Introduction

See “Introduction to Fortran Conversion”

This does NOT teach the new features
See “Introduction to Modern Fortran”
Even then, most details are only in books

This describes only what can be done
Starting from a correct Fortran 90 program
Real junk in “Fortran Archaeology”
What Have We Here?

- Things to take advantage of modern features
  Mostly for “software engineering”
  Clarity, maintainability, error checking etc.

No old code will break in foreseeable future
Old code may not mean what you expect
So cleaning up those aspects is good

- Remember to balance gain against pain
  We shall cover a LOT of points
  Just note the things that you want to change
Reminder: Tools

f2f90 will do a few of them
And others are easy using Python or Perl

- Avoid doing manual edits if you can

Contact me if you have a conversion problem
PARAMETER (1)

INTEGER fortytwo
DATA fortytwo /42/

- If read-only, this can be replaced by:

INTEGER, PARAMETER :: fortytwo = 42

Now can’t write to it by accident
Makes it easier for compiler to optimise
PARAMETER arrays may be more efficient
PARAMETER (2)

PARAMETER defines a true constant
Can be used anywhere a constant can be
KIND=, CASE, initialisation, array bounds
No performance degradation in sane compilers

- Enables a lot of cleaning up
  Reduces problems with finger trouble
  And pre-editing hacks in build scripts
Recursion

Fortran 90 allows it – like DEC and Egtran
You must declare procedures **RECURSIVE**

Can clean up some old, **horrible hacks**
E.g. unnecessarily duplicated procedures
- Otherwise **don’t bother** about it

- Check if necessary **libraries** use/allow it
Procedure/Data Interfaces

Not just `INTERFACE`, but interfaces generally
Lots of improvements in Fortran 90
Much better error checking and ease of use
  • Probably most important improved area

Accounted for half of bugs in Fortran 77
Similar experience with many C codes
  • Fortran 90 can catch most such errors
Modules Are The Key

- Everything depends on modules
  Used to encapsulate declarations

- **Design** your modules carefully
  As the ultimate top-level structure
  Perhaps only a few, perhaps dozens
  Dependency graph shows visibility/usage

- Good place for high-level comments
  Please document purpose and interfaces
What is a Module

Bundles definitions/interfaces into a unit
• Similar to same term in other languages

Includes its static data definitions
And exported procedure interfaces
Actual code not part of module interface

Files may contain several modules
Modules may be split across many files
• But, in simplest use, keep them 1≡1
Module Structure

MODULE name
Static (often exported) data definitions
CONTAINS
Procedure definitions and interfaces
END MODULE name

Code may be included, or may be external
PUBLIC/PRIVATE

Can separate exported from hidden definitions

Fairly easy to use in simple cases
  • Worth considering when designing modules

No more details here, as largely new facility
In simplest uses, just does what you expect
Replace COMMON

Data modules are cleaner form of COMMON
BLOCK DATA becomes initialisation

Then just USE the module – much clearer
• A trivial change in clean code
The simplest use of modules possible

May extend module by moving code in there
E.g. auxiliary routines for that data
COMMON Example

INTEGER :: count, array(1000)
COMMON /awful/ count, array

Make a file (say awful.f90) containing:

MODULE awful
INTEGER :: count, array(1000)
END MODULE awful

USE awful
COMMON And SAVE

- Named COMMON did not imply SAVE
  But many programs assumed it
  And compilers usually implemented it

May need to add SAVE attributes

- Worth thinking whether you do
  E.g. scratch space does not need it
  There can be an efficiency cost
Explicit Interfaces (1)

Full declaration of procedure types
Not just result, but arguments, properties etc.
Like Algol 68, Pascal, ISO C, but more so

• All calls have all properties known
Give much better error checking
Allow use of many other new features
SUBROUTINE fred (array, opt, err)
USE double
REAL(KIND=FP) :: array(:)
REAL, INTENT(IN) :: opt
INTEGER, INTENT(OUT) :: err
Explicit Interfaces (2)

**Automatic** if procedures in modules
Or if calling internal procedures
I.e. any procedures following `CONTAINS`

These are only **fully secure** methods
- Also simplest – start with these

Can have separate interface modules
Or include interface declarations
- No examples given of this sort of use
Internal Procedures

PROGRAM name
Static (often exported) data definitions
CONTAINS
Procedure definitions and interfaces
END PROGRAM name

Also in SUBROUTINE and FUNCTION
But not in internal procedures themselves!
Separate Interfaces

May need to generate interfaces
Needed for multi-module **mutual recursion**
And when defining interfaces for non-Fortran
Including **Fortran 77** libraries as binary

Actual code in separate file (as **Fortran 77**)
- It is **NOT** checked against interface

- Do it by **f2f90** (or **NAGWare**), not by hand
Except for binary libraries and non-Fortran!
Keyword/Optional Arguments

Can simplify and clarify long lists
Often merge many procedures into one

- Don’t rush into this one, though
  Spend time on designing such interfaces
  Choosing the right defaults can be tricky
- KISS – Keep It Simple and Stupid

- Be careful when merging procedures
  Don’t forget to cross-check interactions
Keyword Example

SUBROUTINE fred (this, that, the, other)
REAL :: this, that, the, other

CALL fred(that=3, this=2, other=1, the=4)

Don’t have to remember the order of long lists
CALL fred(2, 3, 4, 1)
Simple Use Of OPTIONAL

• Use **OPTIONAL** for setting defaults only
  On entry, check and copy **ALL** args
  Use **ONLY** local copies thereafter
  Now, all variables well defined when used

• Can do the converse for optional results
  Just before returning, check and copy back

• Beyond this should be done only by experts
OPTIONAL Example (1)

FUNCTION fred (alf, bert)
REAL :: fred, alf, mybert
REAL, OPTIONAL, INTENT(IN) :: bert
IF (PRESENT(bert)) THEN
   mybert = bert
ELSE
   mybert = 0.0
ENDIF

Now use mybert in rest of procedure
OPTIONAL Example (2)

SUBROUTINE fred (alf, bert)
REAL :: alf
REAL, OPTIONAL, INTENT(OUT) :: bert
...
IF (PRESENT(bert)) bert = ...
END SUBROUTINE fred
FUNCTION Definitions

<type> FUNCTION fred (…)

• Fortran is giving up on this form
  Too many new facilities to bolt on

FUNCTION fred (…)
<type>, <attributes> :: fred
Precisions etc.

- Currently, need to use **double precision**
  But will start to need **64-bit integers**
  Already needed for most serious SMP codes
  Then *won’t* want it, by **default**
  Been the case on **Cray** systems for some time

This is what I recommend
Will **future-proof** your code
Also describe **currently** adequate solution
Define Module

Best solution:
MODULE precision
    INTEGER, PARAMETER :: FP = &
    SELECTED_REAL_KIND(14,200)
END MODULE precision

Currently OK (except on Cray):
MODULE precision
    INTEGER, PARAMETER :: FP = KIND(0.0D0)
END MODULE precision
IMPLICIT NONE etc.

• Add to every module, procedure, interface:
  USE precision
  IMPLICIT NONE

Forces declaration of almost everything
• Picks up a LOT of stupid mistakes

Following is allowed but NOT recommended:
  IMPLICIT <type> (<letter>−<letter>)
Using Precisions

REAL(KIND=FP) :: <declarations>
REAL(FP) :: <declarations>
COMPLEX(KIND=FP) :: <declarations>

- Add ‘_FP’ to all floating constants
  Don’t leave hostages to fortune . . .

- Do this using NAGWare or your own tool
  Very error-prone when done manually
Warning: Constants

REAL(), PARAMETER :: pi = <what?>

None of the following work:

. . . = 4.0D0*ATAN(1.0D0)
. . . = 3.14159265358979323846
. . . = M_PI     [ On Linux using cpp ]

The following does:

. . . = 3.14159265358979323846_FP
Make Functions Generic

Most intrinsic functions are now generic
Can change precision and even type easily
BUT you can’t pass them as actual arguments

- Change old, specific names to generic
  See section 13.6 of Fortran 2003 standard
  Clean up any uses of INTRINSIC or EXTERNAL

- Write wrappers if passed as arguments (rare)
Type Conversion

Most painful part of generic intrinsics

- **CMPLX** is a major trap for the unwary
  *MUST* specify **KIND** parameter when using
  Assuming use of non-default **REAL** (as above)

\[
\text{REAL, DBLE } \Rightarrow \text{ REAL(\ldots,KIND=FP)} \\
\text{CMPLX } \Rightarrow \text{ CMPLX(\ldots,KIND=FP)}
\]

- **INT** is *usually* safe enough
  Default is conventionally OK for array indexing
Current Shortcuts

If (and ONLY if) using $FP = \text{KIND}(0.0D0)$:

Can use ‘D’ as exponent letter

\[\ldots \pi = 3.14159265358979323846D0\]

$\text{REAL} \Rightarrow \text{DBLE}$

Not future-proof, but OK for a few years

\[\ldots \text{except on} \ \text{Cray} \ \ldots\]
Argument Passing

Major gotchas in Fortran in this area
It predates usual value-pointer model
It associates actual and dummy arguments
Expressions are stored in a hidden temporary

E.g. a COMMON variable as an argument
And then updated in COMMON during call
Effect is undefined – anything may happen
Other Association

- Also applies if imported or in COMMON
- Any two “names” for one location

Import/export is very like argument passing
- Be very careful exporting imports
- Don’t play games with renaming in USE

Watch out for EQUVALENCE – see later
Read-Only Dummy Arguments

- In Fortran 90, use `INTENT(IN)`
  Unfortunately, only protects against writing

- In Fortran 2003, consider `VALUE`
  Will take a copy of argument if needed
  Generally, not a good idea for arrays

Some existing codes will take a copy on entry
An old, adequate (but not fully safe) defence
Usually protects against it changing
Dummy Argument Example

SUBROUTINE fred (a, b)
REAL :: a
REAL, INTENT(IN), VALUE :: b
a = a * b + b

Following now becomes legal – hurrah!

REAL :: x = 1.23
CALL fred (x, x)
Passing Objects Twice (1)

- Always safe if all uses are read-only
  See above for how to declare that

**Disjoint** array sections are distinct variables
Array elements are distinct if different

Using array sections is clean, but check for copy

```fortran
CALL FRED(WORK(1:N), WORK(N+1:2*N))
```

Older method unclean but OK, **IF** within bounds

```fortran
CALL FRED(WORK, WORK(N+1))
```
Passing Objects Twice (2)

Beyond that, here be dragons . . .
Similar to storage association (see later)

- Avoid this if at all possible

If **ANY** use **MIGHT** update it
    and there is a non–VALUE argument

- Make sure **array sections** are **disjoint**
- Force **scalars** to be copied (**Fortran 77/90/95**)
Forcing A Copy

REAL :: x = 1.23

CALL fred (x, real(x,kind=kind(x)))

CALL fred (x, x+0.0)

y = x
CALL fred (x, y)
An Old Construction

INTEGER WORK(N)
CALL FRED(...,WORK,WORK,WORK)

SUBROUTINE FRED (...,WI,WR,WC)
INTEGER WI(*)
REAL WR(*)
COMPLEX WC(*)

Used for storage management in Fortran 77
• Use local workspace arrays, ALLOCATE etc.
Whole Array Operations

• Almost always much **clearer** and **shorter**
  Simpler code makes tuning much easier
  Efficient implementation isn’t always easy

Should be **more** efficient than **DO**–loops
Sometimes the **converse**, so check if necessary

• Don’t **force** the compiler to take copies
  Watch out for **unnecessary** copying, too
  Unfortunately, look for memory leaks, too
Tuning Array Operations

KISS – Keep It Simple and Stupid

Greatest gain is to move up one level
- Replace sections by BLAS or LAPACK
  Especially MATMUL $\Rightarrow$ xGEMM and up

Experts can do more with DO-loops
More control of space, ordering etc.
- Don’t rewrite well-tuned DO-loops
Array Examples

REAL :: A(:,:), B(:,:), C(:,:)

No compiler should copy anything
A = B*C/SUM(A)
A = MATMUL(B,C)

A = A+MATMUL(B,C)  ! It needn’t, but . . .

A = MATMUL(A,B)  ! Almost certainly a copy
Assumed-Shape Arrays

Can replace explicit shape or assumed size args

Except where bounds are absolute!
• Much more flexible, may be more efficient

• Replace passing array elements by sectioning

• Avoid conversion TO explicit/assumed size
Usually forces copying of the section
Watch out for compilers copying unnecessarily
Assumed-Shape Implementation

- Older methods need pass only pointer
  Almost required to be address of first element
  Bounds are fixed, passed explicitly, or similar
  Essentially same as C, whether C90 or C99

Assumed-shape passes descriptor, like Algol 68
Bounds passed implicitly, can be checked
- May not be contiguous, if section taken
Assumed-Shape Example

SUBROUTINE fred (a, b, c)
DO j = LBOUND(a,2), UBOUND(a,1)
   DO i = LBOUND(a,1), UBOUND(a,1)
      a(i,j) = DOT_PRODUCT(b(i,:),c(:,j))
   ENDDO
ENDDO
ENDDO

Reduces finger trouble when passing bounds
Workspace (Automatic) Arrays

Size of local arrays set at run–time
REAL :: array(<expression>)

• Can remove great deal of messy code
  Including lots of workspace arguments

Space should be recovered on return
Often mixes badly with ALLOCATE,
  array–valued functions, and similar
Details far beyond scope of this course
Array Masking

Operations on selected elements of array

Fortran 90 has WHERE assignment statement

• Much simpler than conditionals in loop

On most systems, little gain in efficiency
Real advantage is improvement in clarity

Watch out for unnecessary copying, again
Simple Masking

INTEGER :: k(1000)
REAL :: a(1000)

DO i = 1,1000
    IF (k(i) > 0) a(i) = SQRT(a(i))
ENDDO

Becomes:

WHERE (k > 0) a = SQRT(a)
More Complex Masking

INTEGER :: k(1000)
REAL :: a(1000)

WHERE (k <= 0)
    A = -1.0
ELSEWHERE (a < 0.0)
    a = 0.0
ELSEWHERE
    a = SQRT(a)
ENDWHERE
FORALL Statement

This is essentially multi-array masking. Fortran 95/2003 included it, from HPF.

Reliable source says slower than DO-loops. Sometimes by orders of magnitude.

- So advice is don’t use it in new code.
  But don’t bother to remove it from old code. Unless analysis shows it is a bottleneck.
Array-Valued Functions

You can write your own array-valued functions
Just as for scalars in Fortran 77
Some subroutines cleaner as functions

- Very commonly causes memory leaks
  This is a major implementation headache
  Details far beyond scope of this course

And unnecessary copying yet again . . .
Remove Labels (1)

Dijkstra was right but misquoted, as usual
Sometimes GOTOs can clarify control flow
• < 1% of those needed in Fortran 66

Can use for branch to error control block
But consider using internal procedures
See later about I/O exception handling

• Tools can handle only the simplest cases
Manual conversion easy but error-prone
Remove Labels (2)

IF (...) GOTO 100
    ...
GOTO 200
100 ...
200 CONTINUE

IF (...) THEN
    ...
ELSE
    ...
ENDIF
DO . . . ENDDO, EXIT, CYCLE, WHILE

- Note that DO loops can now be labelled

outer: DO
  inner: DO
    IF (...) CYCLE outer
    ENDDO inner
ENDDO outer
SELECT, CASE and DEFAULT
Executes one block out of the selection
• Much the rarest control construct
Following is more flexible:

IF (...) THEN
ELSEIF (...) THEN
ELSEIF (...) THEN
ELSE
ENDIF
Remove Labels (5)

Use for **FORMAT**s is cleaner, but unnecessary
Allowing both " ’ " and ’ " ’ is a *great* help!

• Can replace by character string

```
WRITE (*,'("Error ,I0, on ,I0")') IOSTAT, N
CHARACTER(*), PARAMETER :: f1 = '(...)
WRITE (*,f1) IOSTAT, N
```

• Or by calling an internal procedure
I/O

This is a traditional weak point
Fortran 90 included significant upgrades
Fortran 2003 has many minor improvements

Still many unnecessary restrictions
• And most compilers are not Fortran 2003

• Most common problem is free-format input
Localise any problem I/O and possibly call C
I/O Errors

• ERR and END $\Rightarrow$ IOSTAT or IOMSG
  Potentially provides more information anyway
  IOMSG is best, but only in Fortran 2003

I/O error handling is generally no better
• Format errors on reading still undefined
  But all compilers seem to set IOSTAT

• Generally not worth cleaning this up
  Except to remove use of labels
OPEN and INQUIRE

Lots of minor improvements, few important
• Opening file twice for input still illegal

ACTION=’READ’ or ’WRITE’ or ’READWRITE’
• Definitely use this, in all OPENs
Can be critical in some circumstances

Can set defaults for most formatting modes
Non-Advancing I/O (1)

Doesn’t move to new record if more data
Don’t confuse it with C’s streaming model
Unfortunately has huge number of constraints

Not list-directed, not on internal files . . .
Little use for free-format input or output

Can use to build out output records in parts
Useful for prompting, but has problems
Non-Advancing I/O (2)

Can use to read unknown length records
But only as far as raw characters

```
CHARACTER(LEN=1) :: buffer(100)
READ(ADVANCE='NO',EOR=last,SIZE=len)
```

Rest of record (if any) is read next time
Unpick the buffer as an internal file

- Generally, using PAD is easier – see later
Free-Format Input

Still only **list-directed** I/O
- Can now use with internal files!
Still no way to tell how many items read
And non-advancing I/O is not allowed

Can use to unpick buffers created as above
Continue to **set all values** before reading

- Not worth a conversion campaign
Asynchronous I/O

New in Fortran 2003, and fairly clean
• But not widely available, and won’t be

Contact me for sordid details
POSIX makes a complete mess of this
Microsoft doesn’t do much better

• Right semantics for MPI non-blocking
Hope for a decent MPI-3 binding to Fortran 90
Other I/O Enhancements

PAD= allows reading space-trimmed records
DELIM= for strings in list-directed I/O
SIGN= for whether you want ‘+’ or not

Fortran 2003 output allows ’I0,F0.3’ etc.
Plus lots of slightly useful descriptors
• Free-format output is now more-or-less OK

Fortran 2003 FLUSH statement – about time!
ANSI Control Characters

First column of **SOME** formatted output units
Absolutely no way of telling which ones

' ' = next line, '0' = skip line, '1' = new page
'+' = overprint, sometimes also '2'–'7'
[ Latter were unreliable, like C ' \r', ' \f', ' \v' ]

- Dropped in **Fortran 2003** – no replacement
Convert any code that uses old convention
Probably no compilers still rely on it
Pure Procedures

No side-effects – usable in parallel
Like computer science “strictly functional”

They don’t write to global data or use SAVE
All function arguments are INTENT(IN)
No external I/O or STOP statements
Some other constraints on pointers

• If you can, write functions like this
Can declare as PURE or ELEMENTAL
• Not always feasible and hinders debugging
Features To Avoid

Not officially deprecated
Mostly because of political objections
Many have a few justifiable uses
Most have been undesirable for decades

- Remove them if you possibly can
- Localise and document them if you can’t
Ask for advice if you have difficulty
Implicit Main Program

The **PROGRAM** statement is optional
You are recommended to add/use it
Only to make *your* life easier

Especially if **comments outside procedures**
Makes processing easier for simple tools
E.g. checking for only one main program!
**INCLUDE**

INCLUDE ’<name>’ – usually a filename
It replaces the line by the text
May be INCLUDE (<name>) in Fortran IV

- Generally, replace by a **module**
Rare cases where that doesn’t make sense

*Fortran 95* has optional preprocessor “**Coco**”
Open source implementation, but few vendors
Use of C Preprocessor

Very common, but a snare and a delusion
• C’s rules VERY different from Fortran’s
Often if fred.F or joe.F90, vs fred.f or joe.f90

#include ’<filename>’
define <name> <expression>
#if (<expression>)

• Consider whether you can get rid of this
Impure Functions (1)

Have always been **undefined** behaviour
But in a particular way before **Fortran 90**
Basically **write–once / read–many** rule
  * No **guarantee** that any function call is made

Situation is unclear in **Fortran 90/95/2003**
Some people say totally **undefined** (illegal)
Others say same as **ANSI Fortran 77**
Avoid this extremely nasty mess if you can
Impure Functions (3)

Safest use is for *random numbers* and similar

Some *local state* is saved between calls

- Updating *global* state for *experts only*
- Reading *updatable* global state is as bad

Use *separate* module and file; avoid *inlineing*

- *Never* export the local state as data
- Don’t use twice in *same* statement

Includes use within *another* function call
Impure Functions (4)

COMPLEX FUNCTION FRED (ARG)
COMPLEX, SAVE :: COPY
COPY = ARG
FRED = ...

COMPLEX FUNCTION JOE (ARG)
JOE = CONJG(FRED(ARG))

X = FRED(1.23)+JOE(4.56)
• Is NOT allowed and may well not work
Impure Functions (4)

Lots of other, fairly safe uses
Constraints same as for random numbers

Cache of common arguments and results
Can keep trace buffer or update use count
Can do I/O if careful (e.g. diagnostics)

- Twice in same statement needs thread safety
Possible safely, but neither easily nor portably
EQUIVALENCE (1)

Used to overlap arrays to save space
But, strangely, not on dummy arguments
Non-trivial uses create horrible errors
And can interfere with optimisation

Modern computers have lots of memory
Consider ALLOCATABLE or POINTER
• Overlap arrays only when essential
• Use it very simply and very cleanly
EQUIVALENCE (2)

Used to play bit twiddling tricks
E.g. to unpick floating-point formats
Undefined behaviour, and means it, too
- Common cause of portability problems

- Localise any such tricks in small modules
Can sometimes replace by new functions
Can compile them with no optimisation
Or replace them by C or assembler
Reshaping via Arguments

DOUBLE PRECISION X(10,20)
CALL FRED(A(5,5))

SUBROUTINE FRED (A)
DOUBLE PRECISION A(25)

Legal but ill-defined in Fortran 66
Dubiously illegal in Fortran 77
Well-defined in Fortran 90 and beyond
• But should be avoided, anyway
Other Storage Association

Can also be done via COMMON – see earlier

All methods can be used cleanly or revoltingly
Equating different base types is worst form
Get rid of that use, if at all possible

- Legal or safe use is fiendishly tricky
Rules have changed over the years, too
Interferes badly with optimisation
Examples Of Bad Cases

REAL X(20)
INTEGER N(20)
EQUIVALENCE (X, N))

INTEGER N(4,10)
CALL FRED (N)
...
SUBROUTINE FRED (A)
DOUBLE PRECISION A(20)
Routine Structure

Before 1980s, calls were SLOW
Almost no compiler inlined calls

- Consider splitting up complex routines
Repeted code can become internal procedure

Several features to avoid routine calls
Most are strongly deprecated or deleted
Main remaining one is ENTRY
ENTRY (1)

FUNCTION FRED (A, B)
...
ENTRY JOE (N)
...

One procedure with several interfaces
Yes, it’s utterly horrible
VERY hard to use correctly

- Replace by separate, simple wrappers
Different interfaces to a common auxiliary
ENTRY (2)

FUNCTION FRED (A, B)
CALL BERT (1, X, A, B, M, N)
FRED = X

FUNCTION JOE (N)
CALL BERT (2, X, A, B, M, N)
JOE = M

Much easier to understand and debug
ENTRY (3)

You could do that using **OPTIONAL** args
Definitely advanced use for **experts only**

- If you have to ask how, please don’t try
  Even if you do, **think twice** before doing so

- The difficulty is **intrinsic** to the problem
  It is **NOT** caused by **ENTRY** syntax
BACKSPACE, ENDFILE etc.

Correspond to long-dead filesystem models
Fruitful source of traps on modern systems

- Replace BACKSPACE by internal files
- Replace ENDFILE by CLOSE or REWIND
  Or by redesigning I/O interface

- Don’t use formatted, direct-access files
Similar problems to ones for C – lots!
Fortran 66/77 Relics

Obsolescent in Fortran 95/2003
You will definitely see many of these
They will still work but should not be used

Most can be covered fairly briefly
Almost all sane code is easy to modernise
But may be very tedious by hand
Use an automatic tool where possible
Fixed Form Source (1)

Comments have ‘C’ in column 1
Labels in columns 1–5
Statement in columns 7–72 only
Columns 73–* ignored (for sequence numbers)
If column 6 not a space or ’0’:
join columns 7–72 onto previous line
Spaces ignored and not needed (ex. Hollerith)
  G OTO12
  0 1 2 CALLMY SUB(9  8)
  DO 10 I = 1.10
Fixed Form Source (2)

You don’t need to write such perverse code
But details are complicated for newcomers
Truncation at column 72 is a major trap
And not all compilers did it . . .

Main surviving relic of punched cards

- Convert using f2f90 (or NAGWare)
- Or write your own Python/Perl converter
- By hand is very tedious and error-prone
Arithmetic IF

IF (<expression>) <label>, <label>, <label>
Branches to labels if negative, zero or positive

- Useful, clean, but ‘unstructured’

<temporary> = <expression>
IF (<temporary> .LT. 0) THEN
ELSEIF (<temporary> .EQ. 0) THEN
ELSE
ENDIF

Most compilers optimise use of <temporary>
DO Loop Issues

DO 10 K = 1,10
DO 10 J = 1,10
10 CONTINUE

DO 10 K = 1,10
10 WRITE (*,*) K

- Convert to DO . . . ENDDO form
Alternate Return

CALL FRED (A, *<label>, *<label>)
or (in Fortran IV and derivatives):
CALL FRED (A, &<label>, &<label>)
RETURN <N> branches to the Nth label

• Simplest to add an integer code as last argyment
And use it in a CASE statement after the call
Computed GOTO

GOTO (<label>, ...) <integer variable>
Just a GOTO form of the CASE statement

• Replace by the CASE statement

To connoisseurs of the arcane and bizarre:
Look up second-level definition in Fortran 66
Statement Functions

FRED (ARG) = 5.0*ARG+2.0

**IF** in right place, and FRED not function/array
- Infernally hard to recognise in code
  Rather like a C `#define` in some ways

- Replace by an internal procedure
  Cleaner and much more flexible

See also Fortran 2003 ASSOCIATE
DATA Statement Ordering

Could occur almost anywhere (like FORMAT)

**Simple**: just move them into declarations
**Better**: replace by PARAMETER or initialisers

Incidentally, tidying up FORMAT is good
Put after READ/WRITE or at end
Best to replace, as described earlier
Assumed Length Character Functions

FUNCTION FRED (ARG)
CHARACTER (LEN=*) :: FRED

SC22/WG5 finally sees the light . . .
Length taken from context – don’t ask

• Redesign any such function, totally
Most character lengths should be constants
Or result length copied from an argument
A Generic Character Function

FUNCTION FRED (ARG)
CHARACTER (LEN=*) :: ARG
CHARACTER (LEN=LEN(ARG)) :: FRED
FRED = ARG
END

Beyond that, little hope of optimisation
Also can run risk of memory leaks
CHARACTER*<length> Declarations

CHARACTER*80 CARDS(1000)
CHARACTER*80 FUNCTION CARD (ARG)
and (in Fortran IV and derivatives):
CHARACTER FUNCTION CARD*80 (ARG)

- Use LEN= type parameter instead

I recommend avoiding even:
CHARACTER :: A*10, B*20