Chapter 21. Introduction to Fortran 90 Language Features

21.0 Introduction

Fortran 90 is in many respects a backwards-compatible modernization of the long-used (and much abused) Fortran 77 language, but it is also, in other respects, a new language for parallel programming on present and future multiprocessor machines. These twin design goals of the language sometimes add confusion to the process of becoming fluent in Fortran 90 programming.

In a certain trivial sense, Fortran 90 is strictly backwards-compatible with Fortran 77. That is, any Fortran 90 compiler is supposed to be able to compile any legacy Fortran 77 code without error. The reason for terming this compatibility trivial, however, is that you have to tell the compiler (usually via a source file name ending in "`.f`" or "`.for`") that it is dealing with a Fortran 77 file. If you instead try to pass off Fortran 77 code as native Fortran 90 (e.g., by naming the source file something ending in "`.f90`") it will not always work correctly!

It is best, therefore, to approach Fortran 90 as a new computer language, albeit one with a lot in common with Fortran 77. Indeed, in such terms, Fortran 90 is a fairly *big* language, with a large number of new constructions and intrinsic functions. Here, in one short chapter, we do not pretend to provide a complete description of the language. Luckily, there are good books that do exactly that. Our favorite one is by Metcalf and Reid [1], cited throughout this chapter as "M&R." Other good starting points include [2] and [3].

Our goal, in the remainder of this chapter, is to give a good, working description of those Fortran 90 language features that are not immediately self-explanatory to Fortran 77 programmers, with particular emphasis on those that occur most frequently in the Fortran 90 versions of the Numerical Recipes routines. This chapter, by itself, will not teach you to write Fortran 90 code. But it ought to help you acquire a reading knowledge of the language, and perhaps provide enough of a head start that you can rapidly pick up the rest of what you need to know from M&R or another Fortran 90 reference book.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press). [1]

Kerrigan, J.F. 1993, *Migrating to Fortran 90* (Sebastopol, CA: O'Reilly). [2]

Brainerd, W.S., Goldberg, C.H., and Adams, J.C. 1996, *Programmer's Guide to Fortran 90*, 3rd ed. (New York: Springer-Verlag). [3]

21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines

This section is for people who want to jump right in. We'll compute a Bessel function $J_0(x)$, where x is equal to the fourth root of the Julian Day number of the 200th full moon since January 1900. (Now *there's* a useful quantity!)

First, locate the important files `nrtype.f90`, `nrutil.f90`, and `nr.f90`, as listed in Appendices C1, C1, and C2, respectively. These contain *modules* that either are (i) used by our routines, or else (ii) describe the calling conventions of our routines to (your) user programs. Compile each of these files, producing (with most compilers) a `.mod` file and a `.o` (or similarly named) file for each one.

Second, create this main program file:

```
PROGRAM hello_bessel
USE nrtype
USE nr, ONLY: flmoon, bessj0
IMPLICIT NONE
INTEGER(I4B) :: n=200,nph=2,jd
REAL(SP) :: x,frac,ans
call flmoon(n,nph,jd,frac)
x=jd**0.25_sp
ans=bessj0(x)
write (*,*) 'Hello, Bessel: ', ans
END PROGRAM
```

Here is a quick explanation of some elements of the above program:

The first `USE` statement includes a module of ours named `nrtype`, whose purpose is to give symbolic names to some kinds of data types, among them single-precision reals ("`sp`") and four-byte integers ("`i4b`"). The second `USE` statement includes a module of ours that defines the calling sequences, and variable types, expected by (in this case) the Numerical Recipes routines `flmoon` and `bessj0`.

The `IMPLICIT NONE` statement signals that we want the compiler to require us explicitly to declare all variable types. *We strongly urge that you always take this option.*

The next two lines declare integer and real variables of the desired kinds. The variable `n` is initialized to the value 200, `nph` to 2 (a value expected by `flmoon`).

We call `flmoon`, and take the fourth root of the answer it returns as `jd`. Note that the constant 0.25 is typed to be single-precision by the appended `_sp`.

We call the `bessj0` routine, and print the answer.

Third, compile the main program file, and also the files `flmoon.f90`, `bessj0.f90`. Then, link the resulting object files with also `nrutil.o` (or similar system-dependent name, as produced in step 1). Some compilers will also require you to link with `nr.o` and `nrtype.o`.

Fourth, run the resulting executable file. Typical output is:

```
Hello, Bessel:   7.3096365E-02
```

21.2 Fortran 90 Language Concepts

The Fortran 90 language standard defines and uses a number of standard terms for concepts that occur in the language. Here we summarize briefly some of the most important concepts. Standard Fortran 90 terms are shown in *italics*. While by no means complete, the information in this section should help you get a quick start with your favorite Fortran 90 reference book or language manual.

A note on capitalization: Outside a character context, Fortran 90 is not case-sensitive, so you can use upper and lower case any way you want, to improve readability. A variable like `SP` (see below) is the same variable as the variable `sp`. We like to capitalize keywords whose use is primarily at compile-time (statements that delimit program and subprogram boundaries, declaration statements of variables, fixed parameter values), and use lower case for the bulk of run-time code. You can adopt any convention that you find helpful to your own programming style; but we strongly urge you to adopt and follow *some* convention.

Data Types and Kinds

Data types (also called simply *types*) can be either *intrinsic data types* (the familiar `INTEGER`, `REAL`, `LOGICAL`, and so forth) or else *derived data types* that are built up in the manner of what are called "structures" or "records" in other computer languages. (We'll use derived data types very sparingly in this book.) Intrinsic data types are further specified by their *kind parameter* (or simply *kind*), which is simply an integer. Thus, on many machines, `REAL(4)` (with `kind` $= 4$) is a single-precision real, while `REAL(8)` (with `kind` $= 8$) is a double-precision real. *Literal constants* (or simply *literals*) are specified as to kind by appending an underscore, as `1.5_4` for single precision, or `1.5_8` for double precision. [M&R, §2.5–§2.6]

Unfortunately, the specific integer values that define the different kind types are not specified by the language, but can vary from machine to machine. For that reason, one almost never uses literal kind parameters like 4 or 8, but rather defines in some central file, and imports into all one's programs, symbolic names for the kinds. For this book, that central file is the *module* named `nrtype`, and the chosen symbolic names include `SP`, `DP` (for reals); `I2B`, `I4B` (for two- and four-byte integers); and `LGT` for the default logical type. You will therefore see us consistently writing `REAL(SP)`, or `1.5_sp`, and so forth.

Here is an example of declaring some variables, including a one-dimensional array of length 500, and a two-dimensional array with 100 rows and 200 columns:

```
USE nrtype
REAL(SP) :: x,y,z
INTEGER(I4B) :: i,j,k
REAL(SP), DIMENSION(500) :: arr
REAL(SP), DIMENSION(100,200) :: barr
REAL(SP) :: carr(500)
```

The last line shows an alternative form for array syntax. And yes, there *are* default kind parameters for each intrinsic type, but these vary from machine to machine and can get you into trouble when you try to move code. We therefore specify all kind parameters explicitly in almost all situations.

Array Shapes and Sizes

The *shape* of an *array* refers to both its dimensionality (called its *rank*), and also the lengths along each dimension (called the *extents*). The shape of an array is specified by a rank-one array whose elements are the extents along each dimension, and can be queried with the `shape` intrinsic (see p. 949). Thus, in the above example, `shape(barr)` returns an array of length 2 containing the values $(100, 200)$.

The *size* of an array is its total number of elements, so the intrinsic `size(barr)` would return 20000 in the above example. More often one wants to know the extents along each dimension, separately: `size(barr,1)` returns the value 100, while `size(barr,2)` returns the value 200. [M&R, §2.10]

Section §21.3, below, discusses additional aspects of arrays in Fortran 90.

Memory Management

Fortran 90 is greatly superior to Fortran 77 in its memory-management capabilities, seen by the user as the ability to create, expand, or contract workspace for programs. Within *subprograms* (that is, *subroutines* and *functions*), one can have *automatic arrays* (or other *automatic data objects*) that come into existence each time the subprogram is entered, and disappear (returning their memory to the pool) when the subprogram is exited. The size of automatic objects can be specified by arbitrary expressions involving values passed as *actual arguments* in the calling program, and thus received by the subprogram through its corresponding *dummy arguments*. [M&R, §6.4]

Here is an example that creates some automatic workspace named `carr`:

```
SUBROUTINE dosomething(j,k)
USE nrtype
REAL(SP), DIMENSION(2*j,k**2) :: carr
```

Finer control on when workspace is created or destroyed can be achieved by declaring *allocatable arrays*, which exist as names only, without associated memory, until they are *allocated* within the program or subprogram. When no longer needed, they can be *deallocated*. The *allocation status* of an allocatable array can be tested by the program via the `allocated` intrinsic function (p. 952). [M&R, §6.5]

Here is an example in outline:

```
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: darr
...
allocate(darr(10,20))
...
deallocate(darr)
...
allocate(darr(100,200))
...
deallocate(darr)
```

Notice that `darr` is originally declared with only "slots" (colons) for its dimensions, and is then allocated/deallocated twice, with different sizes.

Yet finer control is achieved by the use of *pointers*. Like an allocatable array, a pointer can be allocated, at will, its own associated memory. However, it has the additional flexibility of alternatively being *pointer associated* with a *target* that

already exists under another name. Thus, pointers can be used as redefinable aliases for other variables, arrays, or (see §21.3) *array sections*. [M&R, §6.12]

Here is an example that first associates the pointer `parr` with the array `earr`, then later cancels that association and allocates it its own storage of size 50:

```
REAL(SP), DIMENSION(:), POINTER :: parr
REAL(SP), DIMENSION(100), TARGET :: earr
...
parr => earr
...
nullify(parr)
allocate(parr(50))
...
deallocate(parr)
```

Procedure Interfaces

When a procedure is *referenced* (e.g., called) from within a program or subprogram (examples of *scoping units*), the scoping unit must be told, or must deduce, the procedure's *interface*, that is, its calling sequence, including the types and kinds of all dummy arguments, returned values, etc. The recommended procedure is to specify this interface via an *explicit interface*, usually an *interface block* (essentially a declaration statement for subprograms) in the calling subprogram or in some *module* that the calling program includes via a `USE` statement. In this book all interfaces are explicit, and the module named `nr` contains interface blocks for all of the Numerical Recipes routines. [M&R, §5.11]

Here is a typical example of an interface block:

```
INTERFACE
   SUBROUTINE caldat(julian,mm,id,iyyy)
   USE nrtype
   INTEGER(I4B), INTENT(IN) :: julian
   INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
   END SUBROUTINE caldat
END INTERFACE
```

Once this interface is made known to a program that you are writing (by either explicit inclusion or a `USE` statement), then the compiler is able to flag for you a variety of otherwise difficult-to-find bugs. Although interface blocks can sometimes seem overly wordy, they give a big payoff in ultimately minimizing programmer time and frustration.

For compatibility with Fortran 77, the language also allows for *implicit interfaces*, where the calling program tries to figure out the interface by the old rules of Fortran 77. These rules are quite limited, and prone to producing devilishly obscure program bugs. We strongly recommend that implicit interfaces never be used.

Elemental Procedures and Generic Interfaces

Many *intrinsic procedures* (those defined by the language standard and thus usable without any further definition or specification) are also *generic*. This means that a single procedure name, such as `log(x)`, can be used with a variety of types and kind parameters for the argument `x`, and the result returned will have the same type and kind parameter as the argument. In this example, `log(x)` allows any real or complex argument type.

Better yet, most generic functions are also *elemental*. The argument of an elemental function can be an array of arbitrary shape! Then, the returned result is an array of the same shape, with each element containing the result of applying the function to the corresponding element of the argument. (Hence the name *elemental*, meaning "applied element by element.") [M&R, §8.1] For example:

```
REAL(SP), DIMENSION(100,100) :: a,b
b=sin(a)
```

Fortran 90 has no facility for creating new, user-defined elemental functions. It does have, however, the related facility of *overloading* by the use of *generic interfaces*. This is invoked by the use of an interface block that attaches a single *generic name* to a number of distinct subprograms whose dummy arguments have different types or kinds. Then, when the generic name is referenced (e.g., called), the compiler chooses the specific subprogram that matches the types and kinds of the actual arguments used. [M&R, §5.18] Here is an example of a generic interface block:

```
INTERFACE myfunc
   FUNCTION myfunc_single(x)
   USE nrtype
   REAL(SP) :: x,myfunc_single
   END FUNCTION myfunc_single

   FUNCTION myfunc_double(x)
   USE nrtype
   REAL(DP) :: x,myfunc_double
   END FUNCTION myfunc_double
END INTERFACE
```

A program with knowledge of this interface could then freely use the function reference `myfunc(x)` for `x`'s of both type `SP` and type `DP`.

We use overloading quite extensively in this book. A typical use is to provide, under the same name, both scalar and vector versions of a function such as a Bessel function, or to provide both single-precision and double-precision versions of procedures (as in the above example). Then, to the extent that we have provided all the versions that you need, you can pretend that our routine is elemental. In such a situation, if you ever call our function with a type or kind that we have *not* provided, the compiler will instantly flag the problem, because it is unable to resolve the generic interface.

Modules

Modules, already referred to several times above, are Fortran 90's generalization of Fortran 77's common blocks, `INCLUDE`d files of parameter statements, and (to some extent) statement functions. Modules are *program units*, like main programs or subprograms (subroutines and functions), that can be separately compiled. A module is a convenient place to stash global data, *named constants* (what in Fortran 77 are called "symbolic constants" or "`PARAMETER`s"), interface blocks to subprograms and/or actual subprograms themselves (*module subprograms*). The convenience is that a module's information can be incorporated into another program unit via a simple, one-line `USE` statement. [M&R, §5.5]

Here is an example of a simple module that defines a few parameters, creates some global storage for an array named `arr` (as might be done with a Fortran 77 common block), and defines the interface to a function `yourfunc`:

```
MODULE mymodule
   USE nrtype
   REAL(SP), PARAMETER :: con1=7.0_sp/3.0_sp,con2=10.0_sp
   INTEGER(I4B), PARAMETER :: ndim=10,mdim=9
   REAL(SP), DIMENSION(ndim,mdim) :: arr
   INTERFACE
      FUNCTION yourfunc(x)
      USE nrtype
      REAL(SP) :: x,yourfunc
      END FUNCTION yourfunc
   END INTERFACE
END MODULE mymodule
```

As mentioned earlier, the module `nr` contains `INTERFACE` declarations for all the Numerical Recipes. When we include a statement of the form

```
USE nr, ONLY: recipe1
```

it means that the program uses the additional routine `recipe1`. The compiler is able to use the explicit interface declaration in the module to check that `recipe1` is invoked with arguments of the correct type, shape, and number. However, a weakness of Fortran 90 is that there is no fail-safe way to be sure that the interface module (here `nr`) stays synchronized with the underlying routine (here `recipe1`). You might think that you could accomplish this by putting `USE nr, ONLY: recipe1` into the `recipe1` program itself. Unfortunately, the compiler interprets this as an erroneous double definition of `recipe1`'s interface, rather than (as would be desirable) as an opportunity for a consistency check. (To achieve this kind of consistency check, you can put the procedures themselves, not just their interfaces, into the module.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.3 More on Arrays and Array Sections

Arrays are the central conceptual core of Fortran 90 as a *parallel* programming language, and thus worthy of some further discussion. We have already seen that arrays can "come into existence" in Fortran 90 in several ways, either directly declared, as

```
REAL(SP), DIMENSION(100,200) :: arr
```

or else allocated by an *allocatable* variable or a *pointer* variable,

```
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: arr
REAL(SP), DIMENSION(:,:), POINTER :: barr
...
allocate(arr(100,200),barr(100,200))
```

or else (not previously mentioned) passed into a subprogram through a dummy argument:

```
SUBROUTINE myroutine(carr)
USE nrtype
REAL(SP), DIMENSION(:,:) :: carr
...
i=size(carr,1)
```

```
j=size(carr,2)
```

In the above example we also show how the subprogram can find out the size of the actual array that is passed, using the `size` intrinsic. This routine is an example of the use of an *assumed-shape array*, new to Fortran 90. The actual extents along each dimension are inherited from the calling routine at run time. A subroutine with assumed-shape array arguments *must* have an explicit interface in the calling routine, otherwise the compiler doesn't know about the extra information that must be passed. A typical setup for calling `myroutine` would be:

```
PROGRAM use_myroutine
USE nrtype
REAL(SP), DIMENSION(10,10) :: arr
INTERFACE
   SUBROUTINE myroutine(carr)
   USE nrtype
   REAL(SP), DIMENSION(:,:) :: carr
   END SUBROUTINE myroutine
END INTERFACE
...
call myroutine(a)
```

Of course, for the recipes we have provided all the interface blocks in the file `nr.f90`, and you need only a `USE nr` statement in your calling program.

Conformable Arrays

Two arrays are said to be *conformable* if their shapes are the same. Fortran 90 allows practically all operations among conformable arrays and elemental functions that are allowed for scalar variables. Thus, if `arr`, `barr`, and `carr` are mutually conformable, we can write,

```
arr=barr+cos(carr)+2.0_sp
```

and have the indicated operations performed, element by corresponding element, on the entire arrays. The above line also illustrates that a scalar (here the constant `2.0_sp`, but a scalar variable would also be fine) is deemed conformable with *any* array — it gets "expanded" to the shape of the rest of the expression that it is in. [M&R, §3.11]

In Fortran 90, as in Fortran 77, the default lower bound for an array subscript is 1; however, it can be made some other value at the time that the array is declared:

```
REAL(SP), DIMENSION(100,200) :: farr
REAL(SP), DIMENSION(0:99,0:199) :: garr
...
farr = 3.0_sp*garr + 1.0_sp
```

Notice that `farr` and `garr` are conformable, since they have the same shape, in this case $(100, 200)$. Also note that when they are used in an array expression, the operations are taken between the corresponding elements *of their shapes*, not necessarily the corresponding elements *of their indices*. [M&R, §3.10] In other words, one of the components evaluated is,

```
farr(1,1) = 3.0_sp*garr(0,0) + 1.0_sp
```

This illustrates a fundamental aspect of array (or data) parallelism in Fortran 90. Array constructions should *not* be thought of as merely abbreviations for do-loops

over indices, but rather as genuinely parallel operations on same-shaped objects, abstracted of their indices. This is why the standard makes no statement about the order in which the individual operations in an array expression are executed; they might in fact be carried out simultaneously, on parallel hardware.

By default, array expressions and assignments are performed for all elements of the same-shaped arrays referenced. This can be modified, however, by use of a `where` construction like this:

```
where (harr > 0.0_sp)
   farr = 3.0_sp*garr + 1.0_sp
end where
```

Here `harr` must also be conformable to `farr` and `garr`. Analogously with the Fortran `if`-statement, there is also a one-line form of the `where`-statement. There is also a `where ... elsewhere ... end where` form of the statement, analogous to `if ... else if ... end if`. A significant language limitation in Fortran 90 is that nested `where`-statements are not allowed. [M&R, §6.8]

Array Sections

Much of the versatility of Fortran 90's array facilities stems from the availability of *array sections*. An array section acts like an array, but its memory location, and thus the values of its elements, is actually a subset of the memory location of an already-declared array. Array sections are thus "windows into arrays," and they can appear on either the left side, or the right side, or both, of a replacement statement. Some examples will clarify these ideas.

Let us presume the declarations

```
REAL(SP), DIMENSION(100) :: arr
INTEGER(I4B), DIMENSION(6) :: iarr=(/11,22,33,44,55,66/)
```

Note that `iarr` is not only declared, it is also initialized by an *initialization expression* (a replacement for Fortran 77's `DATA` statement). [M&R, §7.5] Here are some array sections constructed from these arrays:

Array Section	*What It Means*
`arr(:)`	same as `arr`
`arr(1:100)`	same as `arr`
`arr(1:10)`	one-dimensional array containing first 10 elements of `arr`
`arr(51:100)`	one-dimensional array containing second half of `arr`
`arr(51:)`	same as `arr(51:100)`
`arr(10:1:-1)`	one-dimensional array containing first 10 elements of `arr`, but in *reverse order*
`arr( (/10,99,1,6/) )`	one-dimensional array containing elements 10, 99, 1, and 6 of `arr`, in that order
`arr(iarr)`	one-dimensional array containing elements 11, 22, 33, 44, 55, 66 of `arr`, in that order

Now let's try some array sections of the two-dimensional array

```
REAL(SP), DIMENSION(100,100) :: barr
```

Array Section	*What It Means*
`barr(:,:)`	same as `barr`
`barr(1:100,1:100)`	same as `barr`
`barr(7,:)`	one-dimensional array containing the 7th row of `barr`
`barr(7,1:100)`	same as `barr(7,:)`
`barr(:,7)`	one-dimensional array containing the 7th column of `barr`
`barr(21:30,71:90)`	two-dimensional array containing the sub-block of `barr` with the indicated ranges of indices; the shape of this array section is $(10, 20)$
`barr(100:1:-1,100:1:-1)`	two-dimensional array formed by flipping `barr` upside down and backwards
`barr(2:100:2,2:100:2)`	two-dimensional array of shape $(50, 50)$ containing the elements of `barr` whose row and column indices are both even

Some terminology: A construction like `2:100:2`, above, is called a *subscript triplet*. Its integer pieces (which may be integer constants, or more general integer expressions) are called *lower*, *upper*, and *stride*. Any of the three may be omitted. An omitted stride defaults to the value 1. Notice that, if $(upper - lower)$ has a different sign from *stride*, then a subscript triplet defines an empty or *zero-length* array, e.g., `1:5:-1` or `10:1:1` (or its equivalent form, simply `10:1`). Zero-length arrays are not treated as errors in Fortran 90, but rather as "no-ops." That is, no operation is performed in an expression or replacement statement among zero-length arrays. (This is essentially the same convention as in Fortran 77 for do-loop indices, which array expressions often replace.) [M&R, §6.10]

It is important to understand that array sections, when used in array expressions, match elements with other parts of the expression *according to shape*, not according to indices. (This is exactly the same principle that we applied, above, to arrays with subscript lower bounds different from the default value of 1.) One frequently exploits this feature in using array sections to carry out operations on arrays that access neighboring elements. For example,

```
carr(1:n-1,1:n-1) = barr(1:n-1,1:n-1)+barr(2:n,2:n)
```

constructs in the $(n-1) \times (n-1)$ matrix `carr` the sum of each of the corresponding elements in $n \times n$ `barr` added to its diagonally lower-right neighbor.

Pointers are often used as aliases for array sections, especially if the same array sections are used repeatedly. [M&R, §6.12] For example, with the setup

```
REAL(SP), DIMENSION(:,:), POINTER :: leftb,rightb
```

```
leftb=>barr(1:n-1,1:n-1)
rightb=>barr(2:n,2:n)
```

the statement above can be coded as

```
carr(1:n-1,1:n-1)=leftb+rightb
```

We should also mention that array sections, while powerful and concise, are sometimes not quite powerful enough. While any row or column of a matrix is easily accessible as an array section, there is no good way, in Fortran 90, to access (e.g.) the diagonal of a matrix, even though its elements are related by a linear progression in the Fortran storage order (by columns). These so-called *skew-sections* were much discussed by the Fortran 90 standards committee, but they were not implemented. We will see examples later in this volume of work-around programming tricks (none totally satisfactory) for this omission. (Fortran 95 corrects the omission; see §21.6.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.4 Fortran 90 Intrinsic Procedures

Much of Fortran 90's power, both for parallel programming and for its concise expression of algorithmic ideas, comes from its rich set of intrinsic procedures. These have the effect of making the language "large," hence harder to learn. However, effort spent on learning to use the intrinsics — particularly some of their more obscure, and more powerful, optional arguments — is often handsomely repaid.

This section summarizes the intrinsics that we find useful in numerical work. We omit, here, discussion of intrinsics whose exclusive use is for character and string manipulation. We intend only a summary, not a complete specification, which can be found in M&R's Chapter 8, or other reference books.

If you find the sheer number of new intrinsic procedures daunting, you might want to start with our list of the "top 10" (with the number of different Numerical Recipes routines that use each shown in parentheses): `size` (254), `sum` (44), `dot_product` (31), `merge` (27), `all` (25), `maxval` (23), `matmul` (19), `pack` (18), `any` (17), and `spread` (15). (Later, in Chapter 23, you can compare these numbers with our frequency of using the short utility functions that we define in a module named `nrutil` — several of which we think ought to have been included as Fortran 90 intrinsic procedures.)

The type, kind, and shape of the value returned by intrinsic functions will usually be clear from the short description that we give. As an additional hint (though not necessarily a precise description), we adopt the following codes:

Hint	*What It Means*
[Int]	an `INTEGER` kind type
[Real]	a `REAL` kind type
[Cmplx]	a `COMPLEX` kind type
[Num]	a numerical type and kind
[Lgcl]	a `LOGICAL` kind type
[Iarr]	a one-dimensional `INTEGER` array
[argTS]	same type and shape as the first argument
[argT]	same type as the first argument, but not necessarily the same shape

Numerical Elemental Functions

Little needs to be said about the numerical functions with identical counterparts in Fortran 77: `abs`, `acos`, `aimag`, `asin`, `atan`, `atan2`, `conjg`, `cos`, `cosh`, `dim`, `exp`, `log`, `log10`, `max`, `min`, `mod`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`. In Fortran 90 these are all *elemental* functions, so that any plausible type, kind, and shape of argument may be used. Except for `aimag`, which returns a real type from a complex argument, these all return [argTS] (see table above).

Although Fortran 90 recognizes, for compatibility, Fortran 77's so-called *specific names* for these functions (e.g., `iabs`, `dabs`, and `cabs` for the generic `abs`), these are entirely superfluous and should be avoided.

Fortran 90 corrects some ambiguity (or at least inconvenience) in Fortran 77's `mod(a,p)` function, by introducing a new function `modulo(a,p)`. The functions are essentially identical for positive arguments, but for negative `a` and positive `p`, `modulo` gives results more compatible with one's mathematical expectation that the answer should always be in the positive range 0 to `p`. E.g., `modulo(11,5)=1`, and `modulo(-11,5)=4`. [M&R, §8.3.2]

Conversion and Truncation Elemental Functions

Fortran 90's conversion (or, in the language of C, casting) and truncation functions are generally modeled on their Fortran 77 antecedents, but with the addition of an optional second integer argument, `kind`, that determines the kind of the result. Note that, if kind is omitted, you get a default kind — not necessarily related to the kind of your argument. The kind of the argument is of course known to the compiler by its previous declaration. Functions in this category (see below for explanation of arguments in slanted type) are:

[Real] `aint(a,`*`kind`*`)`
: Truncate to integer value, return as a real kind.

[Real] `anint(a,`*`kind`*`)`
: Nearest whole number, return as a real kind.

[Cmplx] `cmplx(x,`*`y,kind`*`)`

Convert to complex kind. If y is omitted, it is taken to be 0.

[Int] `int(a,`*`kind`*`)`
Convert to integer kind, truncating towards zero.

[Int] `nint(a,`*`kind`*`)`
Convert to integer kind, choosing the nearest whole number.

[Real] `real(a,`*`kind`*`)`
Convert to real kind.

[Lgcl] `logical(a,`*`kind`*`)`
Convert one logical kind to another.

We must digress here to explain the use of *optional arguments* and *keywords* as Fortran 90 language features. [M&R, §5.13] When a routine (either intrinsic or user-defined) has arguments that are declared to be optional, then the dummy names given to them also become keywords that distinguish — independent of their position in a calling list — which argument is intended to be passed. (There are some additional rules about this that we will not try to summarize here.) In this section's tabular listings, we indicate optional arguments in intrinsic routines by printing them in smaller slanted type. For example, the intrinsic function

`eoshift(array,shift,`*`boundary,dim`*`)`

has two required arguments, `array` and `shift`, and two optional arguments, `boundary` and `dim`. Suppose we want to call this routine with the actual arguments `myarray`, `myshift`, and `mydim`, but omitting the argument in the `boundary` slot. We do this by the expression

```
eoshift(myarray,myshift,dim=mydim)
```

Conversely, if we wanted a `boundary` argument, but no `dim`, we might write

```
eoshift(myarray,myshift,boundary=myboundary)
```

It is always a good idea to use this kind of keyword construction when invoking optional arguments, even though the rules allow keywords to be omitted in some unambiguous cases. Now back to the lists of intrinsic routines.

A peculiarity of the `real` function derives from its use both as a type conversion and for extracting the real part of complex numbers (related, but not identical, usages): If the argument of `real` is complex, and `kind` is omitted, then the result *isn't* a default real kind, but rather *is* (as one generally would want) the `real` kind type corresponding to the kind type of the complex argument, that is, single-precision real for single-precision complex, double-precision for double-precision, and so on. [M&R, §8.3.1] We recommend *never* using `kind` when you intend to extract the real part of a complex, and *always* using `kind` when you intend conversion of a real or integer value to a particular kind of `REAL`. (Use of the deprecated function `dble` is not recommended.)

The last two conversion functions are the exception in that they *don't* allow a `kind` argument, but rather return default integer kinds. (The X3J3 standards committee has fixed this in Fortran 95.)

[Int] `ceiling(a)`
Convert to integer, truncating towards more positive.

[Int] `floor(a)`
Convert to integer, truncating towards more negative.

Reduction and Inquiry Functions on Arrays

These are mostly the so-called *transformational functions* that accept array arguments and return either scalar values or else arrays of lesser rank. [M&R, §8.11] With no optional arguments, such functions act on all the elements of their single array argument, regardless of its shape, and produce a scalar result. When the optional argument `dim` is specified, they instead act on all one-dimensional sections that span the dimension `dim`, producing an answer one rank lower than the first argument (that is, omitting the `dim` dimension from its shape). When the optional argument `mask` is specified, only the elements with a corresponding true value in `mask` are scanned.

[Lgcl] `all(mask,`*dim*`)`
Returns true if all elements of `mask` are true, false otherwise.

[Lgcl] `any(mask,`*dim*`)`
Returns true if any of the elements of `mask` are true, false otherwise.

[Int] `count(mask,`*dim*`)`
Counts the true elements in `mask`.

[Num] `maxval(array,`*dim,mask*`)`
Maximum value of the array elements.

[Num] `minval(array,`*dim,mask*`)`
Minimum value of the array elements.

[Num] `product(array,`*dim,mask*`)`
Product of the array elements.

[Int] `size(array,`*dim*`)`
Size (total number of elements) of `array`, or its extent along dimension `dim`.

[Num] `sum(array,`*dim,mask*`)`
Sum of the array elements.

The use of the `dim` argument can be confusing, so an example may be helpful. Suppose we have

$$\texttt{myarray} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

where, as always, the `i` index in `array(i,j)` numbers the rows while the `j` index numbers the columns. Then

$$\texttt{sum(myarray,dim=1)} = (15, 18, 21, 24)$$

that is, the `i` indices are "summed away" leaving only a `j` index on the result; while

$$\texttt{sum(myarray,dim=2)} = (10, 26, 42)$$

that is, the `j` indices are "summed away" leaving only an `i` index on the result. Of course we also have

$$\texttt{sum(myarray)} = 78$$

Two related functions return the location of particular elements in an array. The returned value is a one-dimensional integer array containing the respective subscript of the element along each dimension. Note that when the argument object is a *one*-dimensional array, the returned object is an integer *array of length 1*, not simply an integer. (Fortran 90 distinguishes between these.)

[Iarr] `maxloc(array,`*`mask`*`)`
Location of the maximum value in an array.

[Iarr] `minloc(array,`*`mask`*`)`
Location of the minimum value in an array.

Similarly returning a one-dimensional integer array are

[Iarr] `shape(array)`
Returns the shape of `array` as a one-dimensional integer array.

[Iarr] `lbound(array,`*`dim`*`)`
When `dim` is absent, returns an array of lower bounds for each dimension of subscripts of `array`. When `dim` is present, returns the value only for dimension `dim`, as a scalar.

[Iarr] `ubound(array,`*`dim`*`)`
When `dim` is absent, returns an array of upper bounds for each dimension of subscripts of `array`. When `dim` is present, returns the value only for dimension `dim`, as a scalar.

Array Unary and Binary Functions

The most powerful array operations are simply built into the language as operators. All the usual arithmetic and logical operators (`+`, `-`, `*`, `/`, `**`, `.not.`, `.and.`, `.or.`, `.eqv.`, `.neqv.`) can be applied to arrays of arbitrary shape or (for the binary operators) between two arrays of the same shape, or between arrays and scalars. The types of the arrays must, of course, be appropriate to the operator used. The result in all cases is to perform the operation element by element on the arrays.

We also have the intrinsic functions,

[Num] `dot_product(veca,vecb)`
Scalar dot product of two one-dimensional vectors `veca` and `vecb`.

[Num] `matmul(mata,matb)`
Result of matrix-multiplying the two two-dimensional matrices `mata` and `matb`. The shapes have to be such as to allow matrix multiplication. Vectors (one-dimensional arrays) are additionally allowed as either the first or second argument, but not both; they are treated as row vectors in the first argument, and as column vectors in the second.

You might wonder how to form the *outer* product of two vectors, since `matmul` specifically excludes this case. (See §22.1 and §23.5 for answer.)

Array Manipulation Functions

These include many powerful features that a good Fortran 90 programmer should master.

[argTS] `cshift(array,shift,dim)`
: If `dim` is omitted, it is taken to be 1. Returns the result of circularly left-shifting every one-dimensional section of `array` (in dimension `dim`) by `shift` (which may be negative). That is, for positive `shift`, values are moved to smaller subscript positions. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where `shift` is an array.

[argTS] `merge(tsource,fsource,mask)`
: Returns same shape object as `tsource` and `fsource` containing the former's components where `mask` is true, the latter's where it is false.

[argTS] `eoshift(array,shift,boundary,dim)`
: If `dim` is omitted, it is taken to be 1. Returns the result of end-off left-shifting every one-dimensional section of `array` (in dimension `dim`) by `shift` (which may be negative). That is, for positive `shift`, values are moved to smaller subscript positions. If `boundary` is present as a scalar, it supplies elements to fill in the blanks; if it is not present, zero values are used. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where `boundary` and/or `shift` is an array.

[argT] `pack(array,mask,vector)`
: Returns a one-dimensional array containing the elements of `array` that pass the `mask`. Components of optional `vector` are used to pad out the result to the size of `vector` with specified values.

[argT] `reshape(source,shape,pad,order)`
: Takes the elements of `source`, in normal Fortran order, and returns them (as many as will fit) as an array whose shape is specified by the one-dimensional integer array `shape`. If there is space remaining, then `pad` must be specified, and is used (as many sequential copies as necessary) to fill out the rest. For description of `order`, consult a Fortran 90 reference, e.g., [M&R, 8.13.3].

[argT] `spread(source,dim,ncopies)`
: Returns an array whose rank is one greater than `source`, and whose `dim` dimension is of length `ncopies`. Each of the result's `ncopies` array sections having a fixed subscript in dimension `dim` is a copy of `source`. (That is, it spreads `source` into the `dim`th dimension.)

[argT] `transpose(matrix)`
: Returns the transpose of `matrix`, which must be two-dimensional.

[argT] `unpack(vector,mask,field)`
: Returns an array whose type is that of `vector`, but whose shape is that of `mask`. The components of `vector` are put, in order, into the positions where `mask` is true. Where `mask` is false, components of `field` (which may be a scalar or an array with the same shape as `mask`) are used instead.

Bitwise Functions

Most of the bitwise functions should be familiar to Fortran 77 programmers as longstanding standard extensions of that language. Note that the bit *positions* number from zero to one less than the value returned by the `bit_size` function. Also note that bit positions number *from right to left*. Except for `bit_size`, the following functions are all elemental.

[Int] `bit_size(i)`
Number of bits in the integer type of `i`.

[Lgcl] `btest(i,pos)`
True if bit position `pos` is 1, false otherwise.

[Int] `iand(i,j)`
Bitwise logical and.

[Int] `ibclr(i,pos)`
Returns `i` but with bit position `pos` set to zero.

[Int] `ibits(i,pos,len)`
Extracts `len` consecutive bits starting at position `pos` and puts them in the low bit positions of the returned value. (The high positions are zero.)

[Int] `ibset(i,pos)`
Returns `i` but with bit position `pos` set to 1.

[Int] `ieor(i,j)`
Bitwise exclusive or.

[Int] `ior(i,j)`
Bitwise logical or.

[Int] `ishft(i,shift)`
Bitwise left shift by `shift` (which may be negative) with zeros shifted in from the other end.

[Int] `ishftc(i,shift)`
Bitwise circularly left shift by `shift` (which may be negative).

[Int] `not(i)`
Bitwise logical complement.

Some Functions Relating to Numerical Representations

[Real] `epsilon(x)`
Smallest nonnegligible quantity relative to 1 in the numerical model of `x`.

[Num] `huge(x)`
Largest representable number in the numerical model of `x`.

[Int] `kind(x)`

Returns the kind value for the numerical model of `x`.

[Real] `nearest(x,s)`
Real number nearest to `x` in the direction specified by the sign of `s`.

[Real] `tiny(x)`
Smallest positive number in the numerical model of `x`.

Other Intrinsic Procedures

[Lgcl] `present(a)`
True, within a subprogram, if an optional argument is actually present, otherwise false.

[Lgcl] `associated(pointer,target)`
True if `pointer` is associated with `target` or (if `target` is absent) with any target, otherwise false.

[Lgcl] `allocated(array)`
True if the allocatable `array` is allocated, otherwise false.

There are some pitfalls in using `associated` and `allocated`, having to do with arrays and pointers that can find themselves in *undefined* status [see §21.5, and also M&R, §3.3 and §6.5.1]. For example, pointers are always "born" in an undefined status, where the `associated` function returns unpredictable values.

For completeness, here is a list of Fortran 90's intrinsic procedures not already mentioned:

Other Numerical Representation Functions: `digits`, `exponent`, `fraction`, `rrspacing`, `scale`, `set_exponent`, `spacing`, `maxexponent`, `minexponent`, `precision`, `radix`, `range`, `selected_int_kind`, `selected_real_kind`.

Lexical comparison: `lge`, `lgt`, `lle`, `llt`.

Character functions: `ichar`, `char`, `achar`, `iachar`, `index`, `adjustl`, `adjustr`, `len_trim`, `repeat`, `scan`, `trim`, `verify`.

Other: `mvbits`, `transfer`, `date_and_time`, `system_clock`, `random_seed`, `random_number`. (We will discuss random numbers in some detail in Chapter B7.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.5 Advanced Fortran 90 Topics

Pointers, Arrays, and Memory Management

One of the biggest improvements in Fortran 90 over Fortran 77 is in the handling of arrays, which are the cornerstone of many numerical algorithms. In this subsection we will take a closer look at how to use some of these new array features effectively. We will look at how to code certain commonly occurring elements of program design, and we will pay particular attention to avoiding "memory leaks," where — usually inadvertently — we keep cumulatively allocating new storage for an array, every time some piece of code is invoked.

Let's first review some of the rules for using allocatable arrays and pointers to arrays. Recall that a pointer is born with an undefined status. Its status changes to "associated" when you make it refer to a target, and to "disassociated" when you `nullify` the pointer. [M&R, §3.3] You can also use `nullify` on a newly born pointer to change its status from undefined to disassociated; this allows you to test the status with the `associated` inquiry function. [M&R, §6.5.4] (While many compilers will not produce a run-time error if you test an undefined pointer with `associated`, you can't rely on this *laissez-faire* in your programming.)

The initial status of an allocatable array is "not currently allocated." Its status changes to "allocated" when you give it storage with `allocate`, and back to "not currently allocated" when you use `deallocate`. [M&R, §6.5.1] You can test the status with the `allocated` inquiry function. Note that while you can also give a pointer fresh storage with `allocate`, you can't test this with `allocated` — only `associated` is allowed with pointers. Note also that nullifying an allocated pointer leaves its associated storage in limbo. You must instead `deallocate`, which gives the pointer a testable "disassociated" status.

While allocating an array that is already allocated gives an error, you are allowed to allocate a pointer that already has a target. This breaks the old association, and could leave the old target inaccessible if there is no other pointer associated with it. [M&R, §6.5.2] Deallocating an array or pointer that has not been allocated is always an error.

Allocated arrays that are local to a subprogram acquire the "undefined" status on exit from the subprogram unless they have the `SAVE` attribute. (Again, not all compilers enforce this, but be warned!) Such undefined arrays cannot be referenced in any way, so you should explicitly deallocate all allocated arrays that are not saved before returning from a subprogram. [M&R, §6.5.1] The same rule applies to arrays declared in modules that are currently accessed only by the subprogram. While you can reference undefined pointers (e.g., by first nullifying them), it is good programming practice to deallocate explicitly any allocated pointers declared locally before leaving a subprogram or module.

Now let's turn to using these features in programs. The simplest example is when we want to implement global storage of an array that needs to be accessed by two or more different routines, and we want the size of the array to be determined at run time. As mentioned earlier, we implement global storage with a `MODULE` rather than a `COMMON` block. (We ignore here the additional possibility of passing

global variables by having one routine CONTAINed within the other.) There are two good ways of handling the dynamical allocation in a MODULE. Method 1 uses an allocatable array:

```
MODULE a
REAL(SP), DIMENSION(:), ALLOCATABLE :: x
END MODULE a

SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
...
allocate(x(size(y)))
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the global variable x gets assigned storage in subroutine b (in this case, the same as the length of y). The length of y is of course defined in the procedure that calls b. The array x is made available to any other subroutine called by b by including a USE a statement. The status of x can be checked with an allocated inquiry function on entry into either b or the other subroutine if necessary. As discussed above, you must be sure to deallocate x before returning from subroutine b. If you want x to retain its values between calls to b, you add the SAVE attribute to its declaration in a, and *don't* deallocate it on returning from b. (Alternatively, you could put a USE a in your main program, but we consider that bug-prone, since forgetting to do so can create all manner of difficult-to-diagnose havoc.) To avoid allocating x more than once, you test it on entry into b:

```
if (.not. allocated(x)) allocate(x(size(y)))
```

The second way to implement this type of global storage (Method 2) uses a pointer:

```
MODULE a
REAL(SP), DIMENSION(:), POINTER :: x
END MODULE a

SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
REAL(SP), DIMENSION(size(y)), TARGET :: xx
...
x=>xx
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the *automatic array* xx gets its temporary storage automatically on entry into b, and automatically gets deallocated on exit from b. [M&R, §6.4] The global pointer x can access this storage in any routine with a USE a that is called by b. You can check that things are in order in such a called routine by testing x with associated. If you are going to use x for some other purpose as well, you should nullify it on leaving b so that it doesn't have undefined status. Note that this implementation does not allow values to be saved between calls: You can't SAVE automatic arrays — that's not what they're for. You would have to SAVE x in the module, and allocate it in the subroutine instead of pointing it to a suitable automatic array. But this is essentially Method 1 with the added complication of using a pointer, so Method 1 is simpler when you want to save values. When you don't

need to save values between calls, we lean towards Method 2 over Method 1 because we like the automatic allocation and deallocation, but either method works fine.

An example of Method 1 (allocatable array) is in routine `rkdumb` on page 1297. An example of Method 1 with `SAVE` is in routine `pwtset` on p. 1265. Method 2 (pointer) shows up in routines `newt` (p. 1196), `broydn` (p. 1199), and `fitexy` (p. 1286). A variation is shown in routines `linmin` (p. 1211) and `dlinmin` (p. 1212): When the array that needs to be shared is an argument of one of the routines, Method 2 is better.

An extension of these ideas occurs if we allocate some storage for an array initially, but then might need to increase the size of the array later without losing the already-stored values. The function `reallocate` in our utility module `nrutil` will handle this for you, but it expects a pointer argument as in Method 2. Since no automatic arrays are used, you are free to `SAVE` the pointer if necessary. Here is a simple example of how to use `reallocate` to create a workspace array that is local to a subroutine:

```
SUBROUTINE a
USE nrutil, ONLY : reallocate
REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
   init=.false.
   nullify(wksp)
   wksp=>reallocate(wksp,100)
end if
...
if (nterm > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
...
END SUBROUTINE a
```

Here the workspace is initially allocated a size of 100. If the number of elements used (`nterm`) ever exceeds the size of the workspace, the workspace is doubled. (In a realistic example, one would of course check that the doubled size is in fact big enough.) Fortran 90 experts can note that the `SAVE` on `init` is not strictly necessary: Any local variable that is initialized is automatically saved. [M&R, §7.5]

You can find similar examples of `reallocate` (with some further discussion) in `eulsum` (p. 1070), `hufenc` (p. 1348), and `arcode` (p. 1350). Examples of reallocate used with global variables in modules are in `odeint` (p. 1300) and `ran_state` (p. 1144).

Another situation where we have to use pointers and not allocatable arrays is when the storage is required for components of a derived type, which are not allowed to have the allocatable attribute. Examples are in `hufmak` (p. 1346) and `arcmak` (p. 1349).

Turning away from issues relating to global variables, we now consider several other important programming situations that are nicely handled with pointers. The first case is when we want a subroutine to return an array whose size is not known in advance. Since dummy arguments are not allocatable, we must use a pointer. Here is the basic construction:

```
SUBROUTINE a(x,nx)
REAL(SP), DIMENSION(:), POINTER :: x
INTEGER(I4B), INTENT(OUT) :: nx
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
```

```
   init=.false.
   nullify(x)
else
   if (associated(x)) deallocate(x)
end if
...
nx=...
allocate(x(nx))
x(1:nx)=...
END SUBROUTINE a
```

Since the length of x can be found from `size(x)`, it is not absolutely necessary to pass `nx` as an argument. Note the use of the initial logic to avoid memory leaks. If a higher-level subroutine wants to recover the memory associated with `x` from the last call to `SUBROUTINE a`, it can do so by first deallocating it, and then nullifying the pointer. Examples of this structure are in `zbrak` (p. 1184), `period` (p. 1258), and `fasper` (p. 1259). A related situation is where we want a function to return an array whose size is not predetermined, such as in `voltra` on (p. 1326). The discussion of `voltra` also explains the potential pitfalls of functions returning pointers to dynamically allocated arrays.

A final useful pointer construction enables us to set up a data structure that is essentially an array of arrays, independently allocatable on each part. We are not allowed to declare an array of pointers in Fortran 90, but we can do this indirectly by defining a derived type that consists of a pointer to the appropriate kind of array. [M&R, §6.11] We can then define a variable that is an allocatable array of the new type. For example,

```
TYPE ptr_to_arr
   REAL(SP), DIMENSION(:), POINTER :: arr
END TYPE
TYPE(ptr_to_arr), DIMENSION(:), ALLOCATABLE :: x
...
allocate(x(n))
...
do i=1,n
   allocate(x(i)%arr(m))
end do
```

sets up a set `x` of `n` arrays of length `m`. See also the example in `mglin` (p. 1334).

There is a potential problem with dynamical memory allocation that we should mention. The Fortran 90 standard does not require that the compiler perform "garbage collection," that is, it is not required to recover deallocated memory into nice contiguous pieces for reuse. If you enter and exit a subroutine many times, and each time a large chunk of memory gets allocated and deallocated, you could run out of memory with a "dumb" compiler. You can often alleviate the problem by deallocating variables in the reverse order that you allocated them. This tends to keep a large contiguous piece of memory free at the top of the heap.

Scope, Visibility, and Data Hiding

An important principle of good programming practice is *modularization*, the idea that different parts of a program should be insulated from each other as much as possible. An important subcase of modularization is *data hiding*, the principle that actions carried out on variables in one part of the code should not be able to

affect the values of variables in other parts of the code. When it is necessary for one "island" of code to communicate with another, the communication should be through a well-defined interface that makes it obvious exactly what communication is taking place, and prevents any other interchange from occurring. Otherwise, different sections of code should not have access to variables that they don't need.

The concept of data hiding extends not only to variables, but also to the names of procedures that manipulate the variables: A program for screen graphics might give the user access to a routine for drawing a circle, but it might "hide" the names (and methods of operation) of the primitive routines used for calculating the coordinates of the points on the circumference. Besides producing code that is easier to understand and to modify, data hiding prevents unintended side effects from producing hard-to-find errors.

In Fortran, the principal language construction that effects data hiding is the use of subroutines. If all subprograms were restricted to have no more than ten executable statements per routine, and to communicate between routines only by an explicit list of arguments, the number of programming errors might be greatly reduced! Unfortunately few tasks can be easily coded in this style. For this and other reasons, we think that too much procedurization is a bad thing; one wants to find the *right* amount. Fortunately Fortran 90 provides several additional tools to help with data hiding.

Global variables and routine names are important, but potentially dangerous, things. In Fortran 90, global variables are typically encapsulated in modules. Access is granted only to routines with an appropriate `USE` statement, and can be restricted to specific identifiers by the `ONLY` option. [M&R, §7.10] In addition, variable and routine names within the module can be designated as `PUBLIC` or `PRIVATE` (see, e.g., `quad3d` on p. 1065). [M&R, §7.6]

The other way global variables get communicated is by having one routine `CONTAINE`d within another. [M&R, §5.6] This usage is potentially lethal, however, because *all* the outer routine's variables are visible to the inner routine. You can try to control the problem somewhat by passing some variables back and forth as arguments of the inner routine, but that still doesn't prevent inadvertent side effects. (The most common, and most stupid, is inadvertent reuse of variables named `i` or `j` in the `CONTAINE`d routine.) Also, a long list of arguments reduces the convenience of using an internal routine in the first place. We advise that internal subprograms be used with caution, and only to carry out simple tasks.

There are some good ways to use `CONTAINS`, however. Several of our recipes have the following structure: A principal routine is invoked with several arguments. It calls a subsidiary routine, which needs to know some of the principal routine's arguments, some global variables, and some values communicated directly as arguments to the subsidiary routine. In Fortran 77, we have usually coded this by passing the global variables in a `COMMON` block and all other variables as arguments to the subsidiary routine. If necessary, we copied the arguments of the primary routine before passing them to the subsidiary routine. In Fortran 90, there is a more elegant way of accomplishing this, as follows:

```
SUBROUTINE recipe(arg)
REAL(SP) :: arg
REAL(SP) :: global_var
call recipe_private
CONTAINS
```

```
SUBROUTINE recipe_private
...
call subsidiary(local_arg)
...
END SUBROUTINE recipe_private

SUBROUTINE subsidiary(local_arg)
...
END SUBROUTINE subsidiary
END SUBROUTINE recipe
```

Notice that the principal routine (`recipe`) has practically nothing in it — only declarations of variables intended to be visible to the subsidiary routine (`subsidiary`). All the real work of `recipe` is done in `recipe_private`. This latter routine has visibility on all of `recipe`'s variables, while any additional variables that `recipe_private` defines are *not* visible to `subsidiary` — which is the whole purpose of this way of organizing things. Obviously `arg` and `global_var` can be much more general data types than the example shown here, including function names. For examples of this construction, see `amoeba` (p. 1208), `amebsa` (p. 1222), `mrqmin` (p. 1292), and `medfit` (p. 1294).

Recursion

A subprogram is recursive if it calls itself. While forbidden in Fortran 77, recursion is allowed in Fortran 90. [M&R, §5.16–§5.17] You must supply the keyword `RECURSIVE` in front of the `FUNCTION` or `SUBROUTINE` keyword. In addition, if a `FUNCTION` calls itself directly, as opposed to calling another subprogram that in turn calls it, you must supply a variable to hold the result with the `RESULT` keyword. Typical syntax for this case is:

```
RECURSIVE FUNCTION f(x) RESULT(g)
REAL(SP) :: x,g
if ...
   g=...
else
   g=f(...)
end if
END FUNCTION f
```

When a function calls itself directly, as in this example, there always has to be a "base case" that does not call the function; otherwise the recursion never terminates. We have indicated this schematically with the `if...else...end if` structure.

On serial machines we tend to avoid recursive implementations because of the additional overhead they incur at execution time. Occasionally there are algorithms for which the recursion overhead is relatively small, and the recursive implementation is simpler than an iterative version. Examples in this book are `quad_3d` (p. 1065), `miser` (p. 1164), and `mglin` (p. 1334). Recursion is much more important when parallelization is the goal. We will encounter in Chapter 22 numerous examples of algorithms that can be parallelized with recursion.

SAVE Usage Style

A quirk of Fortran 90 is that any variable with initial values acquires the `SAVE` attribute automatically. [M&R, §7.5 and §7.9] As a help to understanding

an algorithm, we have elected to put an explicit SAVE on all variables that really do need to retain their values between calls to a routine. We do this even if it is redundant because the variables are initialized. Note that we generally prefer to assign initial values with initialization expressions rather than with DATA statements. We reserve DATA statements for cases where it is convenient to use the repeat count feature to set multiple occurrences of a value, or when binary, octal, or hexadecimal constants are used. [M&R, §2.6.1]

Named Control Structures

Fortran 90 allows control structures such as do loops and if blocks to be named. [M&R, §4.3–§4.5] Typical syntax is

```
name:do i=1,n
   ...
end do name
```

One use of naming control structures is to improve readability of the code, especially when there are many levels of nested loops and if blocks. A more important use is to allow exit and cycle statements, which normally refer to the innermost do loop in which they are contained, to transfer execution to the end of some outer loop. This is effected by adding the name of the outer loop to the statement: exit name or cycle name.

There is great potential for misuse with named control structures, since they share some features of the much-maligned goto. We recommend that you use them sparingly. For a good example of their use, contrast the Fortran 77 version of simplx with the Fortran 90 version on p. 1216.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.6 And Coming Soon: Fortran 95

One of the more positive effects of Fortran 90's long gestation period has been the general recognition, both by the X3J3 committee and by the community at large, that Fortran needs to evolve over time. Indeed, as we write, the process of bringing forth a minor, but by no means insignificant, updating of Fortran 90 — named Fortran 95 — is well under way.

Fortran 95 will differ from Fortran 90 in about a dozen features, only a handful of which are of any importance to this book. Generally these are extensions that will make programming, especially parallel programming, easier. In this section we give a summary of the anticipated language changes. In §22.1 and §22.5 we will comment further on the implications of Fortran 95 to some parallel programming tasks; in §23.7 we comment on what differences Fortran 95 will make to our nrutil utility functions.

No programs in Chapters B1 through B20 of this book edition use any Fortran 95 extensions.

FORALL Statements and Blocks

Fortran 95 introduces a new `forall` control structure, somewhat akin to the `where` construct, but allowing for greater flexibility. It is something like a do-loop, but with the proviso that the indices looped over are allowed to be done in any order (ideally, in parallel). The `forall` construction comes in both single-statement and block variants. Instead of using the do-loop's comma-separated triplets of lower-value, upper-value, and increment, it borrows its syntax from the colon-separated form of array sections. Some examples will give you the idea.

Here is a simple example that could alternatively be done with Fortran 90's array sections and `transpose` intrinsic:

```
forall (i=1:20, j=1:10:2) x(i,j)=y(j,i)
```

The block form allows more than one executable statement:

```
forall (i=1:20, j=1:10:2)
   x(i,j)=y(j,i)
   z(i,j)=y(i,j)**2
end forall
```

Here is an example that cannot be done with Fortran 90 array sections:

```
forall (i=1:20, j=1:20) a(i,j)=3*i+j**2
```

`forall` statements can also take optional masks that restrict their action to a subset of the loop index combinations:

```
forall (i=1:100, j=1:100, (i>=j .and. x(i,j)/=0.0) ) x(i,j)=1.0/x(i,j)
```

`forall` constructions can be nested, or nested inside `where` blocks, or have `where` constructions inside them. An additional new feature in Fortran 95 is that `where` blocks can themselves be nested.

PURE Procedures

Because the inside iteration of a `forall` block can be done in any order, or in parallel, there is a logical difficulty in allowing functions or subroutines inside such blocks: If the function or subroutine has *side effects* (that is, if it changes any data elsewhere in the machine, or in its own saved variables) then the result of a `forall` calculation could depend on the order in which the iterations happen to be done. This can't be tolerated, of course; hence a new `PURE` attribute for subprograms.

While the exact stipulations are somewhat technical, the basic idea is that if you declare a function or subroutine as `PURE`, with a syntax like,

```
PURE FUNCTION myfunc(x,y,z)
```

or

```
PURE SUBROUTINE mysub(x,y,z)
```

then you are guaranteeing to the compiler (and it will enforce) that the only values changed by `mysub` or `myfunc` are returned function values, subroutine arguments with the `INTENT(OUT)` attribute, and automatic (scratch) variables within the procedure.

You can then use your pure procedures within `forall` constructions. Pure functions are also allowed in some specification statements.

ELEMENTAL Procedures

Fortran 95 removes Fortran 90's nagging restriction that only intrinsic functions are elemental. The way this works is that you write a pure procedure that operates on scalar values, but include the attribute `ELEMENTAL` (which automatically implies `PURE`). Then, as long as the function has an explicit interface in the referencing program, you can call it with any shape of argument, and it will act elementally. Here's an example:

```
ELEMENTAL FUNCTION myfunc(x,y,z)
REAL :: x,y,z,myfunc
...
myfunc = ...
END
```

In a program with an explicit interface for `myfunc` you could now have

```
REAL, DIMENSION(10,20) :: x,y,z,w
...
w=myfunc(x,y,z)
```

Pointer and Allocatable Improvements

Fortran 95, unlike Fortran 90, requires that any allocatable variables (except those with `SAVE` attributes) that are allocated within a subprogram be automatically deallocated by the compiler when the subprogram is exited. This will remove Fortran 90's "undefined allocation status" bugaboo.

Fortran 95 also provides a method for pointer variables to be born with disassociated association status, instead of the default (and often inconvenient) "undefined" status. The syntax is to add an initializing `=> NULL()` to the declaration, as:

```
REAL, DIMENSION(:,:), POINTER :: mypoint => NULL()
```

This does not, however, eliminate the possibility of undefined association status, because you have to remember to use the null initializer if want your pointer to be disassociated.

Some Other Fortran 95 Features

In Fortran 95, `maxloc` and `minloc` have the additional optional argument `DIM`, which causes them to act on all one-dimensional sections that span through the named dimension. This provides a means for getting the locations of the values returned by the corresponding functions `maxval` and `minval` in the case that their `DIM` argument is present.

The `sign` intrinsic can now distinguish a negative from a positive real zero value: `sign(2.0,-0.0)` is -2.0.

There is a new intrinsic subroutine `cpu_time(time)` that returns as a real value `time` a process's elapsed CPU time.

There are some minor changes in the namelist facility, in defining minimum field widths for the I, B, O, Z, and F edit descriptors, and in resolving minor conflicts with some other standards.