Chapter 40
Converting from Fortran 77

Twas brillig, and the slithy toves did gyre and gimble in the wabe; All mimsy were the borogoves, And the mome raths outgrabe.

Lewis Carroll

Aim
This chapter looks at some of the options available when working with older Fortran code.

40.1 Introduction

This chapter looks at converting Fortran 77 code to a modern Fortran style.

The aim is to provide the Fortran 77 programmer (and in particular the person with legacy code) with some simple guidelines for conversion.

The first thing that one must have is a thorough understanding of the newer, better language features of Fortran. It is essential that the material in the earlier chapters of this book are covered, and some of the problems attempted. This will provide a feel for modern Fortran.

The second thing one must have is a thorough understanding of the language constructs used in your legacy code. Use should be made of the compiler documentation for whatever Fortran 77 compiler you are using, as this will provide the detailed (often system specific) information required. The recommendations below are therefore brief.

It is possible to move gradually from Fortran 77 to modern Fortran. In many cases existing code can be quite simply recompiled by a suitable choice of compiler options. This enables us to mix and match old and new in one program. This process is likely to highlight nonstandard language features in your old code. There will inevitably be some problems here.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_40

The standard identifies two kinds of decremented features; deleted and obsolescent. In the long-term these features are candidates for removal from future standards. These deleted and obsolescent features may well be supported by compilers even though they have been removed from the standard.

The following information is taken from the Fortran 2018 standard.

40.2 Deleted Features from Fortran 90

These deleted features are those features of Fortran 90 that were redundant and considered largely unused. The following Fortran 90 features are not required.

- (1) Real and double precision DO variables.
  In Fortran 77 and Fortran 90, a DO variable was allowed to be of type real or double precision in addition to type integer; this has been deleted. A similar result can be achieved by using a DO construct with no loop control and the appropriate exit test.
- (2) Branching to an END IF statement from outside its block.
  In Fortran 77 and Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted. A similar result can be achieved by branching to a CONTINUE statement that is immediately after the END IF statement.
- (3) PAUSE statement.
  The PAUSE statement, provided in Fortran 66, Fortran 77, and Fortran 90, has been deleted. A similar result can be achieved by writing a message to the appropriate unit, followed by reading from the appropriate unit.
- (4) ASSIGN and assigned GO TO statements, and assigned format specifiers.
  The ASSIGN statement and the related assigned GO TO statement, provided in Fortran 66, Fortran 77, and Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, provided in Fortran 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GO TO statement and by using a default character variable to hold a format specification instead of using an assigned integer.
- (5) H edit descriptor.
  In Fortran 77 and Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in Fortran 66; this has been deleted. A similar result can be achieved by using a character string edit descriptor.
- (6) Vertical format control.
  In Fortran 66, Fortran 77, Fortran 90, and Fortran 95 formatted output to certain units resulted in the first character of each record being interpreted as controlling vertical spacing. There was no standard way to detect whether output to a unit resulted in this vertical format control, and no way to specify that it should be applied; this has been deleted. The effect can be achieved by post-processing a

formatted file. See ISO/IEC 1539:1991 for detailed rules of how these deleted features worked.

40.3 Deleted Features from Fortran 2008

These deleted features are those features of Fortran 2008 that were redundant and considered largely unused. The following Fortran 2008 features are not required.

- (1) Arithmetic IF statement.
  The arithmetic IF statement is incompatible with ISO/IEC/IEEE 60559:2011 and necessarily involves the use of statement labels; statement labels can hinder optimization, and make code hard to read and maintain. Similar logic can be more clearly encoded using other conditional statements.
- (2) Nonblock DO construct
  The nonblock forms of the DO loop were confusing and hard to maintain. Shared termination and dual use of labeled action statements as do termination and branch targets were especially error11 prone.

40.4 Obsolescent Features

The obsolescent features are those features of Fortran 90 that were redundant and for which better methods were available in Fortran 90. Subclause 4.4.3 describes the nature of the obsolescent features. The obsolescent features in this document are the following.

- (1) Alternate return
- (2) Computed GO TO
- (3) Statement functions
- (4) DATA statements amongst executable statements
- (5) Assumed length character functions
- (6) Fixed form source
- (7) CHARACTER* form of CHARACTER declaration
- (8) ENTRY statements
- (9) Label form of DO statement
- (10) COMMON and EQUIVALENCE statements, and the block data program unit
- (11) Specific names for intrinsic functions
- (12) FORALL construct and statement

40.4.1 Alternate Return

An alternate return introduces labels into an argument list to allow the called procedure to direct the execution of the caller upon return. The same effect can be achieved with a return code that is used in a SELECT CASE construct on return. This avoids an irregularity in the syntax and semantics of argument association. For example,

```
CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

can be replaced by

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
CASE (1)
...
CASE (2)
...
CASE (3)
...
CASE DEFAULT
...
END SELECT
```

40.4.2 Computed GO TO Statement

The computed GO TO statement has been superseded by the SELECT CASE construct, which is a generalized, easier to use, and clearer means of expressing the same computation.

40.4.3 Statement Functions

Statement functions are subject to a number of nonintuitive restrictions and are a potential source of error because their syntax is easily confused with that of an assignment statement. The internal function is a more generalized form of the statement function and completely supersedes it.

40.4.4 DATA Statements Among Executables

The statement ordering rules allow DATA statements to appear anywhere in a program unit after the specification statements. The ability to position DATA statements amongst executable statements is very rarely used, unnecessary, and a potential source of error.

40.4.5 Assumed Character Length Functions

Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran the philosophy is that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association. Some uses of this facility can be replaced with an automatic character length function, where the length of the function result is declared in a specification expression. Other uses can be replaced by the use of a subroutine whose arguments correspond to the function result and the function arguments. Note that dummy arguments of a function can have assumed character length.

40.4.6 Fixed Form Source

Fixed form source was designed when the principal machine-readable input medium for new programs was punched cards. Now that new and amended programs are generally entered via keyboards with screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology. It is a simple matter for a software tool to convert from fixed to free form source.

40.4.7 CHARACTER* Form of CHARACTER Declaration

In addition to the CHARACTER*char-length form introduced in Fortran 77, Fortran 90 provided the CHAR3 ACTER([LEN =] type-param-value) form. The older form (CHARACTER*char-length) is redundant.

40.4.8 ENTRY Statements

ENTRY statements allow more than one entry point to a subprogram, facilitating sharing of data items and executable statements local to that subprogram. This can be replaced by a module containing the (private) data items, with a module procedure for each entry point and the shared code in a private module procedure.

40.4.9 Label DO Statement

The label in the DO statement is redundant with the construct name. Furthermore, the label allows unrestricted branches and, for its main purpose (the target of a conditional branch to skip the rest of the current iteration), is redundant with the CYCLE statement, which is clearer.

40.4.10 COMMON and EQUIVALENCE Statements and the Block Data Program Unit

Common blocks are error-prone and have largely been superseded by modules. EQUIVALENCE similarly is error-prone. Whilst use of these statements was invaluable prior to Fortran 90 they are now redundant and can inhibit performance. The block data program unit exists only to serve common blocks and hence is also redundant.

40.4.11 Specific Names for Intrinsic Functions

The specific names of the intrinsic functions are often obscure and hinder portability. They have been redundant since Fortran 90. Use generic names for references to intrinsic procedures.

40.4.12 FORALL Construct and Statement

The FORALL construct and statement were added to the language in the expectation that they would enable highly efficient execution, especially on parallel processors. However, experience indicates that they are too complex and have too many restrictions for compilers to take advantage of them. They are redundant with the DO CONCURRENT construct, and many of the manipulations for which they might be used can be done more effectively using pointers, especially using pointer rank remapping.

40.5 Better Alternatives

Below we are looking at the new features of the Fortran standard, and how we can replace our current coding practices with the better facilities that now exist.

- double precision — use the module `precision_module` which was introduced in Chap. 21 and used subsequently throughout the book.
- fixed format — use free format
- implicit typing — use implicit none
- block data — use modules
- common statement — use modules
- equivalence — Invariably the use of this feature requires considerable system specific knowledge. There will be cases where there have been extremely good reasons why this feature has been used, normally efficiency related. However with the rapid changes taking place in the power and speed of hardware these reasons are diminishing.
- assumed-size/explicit-shape dummy array arguments — if a dummy argument is assumed-size or explicit-shape (the only ones available in Fortran 77) then the ranks of the actual argument and the associated argument don't have to be the same. With modern Fortran arrays are now objects instead of a linear sequence of elements, as was the case with Fortran 77, and now for array arguments the fundamental rule is that actual and dummy arguments have the same rank and same extents in each dimension, i.e., the same shape, and this is done using assumed-shape dummy array arguments. An explicit interface is mandatory for assumed-shape arrays.
- entry statement — use module plus use statement.
- statement functions — use internal function, see Chap. 12, and examples later this chapter.
- computed goto — use case statement, see Chap. 13.
- alternate return — use error flags on calling routine.
- external statement for dummy procedure arguments - use modules and interface blocks. See the Runge-Kutta-Merson example in Chap. 26.

Use explicit interfaces everywhere, i.e. use module procedures.
This also provides argument checking and other benefits.

40.6 Free and Commercial Conversion Tools

At the time of writing there are several options. Have a look at our Fortran resource file:

```
https://www.fortranplus.co.uk/
```

for up to date information.

Here are brief details of the tools currently available.

40.6.1 Convert

Fortran 77 to Fortran 90 converter by Mike Metcalf.

```
http://rhymneyconsulting.co.uk/fortran/convert.f90
```

Here are some of the comments from the program.

```
!    A program to convert FORTRAN 77 source form to Fortran 90 source  *
!  form. It also formats the code by indenting the bodies of DO-loops  *
!  and IF-blocks by ISHIFT columns. Statement keywords are             *
!  followed if necessary by a blank, and blanks within tokens are      *
!  are suppressed; this handling of blanks is optional.                *
!    If a CONTINUE statement terminates a single DO loop, it is        *
!  replaced by END DO.                                                 *
!    Procedure END statements have the procedure name added, if        *
!  blanks are handled.                                                 *
!    Statements like INTEGER*2 are converted to INTEGER(2), if blanks  *
!  are handled. Depending on the target processor, a further global    *
!  edit might be required (e.g. where 2 bytes correspond to KIND=1).   *
!  Typed functions and assumed-length character specifications are     *
!  treated similarly. The length specification *4 is removed for all   *
!  data types except CHARACTER, as is *8 for COMPLEX. This             *
!  treatment of non-standard type declarations includes any            *
!  non-standard IMPLICIT statements.                                   *
!    Optionally, interface blocks only may be produced; this requires  *
!  blanks processing to be requested. The interface blocks are         *
!  compatible with both the old and new source forms.                  *
```

40.6.2 Forcheck

A Fortran analyser and programming aid.

```
http://www.forcheck.nl/
```

40.6.3 Nag Compiler Polish Tool

Here is the home page for the Nag compiler.

```
https://www.nag.co.uk/nag-compiler
```

Here is a brief description of the tools.

In addition the Compiler provides software tools to: convert fixed-format code to free-format; pretty print (“polish”) code; list dependency information of modules and include files; produce callgraphs; and generate explicit procedure interfaces as module or INCLUDE files.

40.6.4 Plusfort

Fortran 77 to Fortran 90 converter.

```
https://www.polyhedron.com/
```

40.7 Example 1: Using the plusFORT Tool Suite from Polyhedron Software

Below is an example from their site that looks at the same subroutine in Fortran 66, 77 and 90 styles.

40.7.1 Original Fortran 66

This subroutine picks off digits from an integer and branches depending on their value.

```
      SUBROUTINE OBACT(TODO)
      INTEGER TODO,DONE,IP,BASE
      COMMON /EG1/N,L,DONE
      PARAMETER (BASE=10)
   13 if(TODO.EQ.0) GO TO 12
      I=MOD(TODO,BASE)
      TODO=TODO/BASE
      GO TO(62,42,43,62,404,45,62,62,62),I
      GO TO 13
   42 CALL COPY
      GO TO 127
   43 CALL MOVE
      GO TO 144
  404 N=-N
   44 CALL DELETE
      GO TO 127
   45 CALL print
      GO TO 144
   62 CALL BADACT(i)
      GO TO 12
  127 L=L+N
  144 DONE=DONE+1
      CALL RESYNC
      GO TO 13
   12 RETURN
      END
```

40.7.2 Fortran 77 Version

In addition to restructuring, SPAG has renamed some variables, removed the unused variable IP, inserted declarations, and used upper and lower case to distinguish different types of variable:

```
      SUBROUTINE OBACT(TODO)
      IMPLICIT NONE
C*** START OF DECLARATIONS INSERTED BY SPAG
      INTEGER ACT , LENGTH , NCHAR
C*** END OF DECLARATIONS INSERTED BY SPAG
      INTEGER TODO , DONE , BASE
      COMMON /EG1   / NCHAR , LENGTH , DONE
      PARAMETER (BASE=10)
 100  IF ( TODO.NE.0 ) THEN
         ACT = MOD(TODO,BASE)
         TODO = TODO/BASE
         IF ( ACT.EQ.1 .OR. ACT.EQ.4 .OR.
     &        ACT.EQ.7 .OR. ACT.EQ.8 .OR.
     &        ACT.EQ.9 ) THEN
            CALL BADACT(ACT)
            GOTO 200
         ELSEIF ( ACT.EQ.2 ) THEN
            CALL COPY
            LENGTH = LENGTH + NCHAR
         ELSEIF ( ACT.EQ.3 ) THEN
            CALL MOVE
         ELSEIF ( ACT.EQ.5 ) THEN
            NCHAR = -NCHAR
            CALL DELETE
            LENGTH = LENGTH + NCHAR
         ELSEIF ( ACT.EQ.6 ) THEN
            CALL PRINT
         ELSE
            GOTO 100
         ENDIF
         DONE = DONE + 1
         CALL RESYNC
         GOTO 100
      ENDIF
 200  RETURN
      END
```

40.7.3 Fortran 90 Version

SPAG has used `do while`, `select case`, `exit` and `cycle`. No GOTOs or labels remain.

```
      subroutine obact(todo)
      implicit none
!*** start of declarations inserted by spag
      integer act , length , nchar
!*** end of declarations inserted by spag
```

```
      integer todo , done , base
      common /eg1   / nchar , length , done
      parameter (base=10)
      do while ( todo.ne.0 )
         act = mod(todo,base)
         todo = todo/base
         select case (act)
         case (1,4,7,8,9)
            call badact(act)
            exit
         case (2)
            call copy
            length = length + nchar
         case (3)
            call move
         case (5)
            nchar = -nchar
            call delete
            length = length + nchar
         case (6)
            call print
         case default
            cycle
         end select
         done = done + 1
         call resync
      enddo
      return
      end
```

This tool suite can also be used in the maintenance of code during development.

40.8 Example 2: Leaving as Fortran 77

The simplest option if the function or subroutine works and does not need updating is to leave it as Fortran 66 or 77 fixed source form. The Netlib routines in Chap. 36 are a good example of this. They are

```
dsort.f
ssort.f
isort.f
```

We had to make some changes to get them to compile, and the changes are documented in the earlier chapter.

40.9 Example 3: Simple Conversion to Fortran 90

The Metcalf convert program can be used to simply convert from Fortran 77 to Fortran 90.

Using this utility on the Netlib dsort.f Fortran 77 code will produce a Fortran 90 equivalent. Here is the converted code.

```
      SUBROUTINE DSORT (DX, DY, N, KFLAG)
!***BEGIN PROLOGUE  DSORT
!***PURPOSE  Sort an array and optionally make the same interchanges in
!            an auxiliary array.  The array may be sorted in increasing
!            or decreasing order. A slightly modified QUICKSORT
!            algorithm is used.
!***LIBRARY   SLATEC
!***CATEGORY  N6A2B
!***TYPE      DOUBLE PRECISION (SSORT-S, DSORT-D, ISORT-I)
!***KEYWORDS  SINGLETON QUICKSORT, SORT, SORTING
!***AUTHOR  Jones, R. E., (SNLA)
!           Wisniewski, J. A., (SNLA)
!***DESCRIPTION
!
!   DSORT sorts array DX and optionally makes the same interchanges in
!   array DY.  The array DX may be sorted in increasing order or
!   decreasing order. A slightly modified quicksort algorithm is used.
!
!   Description of Parameters
!      DX - array of values to be sorted   (usually abscissas)
!      DY - array to be (optionally) carried along
!      N  - number of values in array DX to be sorted
!      KFLAG - control parameter
!            =  2  means sort DX in increasing order and carry DY along.
!            =  1  means sort DX in increasing order (ignoring DY)
!            = -1  means sort DX in decreasing order (ignoring DY)
!            = -2  means sort DX in decreasing order and carry DY along.
!
!***REFERENCES  R. C. Singleton, Algorithm 347, An efficient algorithm
!                 for sorting with minimal storage, Communications of
!                 the ACM, 12, 3 (1969), pp. 185-187.
!***ROUTINES CALLED  XERMSG
!***REVISION HISTORY  (YYMMDD)
!     761101  DATE WRITTEN
!     761118  Modified to use the Singleton quicksort algorithm.  (JAW)
!     890531  Changed all specific intrinsics to generic.  (WRB)
!     890831  Modified array declarations.  (WRB)
!     891009  Removed unreferenced statement labels.  (WRB)
!     891024  Changed category.  (WRB)
!     891024  REVISION DATE from Version 3.2
!     891214  Prologue converted to Version 4.0 format.  (BAB)
!     900315  CALLs to XERROR changed to CALLs to XERMSG.  (THJ)
!     901012  Declared all variables; changed X,Y to DX,DY; changed
!              code to  parallel SSORT. (M. McClain)
!     920501  Reformatted the REFERENCES section.  (DWL, WRB)
!     920519  Clarified error messages.  (DWL)
!     920801  Declarations section rebuilt and code restructured to use
!             IF-THEN-ELSE-ENDIF.  (RWC, WRB)
!***END PROLOGUE  DSORT
!     .. Scalar Arguments ..
```

```
      INTEGER KFLAG, N
!     .. Array Arguments ..
      DOUBLE PRECISION DX(*), DY(*)
!     .. Local Scalars ..
      DOUBLE PRECISION R, T, TT, TTY, TY
      INTEGER I, IJ, J, K, KK, L, M, NN
!     .. Local Arrays ..
      INTEGER IL(21), IU(21)
!     .. External Subroutines ..
!     EXTERNAL XERMSG
!     .. Intrinsic Functions ..
      INTRINSIC ABS, INT
!***FIRST EXECUTABLE STATEMENT  DSORT
      NN = N
!      IF (NN .LT. 1) THEN
!         CALL XERMSG ('SLATEC', 'DSORT',
!     +      'The number of values to be sorted is not positive.', 1, 1)
!         RETURN
!      ENDIF
!
      KK = ABS(KFLAG)
!      IF (KK.NE.1 .AND. KK.NE.2) THEN
!         CALL XERMSG ('SLATEC', 'DSORT',
!     +      'The sort control parameter, K, is not 2, 1, -1, or -2.', 2
!     +      1)
!        RETURN
!     ENDIF
!
!     Alter array DX to get decreasing order if needed
!
      IF (KFLAG .LE. -1) THEN
         DO 10 I=1,NN
            DX(I) = -DX(I)
   10    CONTINUE
      ENDIF
!
      IF (KK .EQ. 2) GO TO 100
!
!     Sort DX only
!
      M = 1
      I = 1
      J = NN
      R = 0.375D0
!
   20 IF (I .EQ. J) GO TO 60
      IF (R .LE. 0.5898437D0) THEN
         R = R+3.90625D-2
      ELSE
         R = R-0.21875D0
      ENDIF
!
   30 K = I
!
!     Select a central element of the array and save it in location T
!
      IJ = I + INT((J-I)*R)
      T = DX(IJ)
!
```

```
!     If first element of array is greater than T, interchange with T
!
      IF (DX(I) .GT. T) THEN
         DX(IJ) = DX(I)
         DX(I) = T
         T = DX(IJ)
      ENDIF
      L = J
!
!     If last element of array is less than than T, interchange with T
!
      IF (DX(J) .LT. T) THEN
         DX(IJ) = DX(J)
         DX(J) = T
         T = DX(IJ)
!
!        If first element of array is greater than T, interchange with T
!
         IF (DX(I) .GT. T) THEN
            DX(IJ) = DX(I)
            DX(I) = T
            T = DX(IJ)
         ENDIF
      ENDIF
!
!     Find an element in the second half of the array which is smaller
!     than T
!
   40 L = L-1
       IF (DX(L) .GT. T) GO TO 40
!
!     Find an element in the first half of the array which is greater
!     than T
!
   50 K = K+1
      IF (DX(K) .LT. T) GO TO 50
!
!     Interchange these elements
!
      IF (K .LE. L) THEN
         TT = DX(L)
         DX(L) = DX(K)
         DX(K) = TT
         GO TO 40
      ENDIF
!
!     Save upper and lower subscripts of the array yet to be sorted
!
      IF (L-I .GT. J-K) THEN
         IL(M) = I
         IU(M) = L
         I = K
         M = M+1
      ELSE
         IL(M) = K
         IU(M) = J
         J = L
         M = M+1
      ENDIF
      GO TO 70
!
!     Begin again on another portion of the unsorted array
!
   60 M = M-1
      IF (M .EQ. 0) GO TO 190
      I = IL(M)
      J = IU(M)
!
   70 IF (J-I .GE. 1) GO TO 30
      IF (I .EQ. 1) GO TO 20
      I = I-1
!
   80 I = I+1
      IF (I .EQ. J) GO TO 60
      T = DX(I+1)
      IF (DX(I) .LE. T) GO TO 80
      K = I
!
   90 DX(K+1) = DX(K)
      K = K-1
      IF (T .LT. DX(K)) GO TO 90
      DX(K+1) = T
      GO TO 80
!
!     Sort DX and carry DY along
!
  100 M = 1
      I = 1
      J = NN
      R = 0.375D0
!
  110 IF (I .EQ. J) GO TO 150
      IF (R .LE. 0.5898437D0) THEN
         R = R+3.90625D-2
      ELSE
         R = R-0.21875D0
      ENDIF
!
  120 K = I
!
!     Select a central element of the array and save it in location T
!
      IJ = I + INT((J-I)*R)
      T = DX(IJ)
      TY = DY(IJ)
!
!     If first element of array is greater than T, interchange with T
!
      IF (DX(I) .GT. T) THEN
         DX(IJ) = DX(I)
         DX(I) = T
         T = DX(IJ)
         DY(IJ) = DY(I)
         DY(I) = TY
         TY = DY(IJ)
      ENDIF
      L = J
!
```

```
!     If last element of array is less than T, interchange with T
!
      IF (DX(J) .LT. T) THEN
         DX(IJ) = DX(J)
         DX(J) = T
         T = DX(IJ)
         DY(IJ) = DY(J)
         DY(J) = TY
         TY = DY(IJ)
!
!        If first element of array is greater than T, interchange with T
!
         IF (DX(I) .GT. T) THEN
            DX(IJ) = DX(I)
            DX(I) = T
            T = DX(IJ)
            DY(IJ) = DY(I)
            DY(I) = TY
            TY = DY(IJ)
         ENDIF
      ENDIF
!
!     Find an element in the second half of the array which is smaller
!     than T
!
  130 L = L-1
      IF (DX(L) .GT. T) GO TO 130
!
!     Find an element in the first half of the array which is greater
!     than T
!
  140 K = K+1
      IF (DX(K) .LT. T) GO TO 140
!
!     Interchange these elements
!
      IF (K .LE. L) THEN
         TT = DX(L)
         DX(L) = DX(K)
         DX(K) = TT
         TTY = DY(L)
         DY(L) = DY(K)
         DY(K) = TTY
         GO TO 130
      ENDIF
!
!     Save upper and lower subscripts of the array yet to be sorted
!
      IF (L-I .GT. J-K) THEN
         IL(M) = I
         IU(M) = L
         I = K
         M = M+1
      ELSE
         IL(M) = K
         IU(M) = J
         J = L
         M = M+1
      ENDIF
      GO TO 160
!
!     Begin again on another portion of the unsorted array
!
  150 M = M-1
      IF (M .EQ. 0) GO TO 190
      I = IL(M)
      J = IU(M)
!
  160 IF (J-I .GE. 1) GO TO 120
      IF (I .EQ. 1) GO TO 110
      I = I-1
!
  170 I = I+1
      IF (I .EQ. J) GO TO 150
      T = DX(I+1)
      TY = DY(I+1)
      IF (DX(I) .LE. T) GO TO 170
      K = I
!
  180 DX(K+1) = DX(K)
      DY(K+1) = DY(K)
      K = K-1
      IF (T .LT. DX(K)) GO TO 180
      DX(K+1) = T
      DY(K+1) = TY
      GO TO 170
!
!     Clean up
!
  190 IF (KFLAG .LE. -1) THEN
         DO 200 I=1,NN
            DX(I) = -DX(I)
  200    CONTINUE
      ENDIF
      RETURN
      END
```

The Unix diff command will document the changes between the original Fortran 77 and the new Fortran 90 version.

As can be seen, converting the comment symbol from a C in column 1 to the ! character makes it valid free form Fortran 90.

40.10 Example 4: Simple Syntax Conversion to Modern Fortran

The Nag compiler offers a Polish option that will automatically convert Fortran 77 to Fortran 90.

Here is the converted version of the Netlib dsort.f subroutine.

```
subroutine dsort(dx, dy, n, kflag)
!***BEGIN PROLOGUE  DSORT
!***PURPOSE  Sort an array and optionally make the same interchanges in
!            an auxiliary array.  The array may be sorted in increasing
!            or decreasing order.  A slightly modified QUICKSORT
!            algorithm is used.
!***LIBRARY   SLATEC
!***CATEGORY  N6A2B
!***TYPE      DOUBLE PRECISION (SSORT-S, DSORT-D, ISORT-I)
!***KEYWORDS  SINGLETON QUICKSORT, SORT, SORTING
!***AUTHOR  Jones, R. E., (SNLA)
!           Wisniewski, J. A., (SNLA)
!***DESCRIPTION
!
!   DSORT sorts array DX and optionally makes the same interchanges in
!   array DY.  The array DX may be sorted in increasing order or
!   decreasing order.  A slightly modified quicksort algorithm is used.
!
!   Description of Parameters
!      DX - array of values to be sorted   (usually abscissas)
!      DY - array to be (optionally) carried along
!      N  - number of values in array DX to be sorted
!      KFLAG - control parameter
!                =  2  means sort DX in increasing order and carry DY along.
!                =  1  means sort DX in increasing order (ignoring DY)
!                = -1  means sort DX in decreasing order (ignoring DY)
!                = -2  means sort DX in decreasing order and carry DY along.
!
!***REFERENCES  R. C. Singleton, Algorithm 347, An efficient algorithm
!                 for sorting with minimal storage, Communications of
!                 the ACM, 12, 3 (1969), pp. 185-187.
!***ROUTINES CALLED  XERMSG
!***REVISION HISTORY  (YYMMDD)
!   761101  DATE WRITTEN
!   761118 Modified to use the Singleton quicksort algorithm.  (JAW)
!   890531 Changed all specific intrinsics to generic.  (WRB)
!   890831 Modified array declarations.  (WRB)
!   891009  Removed unreferenced statement labels.  (WRB)
!   891024  Changed category.  (WRB)
!   891024  REVISION DATE from Version 3.2
!   891214  Prologue converted to Version 4.0 format.  (BAB)
!   900315  CALLs to XERROR changed to CALLs to XERMSG.  (THJ)
!   901012  Declared all variables; changed X,Y to DX,DY; changed
!               code to parallel SSORT. (M. McClain)
!   920501  Reformatted the REFERENCES section.  (DWL, WRB)
!   920519  Clarified error messages.  (DWL)
!   920801  Declarations section rebuilt and code restructured to use
!                IF-THEN-ELSE-ENDIF.  (RWC, WRB)
!***END PROLOGUE  DSORT
!      .. Scalar Arguments ..
  integer kflag, n
!      .. Array Arguments ..
  double precision dx(*), dy(*)
!      .. Local Scalars ..
  double precision r, t, tt, tty, ty
  integer i, ij, j, k, kk, l, m, nn
!      .. Local Arrays ..
```

```
  integer il(21), iu(21)
!     .. External Subroutines ..
!      EXTERNAL XERMSG
!     ..  Intrinsic Functions ..
  intrinsic abs, int
!***FIRST EXECUTABLE STATEMENT  DSORT
  nn = n
!      IF (NN .LT. 1) THEN
!         CALL XERMSG ('SLATEC', 'DSORT',
!     +      'The number of values to be sorted is not positive.', 1, 1)
!         RETURN
!      ENDIF
!
  kk = abs(kflag)
!      IF (KK.NE.1 .AND. KK.NE.2) THEN
!         CALL XERMSG ('SLATEC', 'DSORT',
!     +      'The sort control parameter, K, is not 2, 1, -1, or -2.', 2,
!     +      1)
!        RETURN
!    ENDIF
!
!     Alter array DX to get decreasing order if needed
!
  if (kflag<=-1) then
    do i = 1, nn
      dx(i) = -dx(i)
    end do
  end if
!
  if (kk==2) go to 180
!
!     Sort DX only
!
  m = 1
  i = 1
  j = nn
  r = 0.375d0
!
100 if (i==j) go to 140
  if (r<=0.5898437d0) then
    r = r + 3.90625d-2
  else
    r = r - 0.21875d0
  end if
!
110 k = i
!
!     Select a central element of the array and save it in location T
!
  ij = i + int((j-i)*r)
  t = dx(ij)
!
!     If first element of array is greater than T, interchange with T
!
  if (dx(i)>t) then
    dx(ij) = dx(i)
    dx(i) = t
    t = dx(ij)
  end if
  l = j
```

```
!
!     If last element of array is less than than T, interchange with T
!
  if (dx(j)<t) then
    dx(ij) = dx(j)
    dx(j) = t
    t = dx(ij)
!
!        If first element of array is greater than T, interchange with T
!
    if (dx(i)>t) then
      dx(ij) = dx(i)
      dx(i) = t
      t = dx(ij)
    end if
  end if
!
!     Find an element in the second half of the array which is smaller
!     than T
!
120 l = l - 1
    if (dx(l)>t) go to 120
!
!     Find an element in the first half of the array which is greater
!     than T
! 130 k = k + 1
     if (dx(k)<t) go to 130
!
!     Interchange these elements
!
  if (k<=l) then
    tt = dx(l)
    dx(l) = dx(k)
    dx(k) = tt
    go to 120
  end if
!
!     Save upper and lower subscripts of the array yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 150
!
!     Begin again on another portion of the unsorted array
!
140 m = m - 1
     if (m==0) go to 270


     i = il(m)
     j = iu(m)
!
150 if (j-i>=1) go to 110
     if (i==1) go to 100
     i = i - 1
!
160 i = i + 1
    if (i==j) go to 140
    t = dx(i+1)
    if (dx(i)<=t) go to 160
    k = i
!
170 dx(k+1) = dx(k)
    k = k - 1
    if (t<dx(k)) go to 170
    dx(k+1) = t
    go to 160
!
!        Sort DX and carry DY along
!
180 m = 1
    i = 1
    j = nn
    r = 0.375d0
!
190 if (i==j) go to 230
    if (r<=0.5898437d0) then
       r = r + 3.90625d-2
  else
      r = r - 0.21875d0
  end if
!
200 k = i
!
!     Select a central element of the array and save it in location T
!
   ij = i + int((j-i)*r)
   t = dx(ij)
   ty = dy(ij)
!
!     If first element of array is greater than T, interchange with T
!
  if (dx(i)>t) then
    dx(ij) = dx(i)
    dx(i) = t
    t = dx(ij)
    dy(ij) = dy(i)
    dy(i) = ty
    ty = dy(ij)
  end if
  l = j
!
!     If last element of array is less than T, interchange with T
!
  if (dx(j)<t) then
    dx(ij) = dx(j)
    dx(j) = t
    t = dx(ij)
    dy(ij) = dy(j)
    dy(j) = ty
    ty = dy(ij)
!
!        If first element of array is greater than T, interchange with T
!
    if (dx(i)>t) then
      dx(ij) = dx(i)
      dx(i) = t
      t = dx(ij)
      dy(ij) = dy(i)
      dy(i) = ty
      ty = dy(ij)
    end if
  end if
!
!     Find an element in the second half of the array which is smaller
!     than T
!
210 l = l - 1
  if (dx(l)>t) go to 210
!
!     Find an element in the first half of the array which is greater
!     than T
!
220 k = k + 1
  if (dx(k)<t) go to 220
!
!     Interchange these elements
!
  if (k<=l) then
    tt = dx(l)
    dx(l) = dx(k)
    dx(k) = tt
    tty = dy(l)
    dy(l) = dy(k)
    dy(k) = tty
    go to 210
  end if
!
!     Save upper and lower subscripts of the array yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 240
!
!     Begin again on another portion of the unsorted array
!
230 m = m - 1
  if (m==0) go to 270
  i = il(m)
  j = iu(m)
```

```
!
240 if (j-i>=1) go to 200
  if (i==1) go to 190
  i = i - 1
!
250 i = i + 1
  if (i==j) go to 230
  t = dx(i+1)
  ty = dy(i+1)
  if (dx(i)<=t) go to 250
  k = i
!
260 dx(k+1) = dx(k)
  dy(k+1) = dy(k)
  k = k - 1
  if (t<dx(k)) go to 260
  dx(k+1) = t
  dy(k+1) = ty
  go to 250
!
!     Clean up
!
270 if (kflag<=-1) then
    do i = 1, nn
      dx(i) = -dx(i)
    end do
  end if
  return
end subroutine
```

As can be seen we have a much more Fortran 90 style after conversion. We use the Nag compiler polish option on all of our old Fortran 77 style code.

40.11 Example 5: Date Case Study

In this example we look at a variety of conversions. We start with a set of Fortran 77 functions and subroutines for date manipulation put together by Skip Noble.

We next look at a modern Fortran 90 version written by Alan Miller.

Both of these versions manipulate dates using independent integer variables to represent days, months, and years.

We next refer to the version in Chap. 22, where we introduce a date derived type throughout.

We will start by looking at the Fortran 77 version.

```
C======DATESUB.FOR with Sample Drivers.
C      COLLECTED AND PUT TOGETHER JANUARY 1972, H. D. KNOBLE .
C      ORIGINAL REFERENCES ARE CITED IN EACH ROUTINE.
C
```

```
       INTEGER YYYY,MM,DD,JD,WD,DDD,MMA,DDA,NDIFF,I
       INTEGER*2 YYYY2,MM2,DD2
C
C------IDAY IS A COMPANION TO CALEND; GIVEN A CALENDAR DATE, YYYY, MM,
C           DD, IDAY IS RETURNED AS THE DAY OF THE YEAR.
C           EXAMPLE: IDAY(1984,4,22)=113
      IDAY(YYYY,MM,DD)=3055*(MM+2)/100-(MM+10)/13*2-91
      ,          +(1-(MOD(YYYY,4)+3)/4+(MOD(YYYY,100)+99)/100
      ,          -(MOD(YYYY,400)+399)/400)*(MM+10)/13+DD
C
C------IZLR(YYYY,MM,DD) GIVES THE WEEKDAY NUMBER 0=SUNDAY, 1=MONDAY,
C      ... 6=SATURDAY.  EXAMPLE: IZLR(1970,1,1)=4=THURSDAY
       IZLR(YYYY,MM,DD)=MOD((13*(MM+10-(MM+10)/13*12)-1)/5+DD+77
       ,           +5*(YYYY+(MM-14)/12-(YYYY+(MM-14)/12)/100*100)/4
       ,           + (YYYY+(MM-14)/12)/400-(YYYY+(MM-14)/12)/100*2,7)
C
C  Compute date this year for changing clocks back to EST.
C  I.e., compute date for the last Sunday in October for this year.
      CALL GETDAT(YYYY2,MM2,DD2)
      YYYY=YYYY2
      DO I=31,26,-1
       IF (IZLR(YYYY,10,I).EQ.0) THEN
         WRITE(*,*) 'Turn Clocks back to EST on: ',I,' October ',YYYY
         EXIT
       ENDIF
      END DO
C  Compute date this year for turning clocks ahead to DST
C  I.e., compute date for the first Sunday in April for this year.
      CALL GETDAT(YYYY2,MM2,DD2)
      YYYY=YYYY2
      DO I=1,8
       IF (IZLR(YYYY,4,I).EQ.0) THEN
         WRITE(*,*) 'Turn Clocks ahead to DST on: ',I,' April ',YYYY
         EXIT
       ENDIF
      END DO
C
C  Is this a leap year? I.e. is 12/31/yyyy the 366th day of the year?
      CALL GETDAT(YYYY2,MM2,DD2)
C---GETDAT is builtin using most Compilers.
      YYYY=YYYY2
      IF(IDAY(YYYY,12,31).EQ.366) THEN
         WRITE(*,*) YYYY,' is a Leap Year'
      ELSE
         WRITE(*,*) YYYY,' is not a Leap Year'
      ENDIF


C
C  DAYSUB SHOULD RETURN: 1970, 1, 1, 4, 1
         CALL DAYSUB(JD(1970,1,1),YYYY,MM,DD,WD,DDD)
         IF(YYYY.NE.1970.OR.MM.NE.1.OR.DD.NE.1.OR.WD.NE.4.OR.DDD.NE.1)
     *  THEN
        WRITE(*,*)'DAYSUB Failed; YYYY,MM,DD,WD,DDD=',YYYY,MM,DD,WD,DDD
        STOP 1
      ENDIF
C  DIFFERENCE BETWEEN TWO SAME MONTHS AND DAYS OVER 1 LEAP YEAR IS 366.
      NDIFF=NDAYS(5,22,1984,5,22,1983)
      IF(NDIFF.NE.366) THEN
        WRITE(*,*) 'NDAYS FAILED; NDIFF=',NDIFF
      ELSE
C  RECOVER MONTH AND DAY FROM YEAR AND DAY NUMBER.
        CALL CALEND(YYYY,DDD,MMA,DDA)
        IF(MMA.NE.1.AND.DDA.NE.1) THEN
          WRITE(*,*) 'CALEND FAILED; MMA,DDA=',MMA,DDA
                                   ELSE
          WRITE(*,*) '** DATE MANIPULATION SUBROUTINES SIMPLE TEST OK.'
        END IF
      END IF
      STOP
      END

      SUBROUTINE CALEND(YYYY,DDD,MM,DD)
C==============CALEND WHEN GIVEN A VALID YEAR, YYYY, AND DAY OF THE
C             YEAR, DDD, RETURNS THE MONTH, MM, AND DAY OF THE
C              MONTH, DD.
C              SEE ACM ALGORITHM 398, TABLELESS DATE CONVERSION, BY
C              DICK STONE, CACM 13(10):621.
      INTEGER YYYY,DDD,MM,DD,T
      T=0
      IF(MOD(YYYY,4).EQ.0) T=1
C-----------THE FOLLOWING STATEMENT IS NECESSARY IF YYYY IS LESS TNAN
C            1900 OR GREATER THAN 2100.
      IF(MOD(YYYY,400).NE.0.AND.MOD(YYYY,100).EQ.0) T=0
      DD=DDD
      IF(DDD.GT.59+T) DD=DD+2-T
      MM=((DD+91)*100)/3055
      DD=(DD+91)-(MM*3055)/100
      MM=MM-2
C----------MM WILL BE CORRECT IFF DDD IS CORRECT FOR YYYY.
      IF(MM.GE.1 .AND. MM.LE.12) RETURN
      WRITE(*,1) DDD
1     FORMAT('0$$$CALEND: DAY OF THE YEAR INPUT =',I11,
     ,        ' IS OUT OF RANGE.')
      STOP 8
      END

      SUBROUTINE CDATE(JD,YYYY,MM,DD)
C=======GIVEN A JULIAN DAY NUMBER, NNNNNNNN, YYYY,MM,DD ARE RETURNED AS
C               AS THE CALENDAR DATE. JD=NNNNNNNN IS THE JULIAN DATE
C               FROM AN EPOCK IN THE VERY DISTANT PAST.  SEE CACM
C               1968 11(10):657, LETTER TO THE EDITOR BY FLIEGEL AND
C               VAN FLANDERN.
C    EXAMPLE CALL CDATE(2440588,YYYY,MM,DD) RETURNS 1970 1 1 .
C
      INTEGER JD,YYYY,MM,DD,L,N
      L=JD+68569
      N=4*L/146097
      L=L-(146097*N + 3)/4
      YYYY=4000*(L+1)/1461001
      L=L-1461*YYYY/4+31
      MM=80*L/2447
      DD=L-2447*MM/80
      L=MM/11
      MM=MM + 2 - 12*L
      YYYY=100*(N-49) + YYYY + L
      RETURN
      END
      SUBROUTINE DAYSUB(JD,YYYY,MM,DD,WD,DDD)
C========GIVEN JD, A JULIAN DAY # (SEE ASF JD), THIS ROUTINE
C        CALCULATES DD, THE DAY NUMBER OF THE MONTH; MM, THE MONTH
C        NUMBER; YYYY THE YEAR; WD THE WEEKDAY NUMBER, AND DDD
C        THE DAY NUMBER OF THE YEAR.
C        ARITHMETIC STATEMENT FUNCTIONS 'IZLR' AND 'IDAY' ARE TAKEN
C        FROM REMARK ON ALGORITHM 398, BY J. DOUGLAS ROBERTSON,
C        CACM 15(10):918.
C
C   EXAMPLE:  CALL DAYSUB(2440588,YYYY,MM,DD,WD,DDD) YIELDS 1970 1 1 4 1.
C
      INTEGER JD,YYYY,MM,DD,WD,DDD
C
C------IZLR(YYYY,MM,DD) GIVES THE WEEKDAY NUMBER 0=SUNDAY, 1=MONDAY,
C       ... 6=SATURDAY.  EXAMPLE: IZLR(1970,1,1)=4=THURSDAY
C
      IZLR(YYYY,MM,DD)=MOD((13*(MM+10-(MM+10)/13*12)-1)/5+DD+77
     ,             +5*(YYYY+(MM-14)/12-(YYYY+(MM-14)/12)/100*100)/4
     ,             + (YYYY+(MM-14)/12)/400-(YYYY+(MM-14)/12)/100*2,7)
C
C------IDAY IS A COMPANION TO CALEND; GIVEN A CALENDAR DATE, YYYY, MM,
C            DD, IDAY IS RETURNED AS THE DAY OF THE YEAR.
C            EXAMPLE: IDAY(1984,4,22)=113
C
      IDAY(YYYY,MM,DD)=3055*(MM+2)/100-(MM+10)/13*2-91
     ,                  +(1-(MOD(YYYY,4)+3)/4+(MOD(YYYY,100)+99)/100
     ,                  -(MOD(YYYY,400)+399)/400)*(MM+10)/13+DD
C
      CALL CDATE(JD,YYYY,MM,DD)
      WD=IZLR(YYYY,MM,DD)
      DDD=IDAY(YYYY,MM,DD)
      RETURN
      END

      FUNCTION JD(YYYY,MM,DD)
      INTEGER YYYY,MM,DD
C              DATE ROUTINE JD(YYYY,MM,DD) CONVERTS CALENDER DATE TO
C              JULIAN DATE.  SEE CACM 1968 11(10):657, LETTER TO THE
C              EDITOR BY HENRY F. FLIEGEL AND THOMAS C. VAN FLANDERN.
C    EXAMPLE JD(1970,1,1)=2440588
      JD=DD-32075+1461*(YYYY+4800+(MM-14)/12)/4
     ,              +367*(MM-2-((MM-14)/12)*12)/12-3*
     ,              ((YYYY+4900+(MM-14)/12)/100)/4
      RETURN
      END

      FUNCTION NDAYS(MM1,DD1,YYYY1, MM2,DD2,YYYY2)
      INTEGER YYYY1,MM1,DD1,YYYY2,MM2,DD2
C==============NDAYS IS RETURNED AS THE NUMBER OF DAYS BETWEEN TWO
C              DATES; THAT IS  MM1/DD1/YYYY1 MINUS MM2/DD2/YYYY2,
C              WHERE DATEI AND DATEJ HAVE ELEMENTS MM, DD, YYYY.
C-------NDAYS WILL BE POSITIVE IFF DATE1 IS MORE RECENT THAN DATE2.
      NDAYS=JD(YYYY1,MM1,DD1)-JD(YYYY2,MM2,DD2)
      RETURN
      END
```

Here some comments about the code.

- it is fixed format
- the Fortran code is upper case only
- variables names are a maximum of 6 characters
- There is no program statement at the start of the program
- default typing is in effect, with variables that begin with I-N as integer
- the following is a statement function

```
C
C------IDAY IS A COMPANION TO CALEND; GIVEN A CALENDAR DATE, YYYY, MM,
C           DD, IDAY IS RETURNED AS THE DAY OF THE YEAR.
C           EXAMPLE: IDAY(1984,4,22)=113
      IDAY(YYYY,MM,DD)=3055*(MM+2)/100-(MM+10)/13*2-91
     ,                 +(1-(MOD(YYYY,4)+3)/4+(MOD(YYYY,100)+99)/100
     ,                 -(MOD(YYYY,400)+399)/400)*(MM+10)/13+DD
```

- the following is a statement function

```
C
C------IZLR(YYYY,MM,DD) GIVES THE WEEKDAY NUMBER 0=SUNDAY, 1=MONDAY,
C       ... 6=SATURDAY.  EXAMPLE: IZLR(1970,1,1)=4=THURSDAY
      IZLR(YYYY,MM,DD)=MOD((13*(MM+10-(MM+10)/13*12)-1)/5+DD+77
     ,            +5*(YYYY+(MM-14)/12-(YYYY+(MM-14)/12)/100*100)/4
     ,            + (YYYY+(MM-14)/12)/400-(YYYY+(MM-14)/12)/100*2,7)
```

- The program has calls to a non-standard routine GETDAT

Here is the modern Fortran 90 version using independent integer variables for the days, months and years.

```
module date_sub

! COLLECTED AND PUT TOGETHER JANUARY 1972, H. D.
! KNOBLE .

! ORIGINAL REFERENCES ARE CITED IN EACH ROUTINE.

! Code converted using TO_F90 by Alan Miller
! Date: 1999-12-22 Time: 10:23:47
! Compatible with Imagine1 F compiler:
! 2002-07-19

  implicit none

  public :: iday, izlr, calend, cdate, ndays, &
    daysub, jd

contains

! ARITHMETIC FUNCTIONS "IZLR" AND "IDAY" ARE
! TAKEN FROM REMARK ON
! ALGORITHM 398, BY J. DOUGLAS ROBERTSON, CACM
! 15(10):918.
```

```
  function iday(yyyy, mm, dd) result (ival)
!   IDAY IS A COMPANION TO CALEND; GIVEN A
!   CALENDAR DATE, YYYY, MM,
!   DD, IDAY IS RETURNED AS THE DAY OF THE YEAR.
!   EXAMPLE: IDAY(1984, 4, 22) = 113

    integer, intent (in) :: yyyy, mm, dd
    integer :: ival

    ival = 3055*(mm+2)/100 - (mm+10)/13*2 - 91 + &
      (1-(modulo(yyyy,4)+3)/4+(modulo(yyyy, &
      100)+99)/100-(modulo(yyyy, &
      400)+399)/400)*(mm+10)/13 + dd

    return
  end function iday


  function izlr(yyyy, mm, dd) result (ival)
!   IZLR(YYYY, MM, DD) GIVES THE WEEKDAY NUMBER
!   0 = SUNDAY, 1 = MONDAY,
!   ... 6 = SATURDAY.  EXAMPLE: IZLR(1970, 1, 1)
!   = 4 = THURSDAY

    integer, intent (in) :: yyyy, mm, dd
    integer :: ival

    ival = modulo((13*(mm+10-(mm+10)/13*12)-1)/5 &
      +dd+77+5*(yyyy+(mm-14)/12-(yyyy+ &
      (mm-14)/12)/100*100)/4+(yyyy+(mm- &
      14)/12)/400-(yyyy+(mm-14)/12)/100*2, 7)

    return
  end function izlr

  subroutine calend(yyyy, ddd, mm, dd)
!   CALEND WHEN GIVEN A VALID YEAR, YYYY, AND
!   DAY OF THE YEAR, DDD,
!   RETURNS THE MONTH, MM, AND DAY OF THE MONTH,
!   DD.
!   SEE ACM ALGORITHM 398, TABLELESS DATE
!   CONVERSION, BY
!   DICK STONE, CACM 13(10):621.

    integer, intent (in) :: yyyy
    integer, intent (in) :: ddd
    integer, intent (out) :: mm
    integer, intent (out) :: dd

    integer :: t

    t = 0
    if (modulo(yyyy,4)==0) t = 1

!   ------THE FOLLOWING STATEMENT IS NECESSARY
!   IF YYYY IS < 1900 OR > 2100.
    if (modulo(yyyy,400)/=0 .and. &
      modulo(yyyy,100)==0) t = 0

    dd = ddd
    if (ddd>59+t) dd = dd + 2 - t
    mm = ((dd+91)*100)/3055
    dd = (dd+91) - (mm*3055)/100
    mm = mm - 2
!   ----------MM WILL BE CORRECT IFF DDD IS
!   CORRECT FOR YYYY.
    if (mm>=1 .and. mm<=12) return
    write (unit=*, fmt='(a,i11,a)') &
      '$$CALEND: DAY OF THE YEAR INPUT =', ddd, &
      ' IS OUT OF RANGE.'
    stop
  end subroutine calend

  subroutine cdate(jd, yyyy, mm, dd)
!   GIVEN A JULIAN DAY NUMBER, NNNNNNNN,
!   YYYY,MM,DD ARE RETURNED AS THE
!   CALENDAR DATE. JD = NNNNNNNN IS THE JULIAN
!   DATE FROM AN EPOCH
!   IN THE VERY DISTANT PAST.  SEE CACM 1968
!   11(10):657,
!   LETTER TO THE EDITOR BY FLIEGEL AND VAN
!   FLANDERN.
!   EXAMPLE CALL CDATE(2440588, YYYY, MM, DD)
!   RETURNS 1970 1 1 .

    integer, intent (in) :: jd
    integer, intent (out) :: yyyy
    integer, intent (out) :: mm
    integer, intent (out) :: dd

    integer :: l, n

    l = jd + 68569
    n = 4*l/146097
    l = l - (146097*n+3)/4
    yyyy = 4000*(l+1)/1461001
    l = l - 1461*yyyy/4 + 31
    mm = 80*l/2447
    dd = l - 2447*mm/80
    l = mm/11
    mm = mm + 2 - 12*l
    yyyy = 100*(n-49) + yyyy + l
    return
  end subroutine cdate


  subroutine daysub(jd, yyyy, mm, dd, wd, ddd)
!   GIVEN JD, A JULIAN DAY # (SEE ASF JD), THIS
!   ROUTINE CALCULATES DD,
!   THE DAY NUMBER OF THE MONTH; MM, THE MONTH
!   NUMBER; YYYY THE YEAR;
!   WD THE WEEKDAY NUMBER, AND DDD THE DAY
!   NUMBER OF THE YEAR.

!   EXAMPLE:
!   CALL DAYSUB(2440588, YYYY, MM, DD, WD, DDD)
!   YIELDS 1970 1 1 4 1.

    integer, intent (in) :: jd
    integer, intent (out) :: yyyy
    integer, intent (out) :: mm
    integer, intent (out) :: dd
    integer, intent (out) :: wd
    integer, intent (out) :: ddd

    call cdate(jd, yyyy, mm, dd)
    wd = izlr(yyyy, mm, dd)
    ddd = iday(yyyy, mm, dd)

    return
  end subroutine daysub


  function jd(yyyy, mm, dd) result (ival)

    integer, intent (in) :: yyyy
    integer, intent (in) :: mm
    integer, intent (in) :: dd
    integer :: ival

!   DATE ROUTINE JD(YYYY, MM, DD) CONVERTS
!   CALENDER DATE TO
!   JULIAN DATE.  SEE CACM 1968 11(10):657,
!   LETTER TO THE
!   EDITOR BY HENRY F. FLIEGEL AND THOMAS C. VAN
!   FLANDERN.
!   EXAMPLE JD(1970, 1, 1) = 2440588

    ival = dd - 32075 + 1461*(yyyy+4800+(mm-14)/ &
      12)/4 + 367*(mm-2-((mm-14)/12)*12)/12 - &
      3*((yyyy+4900+(mm-14)/12)/100)/4

    return
  end function jd

  function ndays(mm1, dd1, yyyy1, mm2, dd2, &
    yyyy2) result (ival)

    integer, intent (in) :: mm1
    integer, intent (in) :: dd1
    integer, intent (in) :: yyyy1
    integer, intent (in) :: mm2
    integer, intent (in) :: dd2
    integer, intent (in) :: yyyy2
    integer :: ival

!   NDAYS IS RETURNED AS THE NUMBER OF DAYS
!   BETWEEN TWO
!   DATES; THAT IS  MM1/DD1/YYYY1 MINUS
!   MM2/DD2/YYYY2,
!   WHERE DATEI AND DATEJ HAVE ELEMENTS MM, DD,
!   YYYY.
!   NDAYS WILL BE POSITIVE IFF DATE1 IS MORE
!   RECENT THAN DATE2.

    ival = jd(yyyy1, mm1, dd1) - &
      jd(yyyy2, mm2, dd2)

    return
  end function ndays

end module date_sub


program test_datesub

! ======DATESUB.FOR with Sample Drivers.

  use date_sub
  implicit none
  integer :: yyyy, mm, dd, wd, ddd, mma, dda, &
    ndiff, i
  integer, dimension (8) :: val

! Compute date this year for changing clocks
! back to EST.
! I.e.compute date for the last Sunday in
! October for this year.
  call date_and_time(values=val)
  yyyy = val(1)
  do i = 31, 26, -1
    if (izlr(yyyy,10,i)==0) then
      print *, 'Turn Clocks back to EST on: ', &
        i, ' October ', yyyy
      exit
    end if
  end do
! Compute date this year for turning clocks
! ahead to DST
! I.e., compute date for the first Sunday in
! April for this year.
  call date_and_time(values=val)
  yyyy = val(1)
  do i = 1, 8
    if (izlr(yyyy,4,i)==0) then
      print *, 'Turn Clocks ahead to DST on: ', &
        i, ' April ', yyyy
      exit
    end if
  end do


  call date_and_time(values=val)
  yyyy = val(1)

! Is this a leap year? I.e. is 12/31/yyyy the
! 366th day of the year?
  if (iday(yyyy,12,31)==366) then
    print *, yyyy, ' is a Leap Year'
  else
    print *, yyyy, ' is not a Leap Year'
  end if

! DAYSUB SHOULD RETURN: 1970, 1, 1, 4, 1
  call daysub(jd(1970,1,1), yyyy, mm, dd, wd, &
    ddd)
  if (yyyy/=1970 .or. mm/=1 .or. dd/=1 .or. &
    wd/=4 .or. ddd/=1) then
    print *, &
```

```
        'DAYSUB Failed; YYYY, MM, DD, WD, DDD = ', &
        yyyy, mm, dd, wd, ddd
      stop
    end if

  ! DIFFERENCE BETWEEN TO SAME MONTHS AND DAYS
  ! OVER 1 LEAP YEAR IS 366.
    ndiff = ndays(5, 22, 1984, 5, 22, 1983)
    if (ndiff/=366) then
      print *, 'NDAYS FAILED; NDIFF = ', ndiff
    else
  !   RECOVER MONTH AND DAY FROM YEAR AND DAY
  !   NUMBER.
      call calend(yyyy, ddd, mma, dda)
      if (mma/=1 .and. dda/=1) then
        print *, 'CALEND FAILED; MMA, DDA = ', &
          mma, dda
      else
        print *, '** DATE MANIPULATION SUBROUTINES &
          &SIMPLE TEST OK.'
      end if
    end if

    stop
  end program test_datesub
```

The next version using derived types and a modern Fortran 90 syntax can be found in Chap. 22.

This version required manual conversion. As can be seen by comparing the versions there is quite a difference.

The final version using an object oriented style can be found in Chap. 29. Again this required manual conversion.

40.12 Example 6: Creating 64 Bit Integer and 128 Bit Real Sorting Subroutines from the Netlib Sorting Routines

Netlib provides three non recursive sorting routines and they are

- dsort.f - Fortran 77 double precision, 64 bit normally
- ssort.f - Fortran default real type, 32 bit normally
- isort.f - Fortran default integer type, 32 bit normally

The aim is to provide a 64 bit integer sorting subroutine and a 128 bit real sorting subroutine, to accompany the above routines.

The first step is to rewrite the double precision version to use our precision module, and use that to create the 128 bit real subroutine.

The second step is to rewrite the 32 bit integer subroutine to use our integer kind module. We can then create our 64 bit integer sorting routine from that one.

Here are some of the major differences between the original Netlib version which uses double precision and the latest real versions which use kind types.

```
> subroutine dsort_dp(dx, dy, n, kflag)
> use precision_module , wp => dp
> implicit none
54c7
<   double precision dx(*), dy(*)
---
>   real (wp) ::  dx(*), dy(*)
56c9
<   double precision r, t, tt, tty, ty
---
>   real (wp) ::  r, t, tt, tty, ty

95c34
<   r = 0.375d0
---
>   r = 0.375_wp
97,99c36,38
< 100 if (i==j) go to 140
<   if (r<=0.5898437d0) then
<     r = r + 3.90625d-2
---
>   20 if (i==j) go to 60
>   if (r<=0.5898437_wp) then
>     r = r + 3.90625_wp/100.0_wp
101c40
<     r = r - 0.21875d0
---
>     r = r - 0.21875_wp

200c139
<   r = 0.375d0
---
>   r = 0.375_wp
202,204c141,143
< 190 if (i==j) go to 230
<   if (r<=0.5898437d0) then
<     r = r + 3.90625d-2
---
>   110 if (i==j) go to 150
>   if (r<=0.5898437_wp) then
>     r = r + 3.90625_wp/100.0_wp
206c145
<     r = r - 0.21875d0
---
>     r = r - 0.21875_wp
```

Here is the 128 bit real sort subroutine.

```
subroutine dsort_qp(dx, dy, n, kflag)
  use precision_module, wp => qp
  implicit none
```

```
! .. Scalar Arguments ..
  integer kflag, n
! .. Array Arguments ..
  real (wp) :: dx(*), dy(*)
! .. Local Scalars ..
  real (wp) :: r, t, tt, tty, ty
  integer i, ij, j, k, kk, l, m, nn
! .. Local Arrays ..
  integer il(21), iu(21)
! .. Intrinsic Functions ..
  intrinsic abs, int
! ***FIRST EXECUTABLE STATEMENT  DSORT
  nn = n
  kk = abs(kflag)
!
! Alter array DX to get decreasing order if
! needed
!
  if (kflag<=-1) then
    do i = 1, nn
      dx(i) = -dx(i)
    end do
  end if
!
  if (kk==2) go to 180
!
! Sort DX only
!
  m = 1
  i = 1
  j = nn
  r = 0.375_wp
!
100 if (i==j) go to 140
  if (r<=0.5898437_wp) then
    r = r + 3.90625_wp/100.0_wp
  else
    r = r - 0.21875_wp
  end if
!
110 k = i
!
! Select a central element of the array and save
! it in location T
!
  ij = i + int((j-i)*r)
  t = dx(ij)
!
! If first element of array is greater than T,
! interchange with T
!
  if (dx(i)>t) then
    dx(ij) = dx(i)
    dx(i) = t
    t = dx(ij)
  end if
  l = j
!
! If last element of array is less than than T,
! interchange with T
!
  if (dx(j)<t) then
    dx(ij) = dx(j)
    dx(j) = t
    t = dx(ij)
!
!   If first element of array is greater than T,
!   interchange with T
!
    if (dx(i)>t) then
      dx(ij) = dx(i)
      dx(i) = t
      t = dx(ij)
    end if
  end if
!
! Find an element in the second half of the
! array which is smaller
! than T
!
120 l = l - 1
  if (dx(l)>t) go to 120
!
! Find an element in the first half of the array
! which is greater
! than T
!
130 k = k + 1
  if (dx(k)<t) go to 130
!
! Interchange these elements
!
  if (k<=l) then
    tt = dx(l)
    dx(l) = dx(k)
    dx(k) = tt
    go to 120
  end if
!
! Save upper and lower subscripts of the array
! yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 150
```

```
!
! Begin again on another portion of the unsorted
! array
!
140 m = m - 1
  if (m==0) go to 270
  i = il(m)
  j = iu(m)
!
150 if (j-i>=1) go to 110
  if (i==1) go to 100
  i = i - 1
!
160 i = i + 1
  if (i==j) go to 140
  t = dx(i+1)
  if (dx(i)<=t) go to 160
  k = i
!
170 dx(k+1) = dx(k)
  k = k - 1
  if (t<dx(k)) go to 170
  dx(k+1) = t
  go to 160
!
! Sort DX and carry DY along
!
180 m = 1
  i = 1
  j = nn
  r = 0.375_wp
!
190 if (i==j) go to 230
  if (r<=0.5898437_wp) then
    r = r + 3.90625_wp/100.0_wp
  else
    r = r - 0.21875_wp
  end if

!
200 k = i
!
! Select a central element of the array and save
! it in location T
!
  ij = i + int((j-i)*r)
  t = dx(ij)
  ty = dy(ij)
!
! If first element of array is greater than T,
! interchange with T
!
  if (dx(i)>t) then
    dx(ij) = dx(i)
    dx(i) = t
    t = dx(ij)
    dy(ij) = dy(i)
    dy(i) = ty
    ty = dy(ij)
  end if
  l = j
!
! If last element of array is less than T,
! interchange with T
!
  if (dx(j)<t) then
    dx(ij) = dx(j)
    dx(j) = t
    t = dx(ij)
    dy(ij) = dy(j)
    dy(j) = ty
    ty = dy(ij)
!
!   If first element of array is greater than T,
!   interchange with T
!
    if (dx(i)>t) then
      dx(ij) = dx(i)
      dx(i) = t
      t = dx(ij)
      dy(ij) = dy(i)
      dy(i) = ty
      ty = dy(ij)
    end if
  end if
!
! Find an element in the second half of the
! array which is smaller
! than T
!
210 l = l - 1
  if (dx(l)>t) go to 210
!
! Find an element in the first half of the array
! which is greater
! than T
!
220 k = k + 1
  if (dx(k)<t) go to 220
!
! Interchange these elements
!
  if (k<=l) then
    tt = dx(l)
    dx(l) = dx(k)
    dx(k) = tt
    tty = dy(l)
    dy(l) = dy(k)
    dy(k) = tty
    go to 210
  end if
!
! Save upper and lower subscripts of the array
! yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 240
!
! Begin again on another portion of the unsorted
! array
!
230 m = m - 1
  if (m==0) go to 270
  i = il(m)
  j = iu(m)
!
240 if (j-i>=1) go to 200
  if (i==1) go to 190
  i = i - 1
!
250 i = i + 1
  if (i==j) go to 230
  t = dx(i+1)
  ty = dy(i+1)
  if (dx(i)<=t) go to 250
  k = i
!
260 dx(k+1) = dx(k)
  dy(k+1) = dy(k)
  k = k - 1
  if (t<dx(k)) go to 260
  dx(k+1) = t
  dy(k+1) = ty
  go to 250
!
! Clean up
!
270 if (kflag<=-1) then
    do i = 1, nn
      dx(i) = -dx(i)
    end do
  end if
  return
end subroutine dsort_qp
```

Here is the 64 bit integer sort subroutine.

```
subroutine isort_64(ix, iy, n, kflag)
  use integer_kind_module, wp => i64
  implicit none
! .. Scalar Arguments ..
  integer (wp) :: kflag, n
! .. Array Arguments ..
  integer (wp) :: ix(*), iy(*)
! .. Local Scalars ..
  real r
  integer (wp) :: i, ij, j, k, kk, l, m, nn, t, &
    tt, tty, ty
```

```
! .. Local Arrays ..
  integer (wp) :: il(21), iu(21)
! .. Intrinsic Functions ..
  intrinsic abs, int
! ***FIRST EXECUTABLE STATEMENT  ISORT
  nn = n
!
  kk = abs(kflag)
!
! Alter array IX to get decreasing order if
! needed
!
  if (kflag<=-1) then
    do i = 1, nn
      ix(i) = -ix(i)
    end do
  end if
!
  if (kk==2) go to 180
!
! Sort IX only
!
  m = 1
  i = 1
  j = nn
  r = 0.375e0
!
100 if (i==j) go to 140
  if (r<=0.5898437e0) then
    r = r + 3.90625e-2
  else
    r = r - 0.21875e0
  end if
!
110 k = i
!
! Select a central element of the array and save
! it in location T
!
  ij = i + int(((j-i)*r), wp)
  t = ix(ij)
!
! If first element of array is greater than T,
! interchange with T
!
  if (ix(i)>t) then
    ix(ij) = ix(i)
    ix(i) = t
    t = ix(ij)
  end if
  l = j
!
! If last element of array is less than than T,
! interchange with T
!
  if (ix(j)<t) then
    ix(ij) = ix(j)
    ix(j) = t
    t = ix(ij)
!
```

```
!   If first element of array is greater than T,
!   interchange with T
!
    if (ix(i)>t) then
      ix(ij) = ix(i)
      ix(i) = t
      t = ix(ij)
    end if
  end if
!
! Find an element in the second half of the
! array which is smaller
! than T
!
120 l = l - 1
  if (ix(l)>t) go to 120
!
! Find an element in the first half of the array
! which is greater
! than T
!
130 k = k + 1
  if (ix(k)<t) go to 130
!
! Interchange these elements
!
  if (k<=l) then
    tt = ix(l)
    ix(l) = ix(k)
    ix(k) = tt
    go to 120
  end if
!
! Save upper and lower subscripts of the array
! yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 150
!
! Begin again on another portion of the unsorted
! array
!
140 m = m - 1
  if (m==0) go to 270
  i = il(m)
  j = iu(m)
!
150 if (j-i>=1) go to 110
  if (i==1) go to 100
  i = i - 1
```

```
!
160 i = i + 1
  if (i==j) go to 140
  t = ix(i+1)
  if (ix(i)<=t) go to 160
  k = i
!
170 ix(k+1) = ix(k)
  k = k - 1
  if (t<ix(k)) go to 170
  ix(k+1) = t
  go to 160
!
! Sort IX and carry IY along
!
180 m = 1
  i = 1
  j = nn
  r = 0.375e0
! 190 if (i==j) go to 230
  if (r<=0.5898437e0) then
    r = r + 3.90625e-2
  else
    r = r - 0.21875e0
  end if
!
200 k = i
!
! Select a central element of the array and save
! it in location T
!
  ij = i + int(((j-i)*r), wp)
  t = ix(ij)
  ty = iy(ij)
!
! If first element of array is greater than T,
! interchange with T
!
  if (ix(i)>t) then
    ix(ij) = ix(i)
    ix(i) = t
    t = ix(ij)
    iy(ij) = iy(i)
    iy(i) = ty
    ty = iy(ij)
  end if
  l = j
!
! If last element of array is less than T,
! interchange with T
!
  if (ix(j)<t) then
    ix(ij) = ix(j)
    ix(j) = t
    t = ix(ij)
    iy(ij) = iy(j)
    iy(j) = ty
    ty = iy(ij)
!
!   If first element of array is greater than T,
!   interchange with T
!
    if (ix(i)>t) then
      ix(ij) = ix(i)
      ix(i) = t
      t = ix(ij)
      iy(ij) = iy(i)
      iy(i) = ty
      ty = iy(ij)
    end if
  end if
!
! Find an element in the second half of the
! array which is smaller
! than T
!
210 l = l - 1
  if (ix(l)>t) go to 210
!
! Find an element in the first half of the array
! which is greater
! than T
!
220 k = k + 1
  if (ix(k)<t) go to 220
!
! Interchange these elements
!
  if (k<=l) then
    tt = ix(l)
    ix(l) = ix(k)
    ix(k) = tt
    tty = iy(l)
    iy(l) = iy(k)
    iy(k) = tty
    go to 210
  end if
!
! Save upper and lower subscripts of the array
! yet to be sorted
!
  if (l-i>j-k) then
    il(m) = i
    iu(m) = l
    i = k
    m = m + 1
  else
    il(m) = k
    iu(m) = j
    j = l
    m = m + 1
  end if
  go to 240
```

```
!
! Begin again on another portion of the unsorted
! array
!
230 m = m - 1
  if (m==0) go to 270
  i = il(m)
  j = iu(m)
!
240 if (j-i>=1) go to 200
  if (i==1) go to 190
  i = i - 1
!
250 i = i + 1
  if (i==j) go to 230
  t = ix(i+1)
  ty = iy(i+1)
  if (ix(i)<=t) go to 250
  k = i
!
260 ix(k+1) = ix(k)
  iy(k+1) = iy(k)
  k = k - 1
  if (t<ix(k)) go to 260
  ix(k+1) = t
  iy(k+1) = ty
  go to 250
!
! Clean up
!
270 if (kflag<=-1) then
    do i = 1, nn
      ix(i) = -ix(i)
    end do
  end if
  return
end subroutine isort_64
```

All five subroutines are available on our web site.

```
isort_32.f90
isort_64.f90
dsort_sp.f90
dsort_dp.f90
dsort_qp.f90
```

We have also taken the generic recursive sort module from an earlier chapter and converted it to work with the Netlib routines. A copy of this module can also be found on our web site.

40.13 Summary

This chapter has shown some of the options open to you when working with legacy code. The emphasis has been on relatively straightforward code restructuring. The use of software tools to aid in this is highly recommended as converting manually using an editor is obviously going to involve much more work.

40.14 Problems

40.1 Compile and run the examples in this chapter.

40.2 Create a 16 bit integer sorting routine using the 32 bit integer sort subroutine in Example 5.

40.3 Create a generic sorting module from the subroutines in Example 5.