Fortran 202X Feature Survey Results – Final

Steve Lionel

February 6. 2018

Introduction

WG5 published a survey to gather input from the user community on possible new features in Fortran 202X (also referred to as Fortran 2020.) N2142 contained an earlier snapshot of responses as of October 8, 2017. The completed survey, closed on January 23, 2018, gathered 137 responses.

The format of the survey is necessarily constrained. A deliberate choice was made to lead off with a "ranked choice" question naming several suggestions previously received, that required the user to rank them in order of desirability. An open option was provided for the user to supply a suggestion not listed, though these would naturally not get voted on by other users. Given the limited new feature set of F202X, WG5 thought it would be helpful to gauge community support for known ideas, and to collect suggestions WG5 might not have seen before.

The following sections list the suggestions in ranked order (as determined by SurveyMonkey), with the score (a higher score indicates that more users rated that suggestion higher than others), with individual comments. User suggestions are then listed, followed by open-ended "comments from the community". The users were asked, "For the feature you ranked highest, how would it help in development of your applications? Have you used this feature in applications written in other languages?"

The survey asked for names and email addresses – most users opted to provide these – but they are not included in this document. Some responses have the survey response number in parentheses – often the user provided more detail that is included here. Contact the author if you want to see additional detail.

Generic Programming or Templates

Score: 6.21

- Reduce the tedious generation of almost identical code for various datatypes. Reducing scope for errors by not having different versions of the same routine for different types hidden behind an interface.
- Have made use of templating in C++. Should be possible to make Fortran equivalent lighter and easier to use (if less powerful).
- I've used the facility in C++, C# and Java. The first example Jane and I use in our teaching (and book) is based on sorting, and we provide 3 real and 4 integer implementations of the same sort algorithm. The second example implements a generic statistics module for 32, 64 and 128 bit reals. If a template facility existed in Fortran we would add generic container examples to our books and teaching.

- I went to the extent of writing my own SED/AWK preprocessor to handle templated source code. Obviously, I would prefer to have a standard solution, but it has to be really usable, and it has to play nicely with my OO code.
- Generic programming and templates will simplify much the structure of the code, avoiding redundancy when the same routine is used for different kinds of variables. I currently use this feature in C++ programming
- 1. No needs to write subroutines for different kinds, fixed size for vectors inside types (parmeterized derived types, implemented in some compilers is terrible concerning execution time). 2. Yes.
- I would regard templates to be a boon simply to keep repeated code for different types to a minimum. Some of the suggestions that have appeared on clf are interesting for their syntactic simplicity. In doing a little bit of research on templates, I kept coming across the remarks that D templates are syntactically sweeter than C++. On reading the D template Tutorial by P Sigaud (206 pages!), I think that I will start eating lemons.... KISS must rule.

  My preference would be something along the lines of a generic type declaration to mirror/supplement the existing generic language features. Once SELECT TYPE is rolled in, a highly flexible language extension would have been enabled, which fits in nicely with existing Fortran features. By not insisting on dynamic dispatch of instances of templated procedures, implementation would be straightforward too.

  It is time that iterators made their way into Fortran. Not only could these provide syntactic advantages and improved flexibility compared with DO but then the way would be open to the provision of a standard library like STL. Again, KISS must be kept in mind!
- Template like feature can be used with subroutine in F95, but with OOP coming in, I would like polymorphic templates for many applications. I am not an OOP man, so more comment may not be possible. F95 had many indirect ways of doing things, but making cleaner fast methods would help.
- 1. It could save a lot of typing 2. I use it in C++ and Java programming
- Many times, I'm writing the same routine for different types, or different kinds. I used templates and STL containers in C++. I would like to see them in Fortran too. A sort of move semantic should then be also implemented.
- Quite often, we have to write multiple versions of each wrapper wherein the only difference between the versions is in the type of the first argument. All else in the procedure remains the same.
- Fortran really needs support for generic programming and generic data structures and algorithms. Something like C++ templates and the standard template library would be ideal, whether provided as intrinsic (compiler-provided) modules or as user-written code.

  Users should not need to implement common data structures like linked lists or binary trees themselves, then have to cook up a way (invariably using nonstandard preprocessing and/or abuse of the IMPLICIT statement) to avoid duplicating code for each type they want to use with the data structure

- It would allow the community to implement a library of containers, like C++'s STL. Right now something like that can only be done by abusing the C-preprocessor (see for example https://github.com/robertrueger/ftl), which is not exactly pretty and requires manual template instantiation. Even people who never write templates themselves would benefit a lot from these template libraries, so I think it is really the single most important thing missing in Fortran.
- IMHO, the most annoying missing feature in Fortran is generic containers, i.e., the tools to construct and use lists, stacks, trees, etc. without reference to the type of the elements, but nevertheless type-safe at all stages. A significant fraction of our code, regarding data management at any level, could be written more clearly, readable, and concise. To a lesser extent, this applies also to generic algorithms such as sorting.

  Partly for that reason, we use OCaml for algorithms that require formal, non-numerical manipulation of data structures. OCaml is a strongly typed functional language that supports type-parameterized types (such as "list of type x") and type-parameterized modules (a.k.a. functors) in a very straightforward and entirely type-safe way. If such facilities had been available in Fortran, coding such algorithms in Fortran - not just the numerical parts - would have been a viable alternative.
- Writing generic code is painful and tedious. If it were easier, I would actually bother writing reusable modules for lists, trees, etc. to use in my projects. Currently, it's too much trouble.
- c++ template meta programming
- Templates remove lots of duplicate coding. I used this extensively in C++.
- It is a feature that is often asked for (judiging from the number of reads of an old Fortran Forum article of mine on the subject. I should really re-write it, as it is antiquated, even without formal templates in F2020). I have seen it used in languages like Java and C#, especially with respect to data containers like lists. It would help with libraries in various precisions.
- Adding generic programming would significantly simplify the creation of containers. At present, a container has to be written for every type it is desired to contain. This means large sections of code have to be repeated, e.g. a numeric linked list has to be coded for types integer, real(dp) and complex(dp) at a minimum, not to mention writing a linked list of user defined types.

  The ability to define types (in C++ syntax) as container<T>, rather than container_integer, container_real etc. would reduce this to a single definition for each container.

  I have used these features extensively in C++, although I feel that Rust does a better job of implementing them.
- Easily making overloaded double and single types of functions without duplicating code. Being able to have functions like "optimise(f, anyobject)" which will work by calling f(x, anyobject) where anyobject is any type containing extra data used by f (without requiring tiresome redundant select type.. statements inside f).
- A lot of algorithms are applicable to a wide variety of data types or kinds of a particular data type. Using current features, a programmer has to write the procedure implementing the algorithm over again for each data type and kind that the algorithm is implemented in. This requires a tremendous amount of wasted time and effort and many opportunities for making mistakes in one or more implementations of the same algorithm. Implementing procedure

templates would greatly reduce or eliminate this duplication and opportunity for defects. Therefore the benefit-to-cost ratio to application developers of this feature is tremendous.

- I have written many containers to provide e.g. lists, sets and maps. To do this and avoid large amounts of copied code I have used the c-preprocessor so that they can be used with integer(2,4,8) read(4,8). This really seems to me to be a deficiency in the language. I don't think that the extensive way C++ has taken templated is the right way but some form of template functionality is a must. It would also enable STL to be incorporated into Fortran, after all it wasn't originally created for C++ it was just incorporated by it. Containers (including iterators) are a natural way to program and it would be really nice if Fortran supported these paradigms directly rather than through to cpp.
- Generic programming and/or templates is something that seems to get implemented in every Fortran project through some other language (e.g. Python, Ruby or even IDL!). It is very often the case that we need to perform exactly the same operation on arrays of rank 1, 2, 3, etc., or arrays of different types -- e.g. I/O using an external library. The only way to avoid writing the same thing over and over again is through some nature of templating. The current universal polymorphic stuff does come in handy, but the requirement for type guards makes the code a) much uglier and b) much less useful. I use this all the time in other languages, either explicitly using templates as in C++, or "duck-typing" in Python.
- Reduce duplicate and boilerplate code leading to increased maintainability and flexibility. This feature has proved very useful in applications written in other languages.
- It will let me and others implement resuable code. It will let us write a generic "collections" library, for example.
- Generic programming or templates a'la C++ will definitely help me to better express some of the abstract data types in my projects. For example static polymorphism of templated containers like std::vector and std::valarray in C++ where the same class interface(procedure signatures) is presented to user class.
- Generics would help avoid writing redundant code that differ only by argument type (i.e. single- and double precision versions of the same function; data structures such as hash tables, trees, and linked lists)
- An FTL like the C++ STL would be quite useful. While I agree that static programming is easier to understand, generic programming can be more flexible.
- The code I work on makes extensive use the C pre-processor in places to create generic versions of Fortran procedures via macros. Or we just write the code in C++... Unfortunately, the current Parameterized Derived Type facility, even if it were implemented by a majority of compilers, is insufficient for our use.
- It would allow reuse of code. For example, a generic function for resizing an array would be very useful. I use generic programming extensively in OCaml via parametrized modules.
- A generic list implementation is really lacking and this is not fully solved with unlimited polymorphism
- Generic lists facilitate the book-keeping of a lot of applications.
- I have several procedures in various kind versions. Templates may allow one procedure for various kinds. I use this feature frequently in C++
- It would dramatically simplify the programming of generic algorithms. I currently have to use a preprocessor with many shortcomings.

- Reusing the same code for different data types. Parameterized modules are fine for me.
- We already implement template programming on our own. This would eliminate our need to do this by hand on our own, with complex scripting and build mechanisms. Having this built into the language would make this much easier.
- I could write generic containers, like lists, once and not have to keep writing nearly identical code, or resort to ugly use of source pre-processing with includes and macros to automate the source code generation. I've worked some in C++ and found this to be incredibly useful.
- Generic programming is the fundamental key to achieve a real Abstract Calculus by means of which mathematical/physical programs achieve high conciseness, clearness, easy maintainability and flexibility allowing to develop generic libraries the API of which strongly resemble high-level math expression. I used generic programming in Python.
- Writing the 10000th routine for sort, linked lists, ... is tedious and error-prone - reinvented wheels are all too often square. But beware the complexity - it is easy to mess up (as C++ did) and create a Turing-complete language by accident.
- I have used generic programming when working on Java codes. It is definitely a very useful feature that allows to avoid a lot of code duplication and simplifies code structure a lot. I would be hesitant to follow C++ example, though, templates seem to be much more difficult to use correctly than Java generics.

Automatic Allocation on READ into ALLOCATABLE character or array

Score: 5.04

- Simplify coding.
- I prefer to use allocatable arrays when possible. This would make my code cleaner and more readable, since now I need to pre-allocate arrays.
- The "Automatic allocation on READ into ALLOCATABLE character/array" sounds like a capability standard in many scripting languages. Getting data into a Fortran program reliably via current standard mechanisms can be onerous.
- It would make it easier to write code.
- Get rid of tons of clutter and buffer variables in parts of the code that are not performance-critical anyways: Reading human-generated input files etc.
- Automatic allocation would allow easier file processing. I am using this mainly in Python for file I/O.
- Reading file header of unknown content. C# for example
- Hello, I use templates every day in my code (C++ and Java), I remember when they were introduced in java 1.5, it was a really big change. Actually, it is preventing me from developing some projects in Fortran, it will simply take too long to do it without them and the code wouldn't be "correct" from a modern software development point of view.
- Use of allocatable character simplifies string handling and makes it more reliable, but it doesn't work in many situations. Allowing automatic allocation in READ would be a good step; it would simplify many of my programs.

- I have written work-arounds for the lack of this feature since allocatable arrays were first available in Fortran 90. It would be very nice to not have to read the input line into a character buffer, parse the string, count the number of elements, allocate the array (or character) and then use an internal read statement to proceed. Not a life changing event, but a nice cleanup of a "missing" feature.

Block Oriented or Structured Exceptions

Score: 4.91

- Exception handling is still a weakness of modern Fortran. Whilst there is some handling of IEEE numerical conditions, a more general approach based upon throw/catch or similar would be very helpful. It is common in other languages.
- It helps in handling exceptions. I use the feature in java
- The lack of a comprehensive approach to exception handling is probably one of the most significant shortcomings of current Fortran. I've used this in C# and object Pascal / Delphi.
- My primary reason for wanting exceptions is to support unit testing. I use pFUnit for unit testing my Fortran code, and I believe that having an exception mechanism in Fortran would make it much easier to write unit tests of expected error conditions.

Unsigned INTEGER

Score: 4.48

- Unsigned integer data type is implemented almost modern languages. It helps us to count, compare numbers. So i want Fortran to have the feature.
- C interoperability - we make heavy use of mixed language application development. Have used in other languages (notably C!).
- To improve interoperability with other languages, Fortran needs an unsigned integer type. The current situation is confusing and misleading at best.
- Would help with the development of various code features involving bit manipulation and in particular inter-language features with C (where I have libraries that use unsigned integer bit manipulation)
- implementing algorithms in fortran (network, cryptography, etc) similar to C
- With Window programming often an actual argument must be an unsigned integer. Although, usually, a signed one can be used as well, it adds elegance to it and it is, formal speaking, more correct to have unsigned integers to one’s disposal.
- Reading detector data. Counts are always positive. Need C to read data now.
- Many codes use either MPI-based aborts or STOPs in various routines instead of return error codes and/or handling errors properly. This makes factoring out parts of the code into libraries rather difficult since every error will terminate the executable with little to no possibility in catching these and giving the caller an easy way to wrap things up. Having C++ or Python-style exceptions (which I'm used to) would make it possible to easier catch errors in such cases.

Furthermore, C++ or Python-style exceptions would make it possible to add context to errors without having to resort to directly printing out this context at the time the failure occurs, require the caller of a subroutine/function to do error handling if it makes sense at that point or silently send exceptions to its own caller if there is nothing it can do and finally making refactoring much easier and error handling code much less cluttered. Having the exceptions out-of-band could help with the execution flow since branches for (properly written) error handling won't be part of the normal path anymore. (97)

- Size of 8 bits (same size as default character). For easier processing of picture files. I have used unsigned in another language. (99)
- An unsigned integer doubles the range of the integer for use in array indexing and counting. As the size of memory increases, I have found that sometimes I need to address enormous arrays, and sometimes these arrays would best be 1-D. But without an unsigned integer, half of the range is lost to me, forcing me to e.g. index the array with a negative integer (which is doable but cumbersome) or to artificially make the array mult-dimensional. But even so I may not be able to represent the size of the array because again, half of the integer's range is lost. (116)

Easier Manipulation of NUL-terminated strings

Score: 3.64

- I have used C extensively for string and character manipulation. It would be really helpful if Fortran could support automatic string-length (leading blank suppression) and null-terminated strings to handle important text input and searching programs.
- String manipulation is the most elaborate part in our software (and most cumbersome). It takes too many lines of code to do string manipulation. Using mostly C++ I am used to shorter versions, especially not having to always explicitly putting the length of the string in a larger character array.

Bit Strings

Score: 3.58

- We use many flag fields in our codes, we use integers but this is not ideal as the access to the bits is clunky, output is unhelpful without explicit formatting, problems with respect to the sign bit, wasted space if there are only a few bits used, integers are insufficient if there are more than 64 elements
- One use, amongst others is when I have to store the state of a system, in time, each site of which is described using a logical variable. I currently use LOGICAL(KIND=1). No, do not use any other language for scientific computing.

Conditional Expressions (like C’s ? operator)

Score 3.52

No comments received on this item.

Other suggestions

I've made an attempt to separate these into broad categories, based on my (possibly flawed) understanding of the suggestion.

Standardized Libraries and Additional Intrinsics

- An "official" collection of libraries of routines for the many common non-numeric functions required to create applications would be the biggest possible change in making Fortran a general purpose language I could imagine. More and more I require a programming language that supplies more than just numeric analysis. I think it unlikely the required set of features will be added to the base Fortran language. The many individual attempts to fill the gap fail to be maintained or require expertise in too many domains for a single person to support. Although a few attempts exist to provide group-supported repositories are emerging nothing has emerged anywhere near what other languages have. Fortran needs to make a concerted effort to have a central repository for such things as the oft-reinvented interfaces to SQL, HDF5, JSON, date-time libraries, string manipulation routines, generic lists and dictionaries, expression parsers, regular expressions, the POSIX PXF "system" interface, ... I spend a very large amount of time supporting my own collection of such interfaces and know of many others doing the same.
- A standard library for unit testing, preferably something that supports continuous integration and can work with mainstream IDEs such as Eclipse or Visual Studio. Testing is critical and there are almost no language features or libraries which support unit testing in Fortran. Considering we develop nuclear safety software to the NQA-1 standard, it's frightening how few and poor the tools are in the Fortran ecosystem. Many other language maintainers make testability a major focus (see Rust, Go, Python, Perl, Ruby, Lua, Java, Ada/Spark, Racket, etc., etc.) The applications developed & maintained in Fortran are too important to not have committee-level attention devoted to improving testability.
- Intrinsic sorting procedures.
- A working string class and easier way to manipulate strings. This is a pain in Fortran currently!
- Degree trigonometric functions,
- a subroutine SWAP_ALLOCS that swaps the allocations of allocatable variables,
- Fix the useless SAME_TYPE_AS intrinsic so that it does what its name implies -- comparing type and kind just as SELECT TYPE does -- so that it can be used on polymorphic variables with intrinsic dynamic types.
- Intrinsic hash table, sort, FFT and iFFT
- intrinsic resize(array) that keeps the old data while resizing the array.
- Intrinsic change_case(character) invert, to_lower, to_upper, Title_Style
- Also, I'd love a regex built in to Fortran, but that's a bit farfetched as Fortran isn't exactly built for string handling, and one could implement it as a library.
- Standard collections Particularly in conjunction with generics/templates a standard set of collections would really save effort. In particular a linked list and hash-map. (dictionary) These are commonly used data structures and having to write your own implementation each time you need one is obviously less than ideal. All the normal arguments in favour of libraries pertain.

- A text-handling standard library including case-conversion, regular expressions, and read/write capability for common data formats such as CSV and JSON. Consider the Python standard libraries csv & json and the external library regex (also the Perl Compatible Regular Expression library, PCRE) A substantial amount of our coding goes into processing user input and formatting output.
- Please introduce standard template libraries (STL). I would to be able to use vector and tree classes that are present in other languages such as C++. Introduce type auto that is present in C++ which is very useful for finalizers. Shared Pointers are another great feature present in STL for C++. I have use all of these features in C++ application written with STL.
- Define a non-file based, (and would be nice to also be required to be thread-safe) interface for converting ints/reals to a string and back. Currently to do the conversion requires a read/write statement, however i have personally been hit with the case inside a openmp based code where doing this conversion inside a parallel region caused a crash due to non-thread safe nature. I also find the format confusing and non intuitive. The simplest option would be to extend the int() and real() functions to accept a character as a possible argument with an extra optional argument "format" to do a format conversion. If format is not specified then some sensible default should be assumed. result = int(A,[kind],[format]) where a can be a int, real, complex or character and a string based function: result = str(A,[format],[len]) Where A is a integer, real or complex, format a optional extra argument for specifying the conversion and result is a character long enough to store the result, or as long as an optional integer argument len. Having new functions/subroutines allows us to define that they must be thread-safe (not sure anyone would ever want to depend on using read/write and them not being thread-safe but prevents us form having to alter file reading code). Having an explicitly defined functions to do the conversion would bring fortran more in line with other programming languages like python (which has str(),int(),real() functions that take any object and try to convert to their respective output) and make things easier to understand for users. Function based versions would also make it easier to combine strings with numbers on one line: do i=1,10 read_file('file_'//str(i)//'.txt') end do
- A good set of standard libraries. Most important I would put string manipulation functions and vector/array operations. Automatically resizing arrays would be very useful and prevent writing some boilerplate code to query for the need of reallocation and the actual reallocation. Furthermore, a few more types, like dictionary and linked list, would be helpful. And definitely a standard sorting function (qsort).
- Standard library, like linked list and hash table.

Preprocessor

- a Fortran preprocessor modeled after the C preprocessor
- A standard Fortran pre-processor that includes concatenation and stringify operators, and that will automatically split too-long source lines resulting from macro substitution. I use a macro-based assertion system, and the lack of these features has been a continual thorn in my side.
- It would be nice if FORTRAN itself had pre-processor directives. I currently use -cpp with gfortran, when I put use an #ifdef. But is this some thing under FORTRAN's control?

Syntax Enhancements

- Something trivial for free form: In fixed form, it is possible to group numbers with spaces, as in “Million = 1 000 000” which cannot be done in free form.
- A simple labeled block text feature would allow for generating documentation more easily, creating metadata, help text and code comments, as well as more easily initializing character arrays.
- I would like to see the need to precede a subroutine reference with the CALL statement made optional. Obviously, it is needed for legacy codes but I don't see any reason to require it for modern codes where there is probably an explicit interface available (3)
- a simple generic constructor function for complex values named COMPLEX,
- I often have an array with long expressions for the indices on the LHS of an assignment and I would strongly appreciate a way to refer to the LHS on the RHS or some other way to achieve effect similar to += and *= operators from C.
- Fortran's ability to define inner functions and subroutines contained in a subroutine or function is very welcome, however it is currently restricted to code defined in a module (annoying but not a big problem), to only one level of nesting and must be defined in a contains section. Allowing lightweight syntax for defining local functions that can be passed to other subroutines would be very helpful, especially with generic programming. For example a generic sort routine would be parametrized by the type of the array elements and would accept a comparison function. This is available in all functional languages (OCaml) as well as Python and now C++.
- I've never quite understood why in this day and age we cannot use 'ɑ'instead of 'alpha'. Or, if I liked, name a variable 'jalapeño'. Or even '東京'. Sure, that is facetious, but there are good reasons to actually use 'π' instead of 'pi'. The character exists and is well known. Languages like Go, C#, Java, Haskell, Perl, Python, etc. do allow you to use, say, Greek letters as variables (by following UAX 15, I think?)
- "UNLESS" as an opposite to "IF"/ "UNTIL" as an opposite to "WHILE"; a small thing that improves clarity in some situations
- Extended IF syntax than allows "short circut" evaluation. E.g. IF (logical-expr1) AND (logical-expr2) THEN Where logical-expr2 is not evaluated if logical-expr1 is FALSE.
- New DO loop syntax to be consistent with DO CONCURRENT. E.g. DO LOOP (I = 1:N , J = 1:M)
- It would also be nice to be able to overload more existing keywords, e.g. being able to make a read statement for user-defined types.
- "Proper" Constructors Normally constructors are recursive, starting at the root parent class and bubbling up to the leaf child class. This way each level constructs itself and other classes needn't know how their parents work. They can still call parent methods to aid in their own construction. If there is syntax for passing arguments down the tree, children can influence how their parents are built. Again, without having to know their inner workings. This is common to all OO languages I have come across but for specific reference look at C++ or Java. In particular the current "initialiser" falls short because children need to know how to construct their parents or there needs to be a mess of potentially dangerous "mess with my contents" functions.
- It is a minor issue, but a language like MATLAB has a special way to access the end of an array - "end". One example:

```
end = size(array)
diff = array(2:end) - array(1:end-1)
```

Now we always need an extra variable or use size() inline. It is syntactic sugar, sure, but it might help in cases like this.

- New symbol? As numbers can be positive or negative I am keen to see greater emphasis on the unary minus. The standard result of -2**2 for example is -4 but if the -2 should be a negative number the result should be +4. To save writing parentheses around a negative number can we use a new symbol for the unary minus such as the tilde or tab character (or another). It does not seem possible to interpret -2**2 as +4 just by making the subtraction sign a unary minus because the problem shifts to using parentheses if -(2**2) is needed. But the unary minus should have its own symbol. Then e.g. ¬2**2 would be +4.0 but maybe ~ or ^ is less easily confused with -.
- I would like to see the use of nested parentheses allowed to use {} and [] to clarify longer expressions; used in any sequence but in pairs, obviously. Example: exp(-((x/s)**2)) => exp{-[(x/s)**2]}
- Automatic variable re- assignment like C to become standard. For example, if i is an integer and x is real*8, x=i converts i to real in variable x (and this would be ideally complemented with unsigned integers).
- Make it possible to use % to access methods/members of derived types returned from type-bound functions
- Being able to pass an array slice (e.g. a:b) around would simplify the access of type-bound arrays.
- automatic delegation: Assuming that "foo" contains "component" which has a type-bound procedure "bar", a simple syntax extension could tell the compiler to automatically convert "call foo%bar(args)" to "call foo%component%bar(args)". That is, without a user-written trivial subroutine bound to the type of "foo" that just exists to call foo%component%bar. Actually, this feature is already supported, but it applies to the base type of a type extension, not to a component. If it was available for components, type composition could be exchanged with type inheritance in the implementation, and vice versa, without a syntax change in the caller. In particular, in situations where one would like to emulate multiple inheritance.

Pointer Enhancements

- Allow pointer remapping on contiguous assumed rank arrays. (128)
- real, pointer => complex, pointer. (119)
- POINTER is in my opinion the one feature making INTENT annotation less useful than possible. My suggestion would be to have a CONSTPOINTER that disallows changes through it and prevents passing the CONSTPOINTER (really the pointee) as actual argument to INTENT(inout) or INTENT(out) dummy arguments. (113)
- Pointers that are not explicitly null'ed or associated currently have an undefined state, the compiler should simple force all pointers to be null on declaration

Types and Declarations

- The feature I suggest is to add the ability of implement private deferred type-bound procedures of abstract type in its descendants. We know that in an abstract type, if the type-bound procedures are defined private, then they can only be implemented in descendants who are defined within the defining scope (the module) of the abstract type. If there are descendants outside of the defining module, they cannot reference those private defers of ancient abstract so they can't override those procedures. But in the situation, there are many more descendants inherited from the abstract, putting them together is not a good idea. It will cause the module to be so large and lack of efficiency. There are mechanism that child class can reference private methods of the parent (C# maybe?) , I think its necessary for Fortran to add it to fulfill the OOP strategy. (123)
- Associative arrays (also known as hashes or dictionaries). (116)
- One missing feature that has always puzzled me is PROTECTED attributes in derived types. Very often, some attributes need to be accessed from outside the module that contains the definition of a derived type, but making that attribute PUBLIC is dangerous. For example, changing it improperly could lead to inconsistent internal state, or some checks need to be done on the input value. Currently, in these situations we have to have PRIVATE attributes and write getter procedures that do nothing else than assigning the value of the attribute to an output variable. This is cumbersome and leads to a multitude of small functions that clutter the source code. (Really means protected components - 96)
- Independently, something that I would really like to see is something like interfaces in Java and C#. The name is unfortunate, because INTERFACE is something quite different in Fortran, but they are equivalent to protocols in Swift and Objective-C (and I believe Ada has something similar). This is a way of promising that a type implements some procedures without needing to add parent types in the inheritance tree. This decouples what an object is (its type and parent types) from what it does (the protocols it implements), which is very useful when different types might have similar behaviors in in some respects, while being different to such an extent that it would not make sense to group them in the same inheritance tree. A simple example could be a collection of types, some of which can generate a description of their content and put it in a CHARACTER variable. (96)
- Interfaces as used in Java can provide safe limited multiple inheritance without many drawbacks of full multiple inheritance. One would only provide an interface for a type-bound procedure for which the implementing type must provide a conforming procedure.
- The C#-style Interface is a very convenient way to implement different, yet uniform, kinds of capabilities into a variety of not necessarily related classes. E.g. one could write a SelfDocumenting interface that classes could inherit/implement. In C# a class can inherit from / implement multiple interfaces. It provides a nice flexibility.
- Scope for variables: I wish that we can declare variables in any construct. I mean inside do-loop, if-construct..., So we have not to declare variables that we use only one time inside one construct in the top of the program. This feature is used in c/c++ and many other languages even in Fortran77 by using implicit-type. My suggestion include declaring integers for do-loop. Example: program main implicit none real :: d integer :: i do (integer :: i=1,10) complex :: d if (i <=5) then logical :: d .... else character :: d ... end if end do .... end do end program main I know it exists Block-endBlock but it is annoying.

- Fortran should define either a dedicated list type defined by a LIST keyword or an intrinsic extensible type define by the current TYPE or CLASS keywords. (3)
- A 'fix variable' property. The PARAMETER keyword sets a non-changing variable within the compiled code. However, often there are occasions where one cannot use that as the value of the variable needs to be read in at the start of the program via a namelist or read but once it has been set it must not be changed again. It would be useful to be able to specify this to prevent an accidental change. This would need to be a runtime check rather than a compiler check.
- Please consider embracing, EQUIVALENCE and other storage size & order functionalities within Fortran. Star notation, as in real*4, is simple and clear for machines with byte-size representation--all machines that matter now and in the practical future. Explicit storage size and order functionality is crucial. Fortran is useless without it for applications which can run well on specific machines--and still can with Fortran.
- Parameterized derived types and use of other parameters inside derived types
- enum is too primitive. A more advanced version which treats a named enum as a data type would be useful. Modern C++ supports this.
- Dimension of static array declarable by parameter
- Make 'implicit none' the default. Eliminates a whole layer of potential accidental programming errors at compile time.
- "Proper" Enumerations Having first class enumerated types would be a great boon over knife-and-forking it with an integer and a stack of parameters. Even C's approach of some syntactic sugar around the integer and stack of constants is an improvement. Ideally, though, would be a proper enumeration whereby the compiler enforces holding only those values allowed for the enumeration. Being able to specify "one of these things but nothing else" is extremely useful when implementing state machines and for configuration options.
- Something similar to 'final' variables which are set once at run time but are otherwise treated as constants. The use case for this is semi-static array allocation at run time to allow a code's data space to be configured early in a run while preserving the speed and safety of static allocation. The alternative is a massive amount of manual array allocation and risk of accidentally changing what is intended as a read-only variable.
- Static member or class member. All objects of one class shares a single value for class members
- Ada-like constraints on variables. For example, if a physical correlation function is only defined for temperatures between 200 K and 800 K, an argument could be constrained to throw an exception if an out-of-range temperature (150 K, 2000 K) was supplied.
- parameter-ized types. By this I mean user-defined types that can be parameters. This would also provide enum-type capability. For example: type, parameter :: planets integer :: mercury = 1 integer :: venus = 2 integer :: earth = 3 ... end type planets making the type planets a compile-time constant. Then the code could use planets%venus and could also pass the type around although only intent(in) would be valid, e.g. subroutine mysub(ss) type(planets), intent(in) :: ss ...
- Namespaces in the C++ style or packages in the Java style, something to differentiate two thing of the same name.

- Implicit typing: that is to say, when equating for example integers and characters such as k='a' the k assumes the integer value of the ascii character code for a. Some compilers allow this; gfortran (and presumably "correct" usage) does not. It would greatly help manipulating strings along with suggestions (3) and (4), and perhaps (6) too. I have used such a feature in C extensively, also with reals such as x=i which simplifies the line x=real(i)
- heterogeneous arrays, i.e., arrays of objects with properties varying element by element (length of allocatable strings, type of polymorphic objects). Currently, this is easy to emulate, but one has to define a wrapper type in either case and make use of extra component % syntax to access elements. This appears redundant, obscures code clarity, and needlessly introduces a source for errors. Although we now have allocatable strings, string handling without heterogeneous arrays is (still) a most annoying deficiency of Fortran.
- More built in containers:
  - * Set
  - * Vector (as in automatically growing array)
  - * Queue

Procedures

- Intent(none) for dummy variables
- Require explicit interfaces be available for all procedure calls. Eliminates another whole layer of potential accidental programming errors at compile time.
- The ability to pass read-only variables out of subroutines or to put them in derived data types, like C or C++'s const feature. This is important to prevent accidental corruption of data structures. Currently the only way to do this kind of thing would be to encapsulate the data as a private element of a derived type and provide accessor methods, making it impossible to use with other code that just expects arrays.
- We should be considering lambdas, similar to Python.
- Default values for optional arguments.
- Another suggestion would be unlimited nested functions (I mean function inside a function inside a function...). Not that it's used very often, but the current limitation looks awkward, and sometimes it would help to modularize code, and maybe also to optimize.
- Better handling of optional arguments. (Default values)
- Assumed type arguments to subroutines with explicit interfaces, just like assumed size and assumed shape.
- NOSAVE attribute that makes it possible to initialise a variable without giving it the SAVE attribute. E.g. SUBROUTINE SUB INTEGER, NOSAVE :: N = 10 The integer N will be initialised at each entry of the subroutine and the variable does not have the SAVE attribute.
- A means of supplying a default value for an OPTIONAL dummy argument.
- an alternative to alternate returns. There are situations where these are by far the best option for handling error conditions: cluttering the code with logicals or error codes is not necessarily a good thing. Since the facility os now obsolescent it would be good to have a real altenrative. This might be what's meant by 'Structured Exceptions' above, but if it isn't it should definitely be added to the list.
- Duck typing of derived types (available in many modern languages), i.e. in some way a subroutine declares arguments of a type that has specific methods and any type that has

those methods will satisfy the interface, no need for artificial inheritance trees. Ideally remove the need for select type.

Modules

- Also, in my opinion the fact that USE-association does not document whether associated objects will be manipulated or accessed in read-only-fashion is a short-coming for large projects. (113)
- My number one suggestion is improved support for object-oriented programming via EXTENSIBLE MODULES that allow improved scope. Arrangements for data beyond the current ones of private and public with a 3rd option, INTERNAL! I will put together a separate communication on this. A companion feature to include is SCOPED enumerations ala C++.
- I would also like to see some thought given to defining a portable module format (maybe in some markup language) that could be generated in addition to a compiler specific format. This would enable libraries (say a plot library) to generate the portable format with one compiler (say GCC) and users could then specify the compiler look for the portable module and compile it into its native module format prior to any USE association. This would greatly enhance the use of these types of libraries and eliminate the need to maintain a different version for each different compiler a user might have on his/her system. (3)
- Currently it is not easy to distribute libraries. It will be great that mod files where compiler independent or some other way to distribute libraries.
- EXCEPT clause in the USE statement (opposite of ONLY)

Operators and Expressions

- Is it possible to add to the next versions the additional logical operators
  - - To do short circuit, like || and && in C/C++ : ex: .and_sc. / .or_sc.
  - - To do parallel computations of operator's expression, ex: .and_par. / .or_par.
- I would also like to see an "almost equal" logical operator for comparison of floating point numbers. (3)
- Short-circuit logical operators.
- Introduce alternative short-circuit boolean operators, like ||, && etc.Many redundant "if" at the moment to avoid errors on things like this: "if (associated(P) .and. P==...)"
- Unary minus is higher precedence than ** Example -6**2 is +36 (if compiler allowed integer power)

I/O

- coarray IO. I know it was discussed before, but I'd like to discuss it again as part of the F2020 process. Is MPI/IO (and libraries built on top of it) still the best advice? Are there any HPC IO developments happening already or are expected to happen, which a new parallel IO (coarray IO) feature could eventually harness? A direct access file with shared access from multiple images has been implemented in Cray:
  - https://pubs.cray.com/content/S-3901/8.5/cray-fortran-reference-manual-85/enhanced-io-using-the-assign-environment
  - however, the performance is a different issue.

    - Is it even a good idea to write to a single massive file at exascale? Perhaps the expectation is that the file systems will evolve to make IO from each image efficient? etc. etc.
- I suggest a routine to read a single keystroke from the keyboard (standard input). This is easy to do in other languages like C, but impossible in a portable way in Fortran
- Fortran is very limited in command line environments by not supporting the ability to read and write to streams using stdin and stdout. Piped I/O is a requirement for building applications in *nix environments. Have used it in every other language I use (C, C++, perl, shells, python, Julia, ...) except for javascript/PHP. (102)
- I often want to print 0 before the decimal point if f0.n format would put no digits there. My workaround is a character function f0(x,n,s) to simulate it; x is the real value to write, n is as in f0.n, s is a string that contains 'LZ' if the leading zero is wanted, and SS or SP if one of those existing format options is wanted. See http://homepages.ecs.vuw.ac.nz/~harper/fortranstuff.shtml under the heading. . Fortran 95/2003 module with test programs to simulate F0.n format with various options. (106)
- Optional XML format in NAMELIST. I use namelist in configuration files. Supporting XML as an alternative will make the feature compatible with common practice. No need to support full XML: just a subset that offer the same functionality as the current NAMELIST would suffice.
- read(select_explicit_delimiter) so I can pick something weird if I want
- Adding a specifier to the READ statement that allows a specific delimiter (e.g., tab characte) to be specified for list-directed input, bypassing the baroque rules for list-directed input (e.g., "/" character being interpreted as end of record) would be most useful for processing real-world text file formats.
- I've submitted a separate proposal (FLUSH statement SYNC= specifier) to a couple Fortran standards committee members for consideration.
- PROMPT= keyword for "READ(*," statement. Example: OpenVMS DCL READ /PROMPT keyword
- Support for pipes for inter-process communication. E.g. read compressed file like this: OPEN(NEWUNIT=U,PIPE='gunzip -c file.gz',ACTION='READ')
- general filesystem (c++17 std::experimental::filesystem)

Character/Strings

- Also would like strings to be arbitrary length so that several variable names (for example) can be used in an array e.g. parameters(10)=(/'vbe','nf','ikf',...) and not having to define equal sized strings, so that string comparisons can be made easier.
- Also would like trim(adjustr(astring)) to truncate the string not keep the original length just padded. All these issues can be worked around using character(1) arrays but is cumbersome compared with C, which is very good on string handling.
- Formalise the full ASCII character codes (0-255), not just the old sub-set. Some compilers allow an integer definition tab=9 and then printing tab using 'a1' format others don't. This also alludes to the comments above about implicit typing when moving from reals to integers or integers to characters etc. This is not to say that Fortran cannot be strongly typed just that it picks up C's implicit typing capability.

- Easier and better handling of strings. Having arrays of strings with different lengths etc.
- Language level regular expression support.
- more convenient string manipulation (c++ std::string)

Coarray Enhancements

- I would like to suggest a very simple coarray-related extension to the Fortran 2008 language: A build-in counter to track the execution segments, just counting locally on each coarray image (i.e. the number of times the SYNC MEMORY statement gets executed - explicitly as well as implicitly -): Something like a THIS_SEGMENT intrinsic. I did develop such a counter myself, easily in less than 5 minutes using few lines of F2008 syntax. (The working code can be found here, see STEP 1 and the 161125_src subfolder: https://github.com/MichaelSiehl/Using-Atomic-Subroutines-and-Sync-Memory-to-Restore-Segment-Ordering). It works, but the disadvantage is that the whole application can only make use of a single SYNC MEMORY statement (not really a disadvantage) and no other build-in synchronization method can be used (that is a minor disadvantage). The main disadvantage of the user-defined implementation of such a segment counter is that it does prevent independent development of the distinct parts of a parallel application (e.g. independent coarray-based library development) which is my main reason for demanding such a feature: All parts of a parallel application must use the same user-defined segment counter. A build-in segment counter could allow largely independent development of the distinct parts of a coarray-based parallel application.

  Practical use for a segment counter: The practical use of such a segment counter is strongly related to the use of SYNC MEMORY and atomic subroutines and thus, also to the implementation of customized synchronization procedures. In practice, tracking the execution segments on the images is required for checking and for restoring the segment order among coarray images. A simple but working example program to restore segment ordering among a number of coarray images, using Fortran 2008 source code, can be found here: https://github.com/MichaelSiehl/Atomic_Subroutines--How_the_Parallel_Codes_may_look_like--Part_1 . Here, the segment order among the images gets restored just by executing the SYNC IMAGES statement the required times, like this: ... ! restore the segment order (among the involved images) for this image: do intCount = 1, intNumberOfSyncMemoryStatementsToExecute ! call OOOPimsc_subSyncMemory (Object_CA) ! execute sync memory ! end do ...

  The Fortran 95 base language together with (only a small subset of) the Fortran 2008 coarray-related language features might be already the promise for an highly flexible and nearly unlimited parallel programming. Applying some simple programming 'tricks' does already allow to make safe use of the SYNC MEMORY statement together with atomic subroutines to implement customized synchronizations programmed as procedures. Moreover, due to a simple programming technique, we can easily transmit two (limited-size) integer values within a single call to an atomic subroutine. This allows, at the same time, to transmit and to synchronize the remote transfer of an atomic (limited-size) integer value. We should be able to use this to process (integer) array data atomically, just by synchronizing each array element separately. And with the ability to process array data atomically, even with unordered (customized) segment ordering, there are virtually no limitations for the development of parallel applications using Fortran 2008 already. (The only hardware-related limitation might be that atomicity may not be guaranteed between distinct processor types). With, otherwise, safe use of SYNC MEMORY and atomic subroutines, the implementation of all the required synchronization primitives can be accomplished by the Fortran 2008 programmer her-/himself. Further research is required for the design of spin-wait loop synchronizations: nested spin-wait loops do allow for the implementation of NON-BLOCKING customized synchronization primitives.

Units

- Native support for physical units in variable types and compile and run-time exceptions for improper unit combination (e.g. adding two absolute temperatures). This improves code safety and reliability by detecting errors by dimensional analysis.
- Facilities to compute and check consistency of units of measurement in expressions, to check consistency in assignments, to check and convert units in the same family (e.g. length) during formatted input, and to output units during formatted output. The largest program for which I am responsible is an interpreter of a little language in which every input is type checked, and every number is units checked. Units checking in that input has saved significant amounts of time and prevented numerous errors.
- User-Defined Units of Measure (UDUOM). This feature is similar or identical to the proposals by Van Snyder in previous versions of Fortran.

Miscellaneous

- Compatibility with oldest Fortrans. Do not delete or remove things so old code still keeps working.
- I think some old useless features should be dropped and a more clean and fast language is required.
- We would really benefit from some ability to selectively inline small functions declared in other modules - either via some keyword that provides an inlining directive to compilers, and/or via more smarts in the compilers so they can auto-detect good inlining candidates across modules, without greatly sacrificing compilation times.
- Standardized exposure of the application's symbol table to allow application programmability. e.g. SYM("Z") = 1.0 would be treated as Z = 1.0. This can be accomplished now by preprocessing the code to extract symbols and using generated code to create hash tables containing POINTERs to variables or indices into COMMON blocks.
- Threads. Coarrays are not interoperable with any other language with relying on implementation-defined behavior and are inadequate for many applications where threads are used. Application of threads include asynchronous agents to hide latency in I/O (including communication) and dealing with irregular workloads where dynamically scheduled threads naturally deal with load-balancing.

- A general standard of interoperability with python using dll route similar to ctypes in python for C/C++ .

Comments for the committee

- We are very sympathetic about the cost of implementation. We maintain compiler conformance tables and are very aware of the timescales required to get features from 2003 and 2008 implemented in the compilers currently available.
- Fresh from the experience of implementing PDTs in gfortran, I strongly suggest that new features be introduced with a list of associated changes in the text and, if possible, pertinent constraints and restrictions. Better still would be a suite of standard testcases, including errors as well as code that should run.
- The time lag between Fortran standards being published and full compiler support for those standards has to be reduced somehow. The lag in support for F2003 has stunted its adoption. I would go so far as to say that significant new features should be introduced via TR's and subsequently incorporated into Fortran standards. This gives compiler vendors time to implement features in advance of the "big name" release. For example, the TR after F95 that allowed allocatable arrays in derived types. If features aren't widely implemented, have low benefit to implementation cost ratios, or just turn out to be bad ideas, don't be afraid to remove them from the standard. More than likely no one uses them anyway. Think: parameterized derived types, FORALL, etc.
- I happen to write entire Fortran projects in an object-oriented style: abstract base types, procedures called as TBP, etc. Making regular use of alternative type extensions for creating test types and test code, for instance. Also, C++ is the contender for most of our applications. If the committee plans for major additions to the language, such as generic programming tools, I think that any additions should be designed from the beginning to seamlessly collaborate with OO Fortran, at all levels of complexity.

  Also, I would prefer generic and versatile solutions (such as type extension and procedure binding in F2003) to solutions that solve just a particular case (such as parameterized derived types in F2003).
- I'm sure you all will hate my suggestion of threads. However, OpenMP adoption is three orders-of-magnitude greater than coarrays and I don't see any evidence that this will ever change. I understand why multithreading is bad and CSP (or PGAS) is good, but programmers have spoken with their fingers: threads won and PGAS lost. No amount of OpenMP bashing is ever going to change this.
- Nothing specific, except "good work!" - it is easy enough to complain about things, but it is hard work to get the necessary work done.
- Please resist bloat. There is no point adding new features to the standard that are slow to appear in real compilers, or which unnecessarily bloat the language. I fear that this is a trend that has been happening in recent years - many of the new features that have been added are of marginal use and just bloat the compiler and introduce new bugs into a complex piece of software (the compiler).

- It's been a problem for a while that the standard is way ahead of compilers. For example, object oriented features from the 2003 standard are often still buggy with modern compilers. Is there something we can do to improve integration with compilers?
- Thanks for your great work!!
- It's getting increasingly common for people to write the main program in Python and just use Fortran for the heavy lifting. Better/easier interoperability with Python would be great.
- A successful language is much more than a Language Reference Manual; we need a scientific programming ecosystem which supports robust user input processing, testing, and software safety. HPC features do us no good if the risk of re-engineering our code is too great to migrate to an HPC-compatible architecture. The lack of a standard library means effort is wasted reinventing data structures and utility routines which other languages have shipped with for decades. Fortran has been improving but it still has a long way to go.
- I hope that features aren't added for the express purpose of driving out free compilers from the market.
- Live long and prosper, Fortran :) More seriously, some of the suggestions above seem related to existing features in Ada (generics, exceptions...), and I warmly welcome any addition to Fortran that makes it to somehow converge to the Ada language. Rather than C++.
- The more potential errors that can be discovered at compile time, the better.
- I have started using Fortran about 6 months ago to accelerate bits of numerical code. I know that code I write today will compile next year, and I don't need a fancy header library to have basic arrays with bounds checking (unlike C++.) Nevertheless, the semantics of classes, pointers and allocation/automatic deallocation are unclear and confusing.
- Most of the offered suggestions under 1. above are things one might use C/C++ for, extraneous clutter. Why not just use C/C++ if such features are important? There is risk in removing/depreciating unique non-C/C++-like Fortran features; these were initiated by Fortran vendors who served users who actually write and use Fortran in their work.
- Thank you for your work and for considering public feedback on the planned features.
- Continue the good work.
- The committee has tried adding half of the suggested features before. The results of those efforts were not satisfactory. I hope any new proposals are not rehashes of those earlier proposals.
- Thanks for the great work!
- Please don't go overboard. The language is getting bloated, and with each revision, a greater fraction of the language features are rarely used. The compiler writers can't get caught up, so the standard looks like a joke since the features are not available.
- You are great, your work is really appreciated, thank you very much. In my opinion your approach is very sane, Fortran is currently a great language. The only minor suggestion that I humbly propose is to be more "open" to new "technologies". For example, GitHub is a great "hub" to collect, share, test, propose... new idea, resources, connect Fortraners, etc... just to see what are doing our "brothers" the C++ standard committee lives on GitHub.
- I would not be supportive a full blown templating facility because I don't think its necessary for scientific code development IF the data types etc that templating is commonly used to create in other languages are provided as INTRINSIC types.

- One issue I've had with recent standard development is what I perceive as a tendency to define a new feature and then put so many restrictions on it as to make it unusable outside a very narrow range of applications (assumed TYPE would be an example). I can think of a variety of uses for both assumed TYPE and assumed RANK that have nothing to do with C-Interoperability.
- One idea I have for speeding development and implementation of new features would be for WG5 to find funding and a process (maybe through the national labs of the various voting countries) to develop and maintain a "reference" compiler that provides an initial implementation (and it does not necessarily have to be the most efficient implementation) of new features that the commercial and open-source compiler developers can use as a standard reference. The COTS and open-source developers would be free to use the reference compiler implementation as an interim while they develop optimized versions for their specific compilers.
- I would also like to see some thought given to defining a portable module format (maybe in some markup language) that could be generated in addition to a compiler specific format. This would enable libraries (say a plot library) to generate the portable format with one compiler (say GCC) and users could then specify the compiler look for the portable module and compile it into its native module format prior to any USE association. This would greatly enhance the use of these types of libraries and eliminate the need to maintain a different version for each different compiler a user might have on his/her system.
- Don't continue to fall behind contemporary ideas widely available in other languages. This only adds to the "Fortran is obsolete" chorus.
- I have many suggestions:
-
    - 1. Listen to those who are coding actively, particularly in industry. This site and survey are a great start, great job.
    -
    - 2. Help Fortran first achieve parity with languages such as Ada, C++, etc. when it comes to generic and object-oriented programming facilities, PRONTO!
    -
    - 3. Only pay any attention to compiler vendors such as Intel, Cray, and NAG (and to some extent IBM) that consistently show commitment and progress in implementing newer features from newer standard revisions. Opinions of stagnant compiler implementations should carry little weight, they are doing a great disservice to the world of Fortran.
- Thank you for the continuing effort to improve Fortran. I hope that the comments you get from the community will be constructive and useful.
- Do not try to make Fortran into C++! Keep it simple, fast, and gear it toward HPC. If anything, make BLAS, LAPACK , and FFTW intrinsic.
- The key issue I see with Fortran going forward is that there has forever been a tension in what Fortran ought to be:
    - 1. An easy to use language for domain experts to prototype stuff (array syntax is an example of this)
    - 2. A powerful language for people to make things go fast (the rest of the language)
    - 3. A language which is expected to live on top of system programmer languages (which means those system programming languages eat into its "space") I think we need to

make it very clear to ourselves what direction we want Fortran to go. I think overspecializing it will make it disappear, because other languages with 90% of the functionality, or speed, or whatever, will simply start eroding it. My preference is that it be made fast and simple, like Python but not slow. (116)

- Just need to say THANK YOU. Please do not listen to detractors you are doing a great job. (121)
- If templates are added (I hope...) use C++ as a model for what not to do. (126)
- I really appreciate the tremendous efforts of the Fortran committee. I have done essentially all my studies up to now using Fortran, so thanks really much!!