, halting) : flag is the type as above and halting is .true. or .false. depending on whether the programmer wishes to halt processing when the flag exception occurs.

Example

```
program testieee
use ieee_exceptions
!  ieee_overflow and ieee_divide_by_zero are in ieee_exceptions
implicit none

type(ieee_flag_type), parameter  :: overflow=ieee_overflow
real                             :: x,y
logical                          :: flag

  call ieee_set_halting_mode(overflow, .false.)

 !In a real code this would come from someplace other than explicit setting
```

```
  x=1.e12
  y=exp(x)

  call ieee_get_flag(overflow,flag)
  if (flag ) then
    print *, 'Overflow, setting to maximum'
    call ieee_set_flag(overflow, .false.)
    y=huge(x)
    print *, 'Largest value ',y
  else
    print *, 'Computed e^x ',y
  endif

end program testieee
```

Exercise

1. Write a function that computes y/x

   Your function should start with

   ```
   use ieee_arithmetic
   use ieee_features
   ```

   Invoke the `ieee_is_finite` function to check for division by zero and if it occurs, return `y=-999`, otherwise return the quotient. (This is obviously a contrived example.) Test your function with `x` equal and not equal to zero. Remember that the name of the function can be treated as any other floating-point value and can be an argument to the `ieee_is_finite` procedure.

11 Object-Oriented Programming in Fortran

Object-oriented programming is a modern paradigm for programming. At its most basic, it entails *data hiding* or protection of data, and *organization* of code into *units with discrete functionality*.

Data hiding means that the external units cannot access all the entities within the object; only those that are directly needed are "exposed" to the external units via an interface controlled by the programmer. This reduces the problem of side effects and it also simplifies maintenance, since calling statements do not need to be rewritten if the internal organization of the object is changed. Thus objects make heavy use of `private` variables.

Objects organize code into related units. An object consists of *data* and *behaviors* associated with that data. To use an example from realms of programming outside of the usual domain of Fortran, in a graphical windowing system the window itself might be an object. The window has behaviors; it opens, closes, resizes, etc., and it has associated data, such as its location on the screen and the number of pixels it occupies.

In scientific computing, not all entities map quite so naturally to objects, but code can still be grouped into functional units. For instance, in numerical computing a grid might be defined as a derived type. Its attributes might include the x, y, and z values of a particular point. If the grid has behaviors, such as would be the case for adaptive-mesh methods, the then grid should be treated fully as an object.

11.1 Classes

In the simplest terms, a class is a defined type which includes as elements not only variables but also procedures. In general OOP parlance the elements of a class are generally called **members** and member procedures are called **methods**. A variable declared to be of the type is an **instance** of the type; creating such a variable and loading any initial values is called *instantiation* of the type. The methods are accessible only through an instance of the class. Invoking a method is often referred to as *sending a message* to the instance.

Fortran 2003 specifically calls class methods *type-bound procedures* but we will use the terms interchangeably.

An example is probably the best illustration. First let us create a module that defines a class `obj`.

```
module obj_class
  implicit none

   private

   public           :: obj

   integer          :: i, j, l=5
   real            :: s = 25.

 type obj
  integer ::  m
  real, dimension(:,:), allocatable :: a
  contains
   procedure :: change_obj
   procedure :: del_obj
```

```
  end type obj

  interface obj
    module procedure init_obj
  end interface

  contains

   type(obj) function init_obj(n)
   integer, intent(in)  :: n
   allocate (init_obj%a(n,n))
   init_obj%a = s
   init_obj%m = l
   end function init_obj

   subroutine change_obj(self, n, r)
   class(obj), intent(inout) :: self
   integer, intent(in)   :: n
   real,  intent(in)   :: r
    self%a = r
    self%m = n
   end subroutine change_obj

   subroutine del_obj(self)
   class(obj)      :: self
    deallocate(self%a)
   end subroutine del_obj

end module obj_class
```

Several points should be noted about this class definition. First, we use an explicit interface and a module procedure to give the *constructor* the same name as the class. This is to adhere to the convention used in most object-oriented languages; it is not mandatory in Fortran. We could use the same interface to overload the constructor name for different parameter lists if it were necessary or desirable. Also note that the constructor is not a type-bound procedure; this is so that it can be called directly. Secondly, type-bound procedures *must* take as their first parameter a variable of the type of the class. In some object-oriented languages such as C++ the instance parameter is implicit and a keyword `this` exists to reference it. In other OOP languages such as Python the instance must be passed explicitly, as it is in Fortran. We have here followed the Python convention of referring to the instance variable as `self` but Fortran has no keyword for this purpose and the name of the variable is the programmer's choice; some programmers prefer the C++/Java convention and use `this` while others may choose to use an entirely different name. The instance variable must be declared with the `class` keyword and not the `type` keyword.

Also notice that only the constructor is public. This means that all other methods cannot be called directly but must be accessed through a variable (class instance). This is what we mean by data hiding. Outside program units cannot do anything directly with the methods, nor can they access any of the instance's members without "sending a message" to it via a method. This helps to prevent unintended changes to the class members.

The del_obj function is often called a *destructor*. It is not required. Destructors are especially desirable when the class contains an allocatable array or some other item that should be cleaned up when the instance is no longer required, such as a file that should be closed. Fortran 2003 also provides a `final` keyword to indicate a destructor, but at the time of this writing compiler support for this feature is unreliable so our example does not implement it.

We can make use of this object in the following main program:

```
program my_prog
use obj_class
implicit none

integer  :: mval
real   :: rval

integer, parameter  :: ns = 10

type(obj)     :: this_obj, that_obj

 this_obj=obj(ns)
 that_obj=obj(ns)

 print *, this_obj%a(3,3)

 rval = 30.
 mval = 15
 call this_obj%change_obj(mval, rval)

 print *, this_obj%a(3,3)

 print *, this_obj%a(3,3), that_obj%a(3,3)
 call that_obj%del_obj()

end program my_prog
```

11.2 Operators on Classes

It is possible to overload operators to work with class instances. The approach is similar to overloading operators within modules but for classes we use the `generic` declaration. Let us define addition, subtraction, and assignment for our example class.

```
module obj_class
implicit none

 private

 public           :: obj

 integer          :: i, j, l=5
 real           :: s = 25.
```

```
 type obj
  integer         ::  m
  real, dimension(:,:), allocatable :: a
  contains
   procedure :: change_obj
   procedure :: del_obj
   procedure :: add_obj
   procedure :: sub_obj
   procedure :: equate_obj
   generic, public :: operator(+) => add_obj
   generic, public :: operator(-) => sub_obj
   generic, public :: assignment(=) => equate_obj
end type obj

interface obj
  module procedure init_obj
end interface

contains

 type(obj) function init_obj(n)
   integer, intent(in)  :: n
   allocate (init_obj%a(n,n))
   init_obj%a = s
   init_obj%m = l
 end function init_obj

 subroutine change_obj(self, n, r)
 class(obj), intent(inout) :: self
 integer, intent(in)   :: n
 real,  intent(in)   :: r
  self%a = r
  self%m = n
 end subroutine change_obj

 subroutine del_obj(self)
 class(obj), intent(inout) :: self
  deallocate(self%a)
 end subroutine del_obj

 type(obj) function add_obj(obj1,obj2)
 class(obj), intent(in)  :: obj1, obj2
 integer       :: n
   n=size(obj1%a,1)
   add_obj=obj(n)
   add_obj%a=obj1%a+obj2%a
   add_obj%m=obj1%m+obj2%m
 end function add_obj

 type(obj) function sub_obj(obj1,obj2)
 class(obj), intent(in)  :: obj1, obj2
 integer       :: n
```

```
    n=size(obj1%a,1)
    sub_obj=obj(n)
    sub_obj%a=obj1%a-obj2%a
    sub_obj%m=obj1%m-obj2%m
   end function sub_obj

   subroutine equate_obj(new_obj, old_obj)
   class(obj), intent(in) :: old_obj
   class(obj), intent(out) :: new_obj
    new_obj%a=old_obj%a
    new_obj%m=old_obj%m
   end subroutine equate_obj

  end module obj_class
```

This class can be tested with the following driver program:

```
  program my_prog
  use obj_class
  implicit none

  type(obj):: sum_obj,diff_obj
  integer  :: mval
  real   :: rval

  integer, parameter  :: ns = 10

  type(obj) this_obj, that_obj

  this_obj=obj(ns)
  that_obj=obj(ns)

  print *, this_obj%a(3,3)

  sum_obj=this_obj+that_obj
  diff_obj=this_obj-that_obj
  print *, sum_obj%m,diff_obj%m

  rval = 30.
  mval = 15
  call this_obj%change_obj(mval, rval)

  print *, this_obj%a(3,3), that_obj%a(3,3)

  that_obj=this_obj
  print *, this_obj%a(3,3), that_obj%a(3,3)

  end program my_prog
```

In this case we see that the operator overloading occurs with the `generic` keyword and not through an explicit interface as was the case for a type. Also note that both arguments to `add_obj`, `sub_obj`, and `equate_obj` must be declared `class` and not `type`. The `class` keyword is required because type-bound

procedures can be *polymorphic*. We always declare the object with the usual keyword `type` but use `class` to allow for the possibility of inheritance.

The `generic` keyword can be used to overload other types of type-bound procedures, such as we illustrated in Section 9.2 for module procedures to create a generic function that could handle parameters of different variable type.

11.3 Inheritance and Polymorphism

The existence of a class keyword hints at another key principle of object-oriented software design: polymorphism. Polymorphism simply means that a procedure can work for more than one type. Overloading of operators and intrinsic procedures so that they can accept single and double precision and possibly also complex arguments is an example of polymorphism. In the context of object-oriented programming it usually goes hand in hand with inheritance and means that a procedure can take a type/class or any of its decendants.

11.3.1 Inheritance

We have already met type extension in Section 8.1. This is a form of inheritance; the extended type inherits all the attributes of its parent. In the case of full object-oriented programming the analogy is straightforward; when a type contains type-bound procedures any of its extensions inherit the procedures as well as the variable elements. However, in a child object the inherited type-bound procedure (or method) can be overloaded to behave differently with the child object from its behavior with the parent. Once again, an example will probably be most useful:

```
module animal_class
implicit none

private
public     ::  animal

integer, parameter ::  MAX_LEN=80

type animal
private
character(len=MAX_LEN), public  :: species
contains
  procedure,    public  :: speak
  procedure,    public  :: print_my_species
end type animal

interface animal
 module procedure init_animal
end interface

contains

type(animal) function init_animal(what_animal)
  character(len=MAX_LEN), intent(in)  :: what_animal
  init_animal%species=what_animal
end function init_animal
```

```
subroutine speak(self)
class(animal), intent(in) :: self
 print *, 'Base class does not implement this method'
end subroutine speak

subroutine print_my_species(self)
class(animal), intent(in) :: self
 print *, trim(self%species)
end subroutine print_my_species

end module animal_class

module cat_class
use animal_class
implicit none

private
public       ::  cat

integer, parameter   ::  MAX_LEN=80

type, extends(animal)  :: cat
private
character(len=MAX_LEN) :: breed
character(len=MAX_LEN) :: name
contains
  procedure    :: speak
  procedure    :: print=>print_me
end type cat

interface cat
 module procedure init_cat
end interface

contains

type(cat) function init_cat(my_name,breed)
 character(len=*), intent(in) :: breed
 character(len=*), intent(in) :: my_name
 init_cat%animal=animal('Felis sylvestris catus')
 init_cat%name=my_name
 init_cat%breed=breed
end function init_cat

subroutine print_me(self)
  class(cat), intent(in) :: self
  print *, 'I am ',trim(self%name)//",I am a "// trim(self%breed)
end subroutine print_me

subroutine speak(self)
  class(cat), intent(in) :: self
  print *, 'Meow'
end subroutine speak
```

```
  end module cat_class

module dog_class
  use animal_class
  implicit none

  private
  public       ::  dog

  integer, parameter   ::  MAX_LEN=80

  type, extends(animal)  :: dog
  private
  character(len=MAX_LEN) :: breed
  character(len=MAX_LEN) :: name
  contains
    procedure    :: speak
    procedure    :: print=>print_me
  end type dog

  interface dog
   module procedure init_dog
  end interface

  contains

  type(dog) function init_dog(my_name,breed)
   character(len=*), intent(in) :: breed
   character(len=*), intent(in) :: my_name
   init_dog%animal=animal('Canis familiaris')
   init_dog %name=my_name
   init_dog%breed=breed
  end function init_dog

  subroutine print_me(self)
   class(dog), intent(in) :: self
   print *, 'I am ',trim(self%name)//", I am a "// trim(self%breed)
  end subroutine print_me

  subroutine speak(self)
   class(dog), intent(in) :: self
   print *, 'Woof'
  end subroutine speak

end module dog_class

program testpoly
use cat_class, only : cat
use dog_class, only : dog
implicit none

 type(cat) :: Itty
 type(dog) :: Teddy

 Itty=cat('Itty','tuxedo')
```

```
 Teddy=dog('Teddy','poodle')

 call Itty%speak()
 call Teddy%speak()

 call Itty%print_my_species()

 call Itty%print()
 call Teddy%print()

end program testpoly
```

In this example the polymorphic procedure is "speak." In the "animal" class we provide a stub that informs the user that it does not implement the method. We must *override* the "speak" method in each of the derived (child) classes, and we must declare it in each child. However, we do not need to make any changes to the "print_my_species" method since it is inherited by the children.

Also note that members of the base class must be public in order for the derived classes to use them. There is a `protected` keyword in Fortran 2003 but it does not have the same meaning as in C++, where it means that descendant classes can use it but outside entities cannot. The Fortran `protected` attribute means that module variables can be read by outside units but not changed; i.e. they are "read only" outside the module; furthermore derived type members may not have the `protected` attribute.

Finally, notice that we can rename type-bound procedures if we wish, as we have done for the print_me method in the cat and dog types so that we can use a shorter name when invoking the method on an instance. We could do this in any class; it is not related to polymorphism. We can also use the => operator with the `generic` keyword to define a generic procedure, similar to our overloading of the arithmetic and assignment operators in the example in Section 11.2 above.

Example

```
  generic, public :: distance => rdist, ddist
```

We would then write procedures rdist and ddist that would accept and return real and double precision respectively.

11.3.2 Abstract Types

In our example of dog and cat, we never intend to instantiate the "animal" class. However, we would like to prescribe a set of procedures that any descendant of the "animal" class must implement. To accomplish this we can create an abstract class. The abstract class merely defines the interface for each of the methods but does not implement any of them; they are *deferred* to the descendants. It is an error for a derived class to fail to implement an abstract method. Let us make "animal" our abstract (base) class and then immediately derive a "mammal" class from which we may further derive our specific critters.

```
module animal_class
implicit none

private
```

```
public     ::  animal

integer, parameter ::  MAX_LEN=80

type, abstract   ::  animal
private
contains
  procedure (animal_speak),  deferred :: speak
  procedure (print_my_species),deferred :: print_species
  procedure (print_me)    ,deferred :: print
end type animal

abstract interface
 subroutine animal_speak(self)
 import        :: animal
 class(animal), intent(in) :: self
 end subroutine animal_speak
 subroutine print_my_species(self)
 import        :: animal
 class(animal), intent(in) :: self
 end subroutine print_my_species
 subroutine print_me(self)
 import        :: animal
 class(animal), intent(in) :: self
 end subroutine print_me
end interface

end module animal_class

module mammal_class
use animal_class
implicit none

private
public     ::  mammal

integer, parameter ::  MAX_LEN=80

type mammal
private
character(len=MAX_LEN), public  :: species
contains
  procedure,    public  :: speak
  procedure,    public  :: print_species
end type mammal

interface mammal
 module procedure init_mammal
end interface

contains

type(mammal) function init_mammal(what_animal)
 character(len=MAX_LEN), intent(in)  :: what_animal
 init_mammal%species=what_animal
```

```
end function init_mammal

subroutine speak(self)
class(mammal), intent(in) :: self
 print *, 'Grr'
end subroutine speak

subroutine print_species(self)
class(mammal), intent(in) :: self
 print *, trim(self%species)
end subroutine print_species

end module mammal_class

module cat_class
use mammal_class
implicit none

private
public       ::  cat

integer, parameter   ::  MAX_LEN=80

type, extends(mammal)  :: cat
private
character(len=MAX_LEN) :: breed
character(len=MAX_LEN) :: name
contains
  procedure    :: speak
  procedure    :: print=>print_me
end type cat

interface cat
module procedure init_cat
end interface

contains

type(cat) function init_cat(my_name,breed)
 character(len=*), intent(in) :: breed
 character(len=*), intent(in) :: my_name
 init_cat%mammal=mammal('Felis sylvestris catus')
 init_cat%name=my_name
 init_cat%breed=breed
end function init_cat

subroutine print_me(self)
 class(cat), intent(in) :: self
 print *, 'I am ',trim(self%name)//",I am a "// trim(self%breed)
end subroutine print_me

subroutine speak(self)
 class(cat), intent(in) :: self
 print *, 'Meow'
end subroutine speak
```

```
end module cat_class

module dog_class
use mammal_class
implicit none

private
public       ::  dog

integer, parameter   ::  MAX_LEN=80

type, extends(mammal)  :: dog
private
character(len=MAX_LEN) :: breed
character(len=MAX_LEN) :: name
contains
  procedure    :: speak
  procedure    :: print=>print_me
end type dog

interface dog
 module procedure init_dog
end interface

contains

type(dog) function init_dog(my_name,breed)
 character(len=*), intent(in) :: breed
 character(len=*), intent(in) :: my_name
 init_dog%mammal=mammal('Canis familiaris')
 init_dog %name=my_name
 init_dog%breed=breed
end function init_dog

subroutine print_me(self)
 class(dog), intent(in) :: self
 print *, 'I am ',trim(self%name)//", I am a "// trim(self%breed)
end subroutine print_me

subroutine speak(self)
 class(dog), intent(in) :: self
 print *, 'Woof'
end subroutine speak

end module dog_class

program testabstract
use cat_class, only : cat
use dog_class, only : dog
implicit none

type(cat) :: Itty
type(dog) :: Teddy
```

```
  Itty=cat('Itty','tuxedo')
  Teddy=dog('Teddy','poodle')

  call Itty%speak()
  call Teddy%speak()

  call Itty%print_species()

  call Itty%print()
  call Teddy%print()

  end program testpoly
```

The base class is defined with the `abstract` keyword. All its type-bound procedures are `deferred` and will be implemented by derived classes. In the interface we also use the `abstract` keyword, and we must `import` the base class. We could also derive types such as "reptile," "bird," and anything else appropriate and subclass further from those classes.

11.3.3 Type Guarding and SELECT TYPE

We may wish to write a subroutine that will work for any class in an inheritance chain. We will not know until runtime which type will be passed to this subroutine. Fortran requires that the program distinguish the types; the runtime library will not attempt to disambiguate the type on its own. Addition of code to keep the behaviors of different types separated is called *type guarding*. In Fortran we accomplish this via the `select type` construct. As before, an example should make the concept clearer. We are going to write a program to simulate juggling. Jugglers call their items "props" so our base class/type will be called "prop." We will write a subroutine `juggle` that, in a real code, would do something; in our extremely simplified example it will merely print the members. Since we will determine at runtime which items our juggler will use, we must use the `select type` construct to branch to different behavior depending on the type. If we use the keywords `type is` then that branch will be taken only if the exact type is matched; with the `class is` keywords a type or any of its extensions will match.

```
module props_class
  implicit none

  private
  public  ::  prop

  type prop
   private
   character(len=6), public :: color
 contains
   procedure, public :: throw
   procedure, public :: drop
  end type prop

contains

  subroutine throw(self)
  class(prop), intent(in) :: self
```

```
  print *, "throw"
  end subroutine throw

  subroutine drop(self)
  class(prop), intent(in) :: self
  print *, "drop"
  end subroutine drop

end module props_class


module ball_class
use props_class
implicit none

private
public  ::  ball

type, extends(prop)  :: ball
  private
   real,  public :: radius
  contains
   procedure, public :: throw
end type ball

interface ball
 module procedure init_ball
end interface

 contains

   type(ball) function init_ball(color,radius)
   character(len=*), intent(in) :: color
   real    , intent(in) :: radius
   init_ball%color=color
   init_ball%radius=radius
   end function init_ball

   subroutine throw(self)
   class(ball), intent(in) :: self
    print *, "toss"
   end subroutine throw

   subroutine drop(self)
   class(ball), intent(in) :: self
    print *, "oops"
   end subroutine drop

end module ball_class


module baton_class
use props_class
implicit none
```

```
private
public  ::  baton

type, extends(prop)  :: baton
  private
  real,   public :: length
  contains
   procedure, public :: throw
end type baton

interface baton
 module procedure init_baton
end interface

 contains

   type(baton) function init_baton(color,length)
   character(len=*), intent(in) :: color
   real    , intent(in) :: length
   init_baton%color=color
   init_baton%length=length
   end function init_baton

   subroutine throw(self)
   class(baton), intent(in) :: self
    print *, "flip"
   end subroutine throw

   subroutine drop(self)
   class(baton), intent(in) :: self
    print *, "thud"
   end subroutine drop

end module baton_class


module knife_class
use baton_class
implicit none

private
public  ::  knife

type, extends(baton) :: knife
  private
   real,  public :: blade_length
  contains
   procedure, public :: throw
end type knife

interface knife
 module procedure init_knife
end interface

 contains
```

```
   type(knife) function init_knife(color,length,blade_length)
   character(len=*), intent(in) :: color
   real    , intent(in) :: length, blade_length
   !Color is for handle in this class
   init_knife%color=color
   init_knife%length=length
   init_knife%blade_length=blade_length
   end function init_knife

   subroutine throw(self)
   class(knife), intent(in) :: self
    print *, "fling"
   end subroutine throw

   subroutine drop(self)
   class(knife), intent(in) :: self
    print *, "ouch"
   end subroutine drop

end module knife_class


program juggler
use ball_class
use baton_class
use knife_class
implicit none

type(ball),  dimension(:), allocatable :: balls
type(baton), dimension(:), allocatable :: batons
type(knife), dimension(:), allocatable :: knives

integer            :: nballs,nbatons,nknives
integer            :: nprops

character(len=6), dimension(6)   :: colors
integer            :: ncolors, color_index

integer            :: i

interface
subroutine juggle(the_prop)
use props_class
use ball_class
use baton_class
use knife_class
implicit none
class(prop)  :: the_prop
end subroutine juggle
end interface

  ! Would read these in a real code
  nballs=0
  nbatons=2
```

```
  nknives=1

  colors=(/'red ','blue  ','green ','yellow','orange','purple'/)
  ncolors=size(colors)

  allocate(balls(nballs),batons(nbatons),knives(nknives))

  do i=1,nballs
   color_index=mod(1+mod(i,nballs),ncolors)
   balls(i)=ball(colors(color_index),0.3)
  enddo

  do i=1,nbatons
   color_index=mod(1+mod(i,nbatons),ncolors)
   batons(i)=baton(colors(color_index),0.6)
  enddo

  do i=1,nknives
   color_index=mod(1+mod(i,nbatons),ncolors)
   knives(i)=knife(colors(color_index),0.5,1.0)
  enddo

  do i=1,nballs
   call juggle(balls(i))
  enddo

  do i=1,nbatons
   call juggle(batons(i))
  enddo

  do i=1,nknives
   call juggle(knives(i))
  enddo

end program juggler


subroutine juggle(the_prop)
use props_class
use ball_class
use baton_class
use knife_class
implicit none

class(prop)  :: the_prop

 !All members of the chain have a color
 print *, 'The color is ', the_prop%color

 select type(the_prop)
 type is (ball)
   print *, 'The radius is ',the_prop%radius
 type is (baton)
   print *, 'The length is ',the_prop%length
 class is (baton)
```

```
   print *, 'The length is ',the_prop%length
   print *, 'This could be a descendant of baton'
 class default
   print *, 'I only know the color '
 end select

end subroutine juggle
```

Note that in this example the class members such as radius and length are public so that the using unit can print them. This violates the principle of data hiding since the external unit can now also change the attributes. Ideally the programmer should write *accessor* ("getter") and *mutator* ("setter") procedures that return or change the type member variables.

Example

```
print *, the_prop%get_radius()
```

where the `get_radius` method of type `ball` would look something like

```
real function get_radius(self)
   get_radius=self%radius
end function get_radius
```

11.4 Program Design

Object-oriented programming is not a panacea for software engineering, but its core principles of data protection, interface management, and grouping of related code can be applied even to procedurally-oriented code. If we bundle our data and procedures into modules and classes, we can control access to those data and procedures. We can prevent any possibility of unwanted changes by outside program units. We can make software much easier to read and to modify. If we do not change the interface that we expose to the outside program units, we can make changes to the module or to the class without affecting the units that use it.

Good software design involves careful thought about appropriate objects before beginning to write code. Choosing the right objects can make the code much simpler or, if done poorly, more complex. Objects should be neither trivial nor overly complicated. In Fortran it is also frequently more appropriate to use a module without a class than to introduce an unnatural class. For example, a collection of random-number routines does not necessarily need to be a class but would naturally form a module. However, most programs have at least some natural objects. The developer should consider the nouns in a description of the program. For example, if the program models an automobile, that would naturally describe one of the important classes. Design beyond the most important class depends on the nature of the program, the programming team's judgment, and other subjective factors—for instance, should the parts of the auto (doors, tires, etc.) be attributes or should they themselves be types or classes?

In designing an object the programmer can draw a simple diagram laying out the attributes of the object and its behaviors (the methods). More complicated programs can fully implement UML (Unified Markup Language), which is a set of standardized diagrams for describing objects and their interactions. In any case, it is important to spend some time considering the design before a large investment in coding is made; the choice of types/classes can heavily influence the ease of development and maintenance.

Exercises

1. Write a class module that describes a grid for a numerical algorithm. The grid should be two-dimensional (x,y). The object should include a definition of the grid type that includes variables for the number of points in each dimension, the numerical coordinates of the first and last points in each dimension, and the distance Δx, Δy between each point in each dimension (assumed constant, but could differ between x and y). The object should contain methods that create the grid, compute Δx and Δy, and multiply the number of grid points in each dimension by some factor with a corresponding change to Δx and Δy. Overload the assignment operator to copy a grid to another grid. Write a main program that uses the object.
2. Write a class module that implements a `bird_species` object. This object will have attributes such as genus name, species name, incubation period, mean clutch size, diet preference, and average life span. The methods should include procedures `hatch`, which returns a logical (i.e. did the chick survive); `fledge`, which also returns a logical for survival or not, and `forage`, which returns something to indicate whether the bird found food and if so, what it was. Write another module `bird` which extends `bird_species` to describe an individual of a particular species. The attributes of each instance of this object will be those relevant to an individual, including age, gender, and whether it is currently breeding. Write a constructor for each type. Write a short program to test your modules.
3. Write an abstract type `amniote` with attributes order, genus name, species name, diet, average lifespan, and a flag indicating diurnal or nocturnal. Write interfaces for deferred methods `birth`, `breeding`, `forage`, and `death`. The birth and death methods should return logicals; the breeding method should return a logical and number of offspring (which may be zero if the animal is not breeding), and forage should return a logical and if true, what the animal ate. Extend the abstract type with types `reptile`, `bird`, and `mammal`. Include for reptiles at least an attribute to indicate whether it is a turtle, a lizard or snake, or a crocodilian. For birds include at least an attribute indicating whether it is flightless, passerine (perching birds), a pheasant-like bird, or a waterfowl. For mammals indicate whether it is a monotreme (platypus, echidna), marsupial, or placental. Implement at least one polymorphic non-deferred method `feed` which indicates for breeding animals how it feeds its young ("self-feeder" should be a possibility). Include a constructor for each implemented type. The specified methods are minimums and the student should be creative in developing a crude model of an animal that could be used in a larger program such as an ecosystem model. Write a test unit to exercise the objects.

References

1. Clerman, Norman S. and Walter Spector. Modern Fortran: Style and Usage. Cambridge: Cambridge University Press, 2012.
2. Metcalf, Michael, John Reid, and Malcolm Cohen. *Fortran 95/2003 Explained*. Oxford: Oxford University Press, 2004.
3. Metcalf, Michael, John Reid, and Malcolm Cohen. *Modern Fortran Explained*. Oxford: Oxford University Press, 2011.
4. Rouson, Damian, Jim Xia and Xiaofeng Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge: Cambridge University Press, 2011.