Fortran, C or C++ for geophysical software development ?

by

Les Hatton[1]

Abstract

Historically, Fortran has been used for most of the major software development in the geophysical environment since the emergence of digital seismic data processing. Today, largely as a result of an extraordinarily protracted attempt to modernise the language, its position is challenged by a number of competing languages. These include C, recently standardised by the I.S.O., and C++ now being standardised by the A.N.S.I. committee X3J16. Given that standardisation is one of the most important forces in modern computing as exemplified by the emergence of P.O.S.C. in the petroleum exploration industry, a discussion of this much aired question seemed timely. This paper discusses the advantages and disadvantages of these languages with particular emphasis on the reliability of the software produced using them and the notion of the "safe subset". The paper concludes by defining a safe(r) subset of the Fortran 77 language. It is intended that further safe(r) subsets be defined in future papers. Note that the use of the parenthesised (r) in safe is deliberate. There is no truly safe subset of a high-level language as errors occur in compilers and chip firmware also but use of so-called safe features greatly improves the overall reliability and reduces the maintenance costs.

Introduction

One of the most common questions asked of the author in recent years has been which of the many programming languages should now be used for geophysical software development ? The question has never been more difficult to answer. In times past, the more or less absolute pre-eminence of Fortran admitted only one answer. Today, the position is more confused

1Programming Research Ltd., Waynflete House, 74-76 High Street, Esher, Surrey, KT10 9QS, U.K

given the number of candidate languages, but oddly enough, clearer in the sense that mixed language environments are becoming increasingly more common. Perhaps the latter follows naturally from the former.

What then are the relevant competing programming languages ? To the author's knowledge, the following languages have been used in some way for geophysical software development in recent times:

Fortran

This is by far the most commonly used language and therefore worthy of the biggest sub-section. It's first standardisation was in 1966 and a later one intended to promote portability in 1978, (known as Fortran 77). Most of the world's geophysical software is written in one or other of these two dialects and a large amount of effort has been expended in recent years bringing Fortran 66 packages 'up to date' with the Fortran 77 standard.

At the end of the 1970's, standardisation re-started so as to define a modern Fortran for the 1980's. An extraordinary saga then began to unfold with more intrigue than the average 'soap opera'. To date, this standardisation is not yet formally complete after some 13 years, although the language was restricted to editorial change only during 1991. Even at this stage, it had some very powerful detractors on the standards committee itself. The standard is to be known as Fortran 90. It is the ultimate 'kitchen-sink' language with many new features, some of them untried in any previous language whilst still maintaining complete compatibility with Fortran 77, (although nobody seems quite sure of this).

The standard is very controversial as exemplified by the fact that the development was taking so long that ANSI re-validated the older Fortran 77 standard in an unexpected step. So, there will be *two* Fortran standards, which was supposed never to happen. Then again, language standards committees are supposed to standardise existing practice rather than inventing new and largely untested features so more than one fundamental rule was broken. Where does this leave the potential user ? The general appreciation seems to be that Fortran 90 is so complicated with many features and more importantly, many exceptions to each rule that compilers for the language are unlikely to be commonly available for some considerable time, although there is at least one apparently full

implementation available now. There remain unfortunately a large number of potential unreliability concerns.

What does this language contain for the potential geophysical user ? The answer quite simply is everything and more besides. There are structures, pointers, array and vector notation, provision for information hiding in modules and many other features. This has led to a language which is some two and a half times bigger than Fortran 77. This author at least believes that the language will be a minefield for the unwary for a long time to come and prospective users should be aware of this. It is not a language in which to start writing a large package using relatively inexperienced programmers, bearing in mind that experience in Fortran 77 is not much help as the new features largely involve throwing away most of the Fortran 77 features. (This practice was actually formalised at one stage by the practice of *deprecating* them, although this has now been removed leaving a somewhat smaller number of the original Fortran 77 features as *obsolescent,* i.e. candidates for eventual removal). The step from Fortran 77 to Fortran 90 is arguably therefore as large as the step from Fortran 77 to C if the new features are to be used.

C

A language now perceived in many quarters as a replacement to Fortran in view of the standardisation traumas of that language as described above. Its use is growing rapidly in areas as diverse as banking systems and personal computer software which it dominates.

A surprisingly lengthy standardisation process culminated in December 1989 when the language was standardised as ANSI (and now ISO) C. A language originally infamous for allowing the programmer enough rope to hang him or herself with, the standardisation process was successful in eliminating some of its worst reliability problems although there remain many compromises, some simply ugly, for example, the pre-processor definition. Most manufacturers are striving to make their C compilers ISO compliant although few are yet close.

PL/1

Another very large programming language popular in the 1970' s. It

was originally supposed to contain all that was good from Algol, Cobol and Fortran. It finished up containing everything, good, bad or indifferent. It's use today seems to be confined almost entirely to the ever-shrinking mainframe market.

Assembler

The original programming language, (the "ultimate dead language" according to Kernighan & Ritchie of Bell Labs), just one step up from machine language. Surprisingly, a few packages written in this language are still in use today under the excuse of efficiency. This must be considered to be at best highly suspect and at worst deranged in modern computing terms, with horrendous implications for maintenance and portability.

Basic

Very popular for modest micro-computer based geophysical packages but not to the author's knowledge used seriously on any large packages. Modern structured basic should not however be considered a diminutive language.

Pascal

Once the great white hope of language purists, its use has unfortunately been very restricted in the geophysical industry although the author knows of a number of serious packages written in dialects of this language. As a programming language, its use positively encourages safety in software development, although this admirable goal is still unfortunately a second best to those people who want ever more features in programming languages. It did not survive the standardisation process well.

APL

This iconoclastic language is actually used quite widely in the geophysical world, a fact which rather took the author by surprise, although it is not used for large scale development. Its legendary brevity is both a formidable strength and also a formidable weakness in maintenance terms.

To the above list could be added the following which although not in widespread use in the industry, could easily be:

Ada

Until relatively recently, the darling of the defence industries. It was originally designed as the all-purpose language with the reliability to be used in safety critical environments. After discussing this with a number of companies, the author concludes that it appears to have the following major drawbacks:

1. A very large retraining cost in converting programmers from other languages, estimated by some at around $100,000 per programmer.

2. It appears to have portability problems, or at least some features of the language do. This may appear surprising but has been substantiated by a number of users. Note that, unfortunately, subsets are forbidden and this may work against the development of a "safe(r) subset".

3. Its use appears to be diminishing. At one time, some major governments, notably the U.S.A., mandated its use. However, its high implied development costs have led to the unusual practice developing whereby contractors have to propose an Ada solution but are also allowed to propose a (much) cheaper solution using another language, for example, C. In these harsh economic times, the result is obvious.

C++

Perceived as the new great white hope of software quality and the vehicle for the much vaunted object orientated programming in spite of the fact that its support for this paradigm is not the best. The growth of interest in this language is little short of phenomenal, with many users of C interested in its development, particularly as it removes a number of reliability problems present in C. There is a widely held belief that it a superset of ANSI C. **This is wrong** as evidenced by the attitudes of both ANSI C and ANSI C++ committees. Actual differences will be described in more detail in the section comparing the two languages.

Public opinion in the geophysical industry as far as the author can see generally covers Fortran, C or C++ and it is these languages to which the rest of the paper will confine itself.

Comparisons

Fortran v. C/C++

As has been discussed above, Fortran has in the past almost always been the choice of the numerical computation practitioners whereas C has historically been associated with text processing and Unix. Be that as it may, C has become a language in which numerical computation needs can be fulfilled most satisfactorily to the extent that C compilers have appeared before Fortran compilers on some commonly used machines in scientific computation. To complicate things further, C is quite often the *output* language of Fortran compilers, which is then subsequently compiled by a C compiler. The author has had considerable experience with *f2c*, (Feldman, Gay, Maimone and Schryer, (1990)), which is a Fortran 77 to C converter. This admirable piece of public-domain software not only produces more sensible error messages than many Fortran compilers, but also for some applications, produces code which is more efficient. It should be noted however, that if frequent use is made of extensions to the Fortran 77 language, *f2c* will **not** be of use. When parsimonious use is made of Fortran, the results are indeed impressive. As an exercise, the author converted a 128,000 line geophysical application, compiled the resulting C and produced an executable image without any problems either then or subsequently with the executable in about 30 minutes on an 18 mip Unix workstation.

In the author' sexperience, almost all Fortran sites interviewed are beginning to use C and most see it ultimately as either a replacement for their Fortran or as an important adjunct leading to mixed Fortran and C environments. This latter will be discussed further below. The author has never encountered a site moving from C to Fortran. The rapid shift to Unix in general and graphical user interfaces such as X11 in particular, naturally favour the growth of C and C++.

One important development in C++ is the appearance of *class libraries* for vector and matrix arithmetic. These may offer the numerical

scientist today what may take several years to appear reliably in Fortran 90 compilers.

C v. C++

As was described above, these are considered separate languages, even given their strong relationship. There are unfortunately 8 key areas in which the languages differ. These are:

1. Identifier name space, (objects / tags).

   Example:

   struct s_tag {int i,j;} s_tag;

   is legal C but illegal C++.

2. Syntax

   C allows redundant parenthese in declarations, C++ does not.

   Example:

   int (*obj);

   is legal C but illegal C++.

3. Keywords

   C++ contains keywords which are not in C, such as class, delete, handle and a few others. For true compatibility, these should not be used in C.

4. Type safe function calls

   C++ requires functions to be declared before use, C does not, although it is a good idea anyway. Also *extern int f()* in C++ is equivalent to the C construct *extern int f(void).* This will cause problems for C programmers. There are a number of other subtle issues in this category.

5. The use of const.

C++ treats objects declared as *const* as static so they only have file scope at most. For compatibility, C programmers should precede declarations of *const* with *extern* and in one of the occurrences, the declared constant should be initialised.

6. Multiple definitions.

   C allows multiple definitions of the same object in the same file with external scope, for example:

   int i;

   int i;

   This is illegal in C++, (and pretty bad C practice).

7. Character literals such as ' a' .

   In C, character literals are of type *int* and in C++ of type *char*. This has implications when *sizeof ('c')* is used for example.

8. Declarations and array initialization.

   C++ requires strings of characters to be null terminated, C does not.

   Example:

   char string[2] = "ab";

   is legal in C but illegal in C++ which requires:

   char string[2] = "ab\0";

If these are avoided, users can remain compatible with the two languages assuming that C++ does not diverge further. Many C users seem to view C++ as an eventual replacement for C, although there is no guarantee of this so *compatibility as described above is an important goal to attain*. If the prospective user would like more detail, the paper by Jones (1991) is recommended.

The mixed environment

There is a rapid growth of packages which use C in some places and Fortran in others. This can be *tightly coupled* whereby C programs reference Fortran functions and vice versa or *loosely coupled* whereby a C based G.U.I. starts a Fortran based application via a C system call for example and captures the output. There seem to be few portability penalties at least in the Unix environment. Consequently, this seems a far more sensible step than the practice of using strange extensions to Fortran 77. For example, if an application requires structures and unions or pointers, that part should be written portably in C and then called from Fortran. Such extensions in Fortran 77, as are available in a few compilers, *should not be used.*.

It should remembered that even efficiency follows from portability as it allows immediate use of the latest technology. The author has encountered applications which have taken so long to port that the target machine has become obsolete before the port was completed.

The treatment of parallelism

It would be inappropriate in a discussion of this kind to ignore the appearance of massively parallel systems. This offers a new class of problem for the numerical scientist, who is really only interested in performance. The central problem is that effective use of parallelism usually requires non-portable changes to the software as there is no standard model of parallelism. However, parallel extensions to both Fortran and C exist in a number of compilers but should be used with discretion. The practice of running existing (and hopefully portable) Fortran and C applications in a *parallel harness* appears to offer the best compromise between portability and performance. This may either be explicit with directives in the code or implicit in the compiler itself, this latter being the most suitable.

Reliability and the Safe(r) Subset

The nature of language standardization, whereby a lot of people argue about things for several years and then attempt to agree enough to write it all up, guarantees that languages are full of compromises, ("you vote for my feature and I' llvote for yours ..."). Many of these unfortunately affect the

reliability of applications written in those languages. What might seem a great idea to a compiler writer could ultimately lead to a catastrophe in a safety critical environment in inexperienced hands, (and sometimes in experienced hands !).

A natural and essential development of this is that users identify dangerous bits of a language and effectively ostracize them using automatically enforced programming standards. Users should share such knowledge. Only by this professional approach can the prospective user protect their software investment and their own users. This leads to the notion of a safe(r) subset. Such subsets are not static. As more unsafe features come to light, the safe(r) subset should be contracted to exclude them. If the resulting process leads ultimately to an empty subset, it is probably time to throw away the language ! The author cannot over-emphasize the importance of this process in the overall improvement of software quality. It is simply negligent to use features which are known to compromise reliability. If any reader disagrees with this point and thinks that software is O.K., they should read the "Risks to the Public" section which appears quarterly in the Software Engineering Notes of the A.C.M.

As a service to the numerical scientists generally, the Appendix contains a list of features which have been contributed by many Fortran 77 users around the world and which should be excluded from normal programming practice on the grounds that they have failed at least once and often repeatedly.

Conclusions

As will be obvious from the above, there is no natural choice of language for the scientific user. The growing trend towards hybrid systems seems essentially healthy, although it places greater demands for multi-linguality on programmers. This should not be considered a bad thing. Furthermore, the importance of the safe(r) subset should not be under-estimated. Use of this concept can reduce maintenance costs and improve reliability significantly. If C is to be used, it should be written compatibly with C++. Fortran 90 features should be used with great care and there may be a better alternative in C++.

The author would like to invite Fortran, C or C++ users to send any details of features which have caused them problems to him, care of this journal. This will help him update the safe(r) subsets of these languages periodically.

Acknowledgements

A very large number of individuals and companies have contributed to the knowledge base of potentially unsafe Fortran 77 features presented in this paper, although the opinions expressed and any inaccuracies made in reporting are the author's. In alphabetical order, the organisations include: APT Netherlands, Aerospatiale, Atomic Energy Authority (U.K.), Berkeley Nuclear Laboratories, British Aerospace, British Gas, BP, ComputerVision, Cray Research, Dynamic Graphics, Electricite de France, European Centre for Medium Range Weather forecasting, European Space Agency, FEGS, GECO, General Dynamics, Halliburton Geophysical Services, Ministry of Defence, McDonnell-Douglas, Meteorological Office (U.K.), Mobil, N.A.S.A., Ordnance Survey, Philips, Robertson ERC, SNEA(P), SPSS inc., Scott-Pickford, Shell, Simon-Horizon, Studsvik of America, Texaco, University of Bath, University of Kent Computing Laboratory, Unocal, Westland Helicopters and Zycor. The author apologises to any he might have missed.

Appendix A: A safe(r) subset of Fortran 77, (ANSI X3.9-1978).

This Appendix defines a safe(r) subset of Fortran 77 resulting from user experiences distilled from a large number of companies around the world by defining those features *to avoid*. In this context, unsafe implies those features which either are unsafe to use or are safe in themselves but tend to lead to unsafe practices. *It should be emphasised that this list is not simply the author's choice. Each feature below has been reported to the author by a Fortran user and most of them caused financial penalty. As such the list should be considered as a shared knowledge base of unsafe practice. Ignore it at your financial peril !*

1.0 Features which should be avoided

1.1 Unreachable code.

Reduces the readability and therefore maintainability.

1.2 Unreferenced labels.

Confuses readability.

1.3 The EQUIVALENCE statement except with the project manager' s permission.

This statement is responsible for many questionable practices in Fortran giving both reliability and readability problems. Permission should not be given lightly. A really brave manager will unequivocally forbid its use. Some programming standards do precisely this.

1.4 Implicit reliance on SAVE. (This prejudices re-usability).

A particular nasty problem to debug. Some compilers give you SAVE whether you specify it or not. Moving to any machine which implements the ANSI definition from one which SAVE' s by default may lead to particularly nasty run and environment sensitive problems. This is an example of a statically detectable error which is almost impossible to find in a source debugger at run-time.

1.5 The computed GOTO except with the project manager' s permission.

Often used for efficiency reasons when not justified. Efficiency should *never* precede clarity as a programming goal. The motto is "tune it when you can read it".

1.6 Any Hollerith.

This is non-ANSI, error-prone and difficult to manipulate.

1.7 Non-generic intrinsics.

Use generic intrinsics only on safety grounds. For example, use REAL() instead of FLOAT().

1.8 Use of the ENTRY statement.

This statement is responsible for unpredictable behaviour in a number of compilers. For example, the relationship between dummy arguments specified in the SUBROUTINE or FUNCTION statement and in the ENTRY statements leads to a number of dangerous practices which often defeat even symbolic debuggers.

1.9 BN and BZ descriptors in FORMAT statements.

These reduce the reliability of user input.

1.10 Mixing the number of array dimensions in calling sequences.

Although commonly done, it is poor practice to mix array dimensions and can easily lead to improper access of n-dimensional arrays. It also inhibits any possibility of array-bound checking which may be part of the machine' s environment. Unfortunately this practice is very widespread in Fortran code.

1.11 Use of BLANK=' ZERO' in I/O.

This degrades the reliability of user input. See also 1.9.

1.12 Putting DO loop variables in COMMON.

Forbidden because they can be inadvertently changed or even lead to bugs in some optimising compilers.

1.13 Declarations like REAL R(1)

An old-fashioned practice which is frequently abused and leads almost immediately to array-bound violations whether planned or not. Array-bound violations are responsible for a significant number of bugs in Fortran.

1.14 Passing an actual argument more than once in a calling sequence.

Causes reliability problems in some compilers especially if one

of the arguments is an output argument.

1.15 A main program without a PROGRAM statement.

Use of the program statement allows a programmer to give a module a name avoiding system defaults such as main and potential link clashes.

1.16 Undeclared variables.

Variables must be explicitly declared and their function described with suitable comment. Not declaring variables is actually forbidden in C and C++.

1.17 The IMPLICIT statement.

Implicit declaration is too sweeping unless it is one of the non-standard versions such as IMPLICIT NONE or IMPLICIT UNDEFINED.

1.18 Labelling any other statement but FORMAT or CONTINUE.

Stylistically it is poor practice to label executable statements as inserting code may change the logic, for example, if the target of a DO loop is an executable statement. This latter practice is also obsolescent in Fortran 90.

1.19 The DIMENSION statement.

It is redundant and on some machines improperly implemented. Use REAL etc. instead.

1.20 READ or WRITE statements without an IOSTAT clause.

All READ and WRITE statements should have an error status requested **and** tested for error-occurrence.

1.21 SAVE in a main program.

It merely clutters and achieves nothing.

1.22 All referenced subroutines or functions must be declared as

EXTERNAL. All EXTERNALS must be used.

Unless EXTERNAL is used, names can collide surprisingly often with compiler supplied non-standard intrinsics with strange results which are difficult to detect. Unused EXTERNALS cause link problems with some machines, leading to spurious unresolved external references.

1.23 Blank COMMON.

Use of blank COMMON can conflict with 3rd. party packages which also use it in many strange ways. Also the rules are different for blank COMMON than for named COMMON.

1.24 Named COMMON except with the project manager' s permission.

COMMON is a dangerous statement. It is contrary to modern information hiding techniques and used freely, can rapidly destroy the maintainability of a package. The author has bitter, personal experience of this ! Some company' ssafety-critical standards for Fortran explicitly forbid its use.

1.25 Use of BACKSPACE, ENDFILE, REWIND, OPEN, INQUIRE and CLOSE.

Existing routines for each of these actions should be designed and must always be used. Many portability problems arise from their explicit use, for example, the position of the file after an OPEN is not defined. It could be at the beginning or the end of the file. The OPEN should always therefore be followed by a REWIND, which has no effect if the file is already positioned at the beginning. OPEN and INQUIRE cause many portability problems, especially with sequential files.

1.26 DO loops using non-INTEGER variables.

The loop may execute a different number of times on some machines due to round-off differences. This practice is obsolsecent in Fortran 90.

1.27 Logical comparison of non-INTEGERS.

Existing routines for this should be designed which understand the granularity of the floating point arithmetic on each machine to which they are ported and must always be used. Many portability problems arise from it's explicit use. The author has personal experience whereby a single comparison of two reals for **in**equality executed occasionally in a 70,000 line program caused a very expensive portability problem.

1.28 Any initialisation of COMMON variables or dummy arguments is forbidden inside a FUNCTION, (possibility of side-effects).

Expression evaluation order is not defined for Fortran. If an expression contains a function which affects other variables in the expression, the answer may be different on different machines. Such problems are exceedingly difficult to debug.

1.29 Use of explicit unit numbers in I/O statements.

Existing routines to manipulate these should be designed and must always be used. Many portability problems arise from their explicit use. The ANSI standard only requires them to be non-negative. What they are connected to differs wildly from machine to machine. Don't be surprised if your output comes out on a FAX machine !

1.30 CHARACTER*(N) where N>255.

A number of compilers do not support character elements longer than 255 characters.

1.31 FORMAT repeat counts > 255.

A number of compilers do not support FORMAT repeat counts of more than 255.

1.32 COMMON blocks called EXIT.

On one or two machines, this can cause a program to halt unexpectedly.

1.33 Comparison of strings by other than the LLE functions.

Only a restricted collating sequence is defined by the ANSI standard. The above functions guarantee portability of comparison.

1.34 Using the same character variable on both sides of an assignment.

If character positions overlap, this is actually forbidden by the standard but some compilers allow it and others don' t. It should simply be avoided. The restriction has been removed in Fortran 90.

1.35 Tab to a continuation line.

Tabs are not part of the ANSI Fortran definition. They are however easily removable if used only to code lines and for indentation. If they are also used for continuation (like the VAX for example), it means they become *syntactic* and if your compiler does not support them, removing them is non-trivial.

1.36 Use of PAUSE

An obsolescent feature with essentially undefined behaviour.

1.37 Use of ' /' or ' !' in a string initialised by DATA.

Some compilers have actually complained at this !

1.38 Using variables in PARAMETER, COMMON or array dimensions without typing them explicitly before such use.

e.g.

PARAMETER (R=3)

INTEGER R

Some compilers get it wrong.

1.39 Use of CHAR or ICHAR.

These depend on the character set of the host. Best to map onto ASCII using wrapper functions, but almost always safe today.

1.40 Use of ASSIGN or assigned GOTO.

An obsolescent feature legendary for producing unreadable code.

1.41 Use of Arithmetic IF.

An obsolescent feature legendary for producing unreadable code.

1.42 Non-CONTINUE DO termination.

An obsolescent feature which makes enhancement more difficult.

1.43 Shared DO termination for nested DO loops.

An obsolescent feature which makes enhancement more difficult.

1.44 Alternate RETURN.

An obsolescent feature which can easily produce unreadable code.

1.45 Use of Fortran keywords or intrinsic names as identifier names.

Keywords may be reserved in future Fortran standards. The practice also confuses readability, for example,

IF (IF(CALL)) STOP=2

Some people delight in this sort of thing. Such people do not take programming seriously.

1.46 Use of the INTRINSIC statement.

The ANSI standard is particularly complex for this statement with many exceptions. Avoid.

1.47 Use of END= or ERR= in I/O statements. (IOSTAT should be used instead).

Using END= and ERR= with associated jumps leads to unstructured and therefore less readable code.

1.48 Declaring and not using variables.

This just confuses readability and therefore maintainability.

1.49 Using COMMON block names as general identifiers, where use of COMMON has been approved.

This practice confuses readability and unfortunately, compilers from time to time.

1.50 Using variables without initialising them.

Reliance on the machine to zero memory for you before running is not portable. It also produces unreliable effects if character strings are initialised to zero, (rather than blank). Always initialise variables explicitly.

1.51 Use of manufacturer specific utilities unless specifically approved by the project manager.

This simply reduces portability, in some cases pathologically.

1.52 Use of non-significant blanks or continuation lines within user-supplied identifiers.

This leads both to poor readability and to a certain class of error when lists are parsed, (it may have been a missing comma).

1.53 Use of continuation lines in strings.

It is not clear if blank-padding to the end of each partial line is required or not.

1.54 Passing COMMON block variables through COMMON *and* through a calling sequence.

This practice is both illegal and unsafe as it may confuse optimising compilers and in some compilers simply not work. It is a very common error.

2.0 Other undesirable features

2.1 An IF..ELSEIF block IF with no ELSE.

This produces a logically incomplete structure whose behaviour may change if the external environment changes. A frequent source of "unexpected software functionality".

2.2 DATA statements within subroutines or functions.

These can lead to non-reusability and therefore higher maintenance and development costs. If constants are to be initialised, use PARAMETER.

2.3 DO loop variables passed as dummy arguments.

See also section 1.12.

2.4 Equivalencing any arrays other than at their base, even if use of EQUIVALENCE has been approved.

Some machines still have alignment problems and also modern RISC platforms rely on good alignment for efficiency. So at best, it will be slow and at worst, it will be wrong.

2.5 Equivalencing any variable with COMMON, even if use of EQUIVALENCE and COMMON has been approved.

This rapidly leads to unreadable code.

2.6 Type conversions using the default rules, either in DATA or assignment statements.

Type conversions should be performed by the programmer - state what you mean. For example:

R = I Wrong

R = REAL(I) Right

2.7 Use of mixed-type arithmetic in expressions.

See 2.6 above.

2.8 Use of precedence in any kind of expression.

Parenthesize to show what you mean. Although Fortran precedence is relatively simple compared with C which has 15 levels of precedence, it is still easy to get it wrong.

2.9 Concatenated exponentiation without parenthesizing, e.g. a**b**c.

People too often forget what this means. Exponentiation associates from the *right.*

2.10 Calling sequence matching.

Make sure that calling sequence arguments match in type number *and* direction. Inconsistencies here are responsible for many unreliability problems in Fortran.

The above concerned only legitimate Fortran 77 features. However, there are a large number of commonly occurring extensions to this standard which could be considered. This fall into various categories such as widely used good ideas, widely-used bad ideas, sparsely-used good ideas and sparsely-used bad ideas. In principle, all should be avoided, but the widely-used good ideas are almost certainly a net asset in the often conflicting struggle between portability and engineering quality. The following, although not a complete list, can be considered on the grounds that it is usually quite simple to remove them automatically.

3.0 Good extensions which are widely supported

3.1 Up to 31 character identifier names including the underscore ' _' , but **not** the ' $' .

3.2 DO terminated with ENDDO.

3.3 IMPLICIT NONE.

3.4 Use of lower-case letters.

3.5 The INCLUDE statement in one of its numerous forms. The Fortran 90 committee tried to leave this out in favour of an

arcane alternative but gave way to public pressure eventually, so a form of it is now part of Fortran 90.

Abbreviations

A.C.M.	Association for Computing Machinery
A.N.S.I.	American National Standards Institute
G.U.I.	Graphical User Interface
I.S.O.	International Standards Organisation
P.O.S.C.	Petrotechnical Open Standards Consortium XXXX.

References

Feldman S.I., Gay D.M., Maimone M.W., Schryer N.L. (1990), "A Fortran-to-C converter", Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 07974

Jones D., (1991) "C++ or C9X", *.EXE*, Vol 5., no. 8, February 1991.