Fortran 95

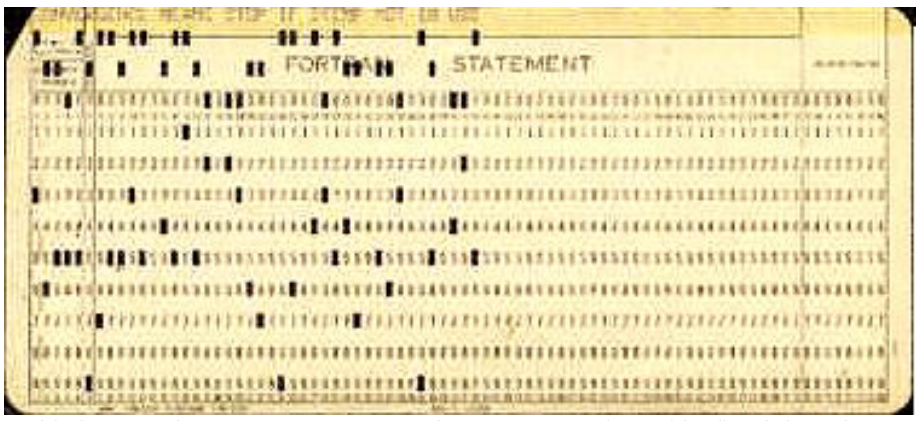
By Eric Greene and Xing Qu

1) Introduction

At more than 50 years of usage, Fortran is one of the oldest computer languages. It was, in fact, the first commonly used high-level computer language. It is a procedural programming language built for numerical computation. It has gone through many version changes, starting off somewhat simple and becoming increasingly complex over the years. In this paper we will discuss: 2) The history of Fortran; 3) who uses the language; 4) what aspects make it different from other languages; 5)specifically what concepts does Fortran embody that relate directly to CS520 topics; 6) and finally then what direction do this paper's authors believe is Fortran's future.

2) History of Fortran, the grandfather of scientific programming languages

Fortran was conceived by IBM worker John Backus. Backus envisioned the idea of an automatic programming system that would take his formulas and translate them into machine code – thus making his programs easier to write. In 1953, he successfully proposed to his superiors that an alternative to assembly language could benefit the company when programming for the IBM 704. Fortran's name was derived from the draft specification that Backus's team submitted in 1954, "*The IBM Mathematical FORmula TRANslating System*". This indicates that the original intention of the language was to translate math and scientific formulas into computer code. At the time of Fortran's inception, programmers Created virtually all code by hand using machine language (assembly, etc.). IBM implemented the language between 1954 (when the specification was made) to 1957 (when the first compiler was delivered). Programmers were initially skeptical of this new type of programming (high-level programming), thinking it's code would lag in performance compared to assembly code. However, once these engineers discovered that Fortran actually out-performed hand-coding (and reducing the required statements to operate the IBM 704 by a factor of 20), the language caught on quickly. Not only was the code more optimal than machine code, it was far less error prone. By the end of 1958, more than half of the programs written for the IBM 704 were in Fortran.



This is an early Fortran program, written on a punch card in fixed-form input

The initial Fortran release (called FORTRAN) was limited, only included 32 statements, using only two data types (integer and real), and had no subroutines. This is to be expected, since FORTRAN was the first attempt at creating a high level programming language. FORTRAN was created to support fixed-form input, primarily for punch card input (this was the primary input source of the day). Flaws became apparent quickly, and IBM released FORTRAN II in 1958. FORTRAN II provided support for procedural programming, thus allowing users to create functions and subroutines.

FORTRAN III was also developed in 1958, however, like previous versions, it did not offer portability as it was machine dependant. This led to IBM creating a machine-independent Fortran in 1961, coined as FORTRAN IV. This iteration of Fortran became the de facto version for the next 15 years. ANSI decided in 1966 to approve FORTRAN IV as a standard Fortran – renamed FORTRAN 66, thus making Fortran the first standardized programming language.

The next version of Fortran was FORTRAN 77. Oddly enough, this standard was formally approved by ANSI in 1988. This edition introduced many valuable upgrades, such as block IF statements (previous versions only used arithmetic IF statements), easy to manipulate characters, and improved I/O capability. Overall, this version made code easier to write and maintain.

In 1992, the ANSI standard for the next major Fortran was released: Fortran 90. This revision contained FORTRAN 77 as a subset and added many new features to update Fortran to match common current programming practices and lead Fortran in a new direction. Notably this version introduced Free-form source input, array sections, array manipulations, inline comments, recursive procedures, parameterized data types, modules, pointers, operator overloading, and more. Fortran 95 is a minor update to Fortran 90, adding several new features to the language such as FORALL, nested WHERE constructs, and pointer initialization. Also, Fortran 95 cleaned up many ambiguities that were prevalent in Fortran 90.

3) Target Audience

Fortran is a language created specifically for engineers, mathematicians, and scientists. As Fortran's original draft name (Formula Translating) implies, Fortran is all about math - it is especially suited for general scientific computations and numerical analysis. It is used to model power plants, climatology systems, and aircraft design systems, as well as create intensive programs in the areas of computation heavy analysis, computational fluid dynamics, and physics, and is also used at times in financial computing. However, Fortran is not only in use due to legacy code and its historical dominance, it truly offers several benefits over more recently created languages commonly in use.

Fortran provides these engineers, scientists, mathematical, and otherwise technical programmers with vectors, matrices, and other higher dimensional arrays containing binary, integer, real, and complex data. Also, Fortran is machine independent, allowing for near full portability (some user written libraries are machine dependent, but the ANSI standard is not).

4) How is it Different? How does it Compare? How is it Unique?

Fortran contains many unique features. Some of the best features have been emulated in other languages (as is common in the industry). In this section, we will visit Fortran's distinctive approach to arrays, parallelism, and modularity.

4.1) *Arrays*:

Array manipulation, speed, efficiency, and flexibility truly are some of the strongest highlights of Fortran 95. Arrays are natural elements in Fortran. This is different than most procedural languages. For example, in C++, you can make an array of integers, but you can't use native array operations on that array without adding definitions to the class dictionary. In Fortran, all array operations are available intrinsically to all array types. Thus, Fortran array manipulation can be very sophisticated. Fortran 95 fully supports efficient vectors and matrices that are often required in scientific computing through multidimensional arrays (unlike Java which contains jagged arrays – or arrays of arrays for imitation multidimensionality – which can take longer in memory to operate on). This true multidimensionality combined with intrinsic operations and quick execution speed makes Fortran a true leader in scientific computing.

There are several interesting aspects about Arrays in Fortran:

4.1.1) There is no specific index in Fortran arrays, unlike most other languages (such as C and Java) that index starting at 0 or 1 and ending at N-1 or N. Thus, array index ranges may start and end at any integer - this is advantageous to engineers and mathematicians who would otherwise have to map the indices to more applicable values. This is not just a nice feature – it can save valuable processing time in computationally complex tasks. The following example code snippets all are arrays that consist of ten elements, with differing index ranges:

- **INTEGER, DIMENTION(10) :: array1**
- **INTEGER, DIMENTION(11:20) :: array2**
- **INTEGER :: array3(-4:5)**

4.1.2) Fortran 95 contains quick manipulations on the entire or part (whether small or large) of an array, often without requiring any loops. It is easy to assign values to a whole array. This is more difficult in C/C++, or pretty much any other procedural language. One of the great benefits in using Fortran 95 is that array operations are not only more easily manipulated code wise, but they are optimized and compile/run faster than in other languages.

Take the following example code which demonstrates Fortran's dynamic and sophisticated array operations:

```
PROGRAM Show_array_ops
    ! create three arrays of size 100
    INTEGER, DIMENSION(0,99) :: array1, array2, array3
    ! create an array of size 200
    INTEGER :: array4(0,199)
    ! this assigns the value 4 to all values in array1
    array1 = 4
    ! this assigns the value 6 to the last 50 elements in array1
    array1(50:99) = 6
    ! this assigns twice the value of array1 at the same position to array2, i.e. array2 elements 0
    ! through 49 will have the value 8; elements 50 through 99 will be 12.
    array2 = 2 * array1
    ! this is a stepping assignment - for every one position in array1 + array2 you return to that
    ! position stepped in array3 (i.e. array 3(0) = array1(0) + array2(0); array 3(2) = array1(1) +
    ! array2(1); array 3(4) = array1(2) + array2(2))
    array3(0:99:2) = array1(0:49) + array2(0:49)
    ! this is basically a shift up, without having to use a loop which is a much faster operation
    ! than in other languages.
    array1(1:99) = array1(0:98)
    ! these next two statements illustrate that arrays of different sizes can be operated on as
    ! long as the size conformity is met.
    array4(0:99) = array1(0:99)
    array4(100:199) = array2(0:99)
END PROGRAM Show_array_ops
```

As is exemplified above, large sections of arrays can be changed without using many lines of code. However, what is not shown is that the execution of this code is incredibly quick as well, rivaling vector operations in Matlab and other matrix-centric languages.

Here is another good example of quick and easy Fortran array manipulation compared to C++:

A recursive for loop in C++:

```
For (int i = 1, i <= n, i++)
{
  arr(i) = arr(i) + arr(i-1)
}
```

The same code, optimized using intrinsic array functions in Fortran:

```
arr(1:n) = (/(SUM(arr(1:i)), i=1,n)/)
```

4.1.3) Fortran arrays can be static, semi-dynamic, and fully dynamic. Most memory in Fortran is static, thus not requiring garbage collection. Most arrays, therefore, are static, unless they are specified as allocatable (dynamic) or an undetermined size (semi-dynamic).

Dynamic arrays are beneficial, because without these allocated arrays, Fortran programs would not be able to efficiently handle variable data (either would have to statically define an array as the largest size that the data could contain at any time or the array would be given the correct size every time the data is analyzed, but requiring constant recompilation and rework). With the allocated arrays, only the size required to process the data is taken in memory. Dynamic arrays initially take no shape – and thus have no overhead in memory.

Dynamic arrays require careful coding due to the user in effect claiming responsibility for allocating and deallocating memory. There are only two ways to reclaim allocated memory in Fortran 95: Deallocate the corresponding structure in the structure's scope; or terminate the program. The syntax for allocating arrays is:

```
INTEGER, ALLOCATABLE :: arr(:,:)! declare a level-2 integer array
~~~~ ; do stuff
ALLOCATE( arr(4,3)) ! allocate arr as a 4x3 array
~~~~ ; do stuff
DEALLOCATE(arr) ! deallocate arr from memory
```

One other useful benefit of Fortran arrays is that variable-dimension array arguments can be passed through subroutines. This feature is unavailable in C. These are semi-dynamic arrays. What happens is that the subroutines take an undefined array as one of its parameters. When a value is passed into the subroutine, that parameter is set to the size of the value and now static in memory.

4.2) *Parallelism*:

With the widespread availability of multiprocessors, scientific computing has recently trended towards using parallelism. Large and time intensive tasks can have processing time drastically reduced using concurrency. However, most procedural languages do not handle parallelism very well. The reason for this is that historically procedural languages used a linear model for finding array values. The model assumes that consecutive elements in an array are in sequential blocks in memory. This model is faulty on parallel computers.

Fortran 95 in turn provided standard language support for multiprocessor operations through its array syntax and intrinsic array operations (such as matrix operations and reduction operations like array sums). This, coupled with polymorphism and overloading, extends the departure from reliance on the sequential memory model in Fortran.

In Fortran 90/95 and on, any data independent loop can be processed in parallel. Each processor used in the multiprocessor execution will take a different (often consecutive) iteration of the loop and execute the loop's code concurrently with other processors. Since different iterations of a loop will be processed at the same time, nothing that changes the expected value of statements between these iterations is allowed. For example, the following code is not data independent, and thus can't be processed in parallel with the same values returned as a sequential execution:

```
Do i = 1, 2
  A(i) = C(i)
  B(i) = A(i+1)
ENDDO
```

Sequential Execution:

T1) A(1) =C(1)
T2) B(1) = A(2)
T3) A(2) = C(2)
T4) B(2) = A(3)

Parallel Execution:

	Process #1	Process #2
T1)	**A(1) =C(1)**	**A(2) = C(2)**
T2)	B(1) = A(2)	**B(2) = A(3)**

The problem with this loop is that A(2)'s value is changed in process #2 at T1 before process #1 accesses A(2)'s value at T2. Thus an unexpected, currupted value will be returned to B(1) in parallel, where the correct value of A(2) is returned in sequential execution.

To process the same loop code to return the same results in parallel as sequentially, the programmer would have to modify the code slightly:

```
Do i = 1, 2
  TEMP =  A (i+1)
  A (i) = C(i)
  B(i) = TEMP
ENDDO
```

Sequential Execution:

T1) TEMP = A(2)
T2) A(1) =C(1)
T3) B(1) = TEMP
T4) TEMP = A(3)
T5) A(2) = C(2)
T6) B(2) = TEMP

Parallel Execution:

	Process #1	Process #2
T1)	**TEMP = A(2)**	**TEMP = A(3)**
T2)	**A(1) =C(1)**	**A(2) = C(2)**
T3)	**B(1) = TEMP**	**B(2) = TEMP**

This loop is now data independent, and thus safe to run in parallel.

Due to its availability to handle concurrency well, Fortran 95 has many specialized parallel-processing versions available across most platforms (High Performance Fortran, Fortran D, among others).

4.3) *Modularity*:

People usually think of Fortran as non-object programming language. Actually, Fortran has fully implemented all of OO stuff since Fortran 2003. But Fortran 95 is only a module-based andobjec t-based programming language, which means the behavior of modularity looks like class. Basically, modules are just static classes. They don't require any memory allocation during runtime. Instead they are static classes. Here we are talking about some object features of modularity which are used to implement object-based Fortran 95:

4.3.1) At first, modules are aimed to group related procedures and data together. This feature is very important especially when several groups or developers cooperated with each other to develop big programs. It is by use of modules that programs communicate with each other. Also, developers don't need to rewrite programs since they can reuse modules which have been developed by others. Later on, this feature is used to implement inheritance between modules in Fortran95. Modules which are correlated with each other are hierarchy grouped together. Functions in module at one lever can be used only by those modules residing on the next level. It goes as if the latter modules inherited all functions from the former module.

Here we have an example about modularity as following:

```
MODULE newbank
  USE bank
  ! Variables in module
  PRIVATE money
  PUBLIC id

  ! functions or procedures
  INTERFACE Report
        MODULE PROCEDURE Report_byID()
        MODULE PROCEDURE Report_byName()
  END INTERFACE

CONTAINS
     SUBROUTINE Report_byID(num)
     ~~~
     END SUBROUTINE Report_byID

     SUBROUTINE Report_byName(name)
     ~~~
     END SUBROUTINE Report_byName
END MODULE newbank
```

In the example, module newbank declares to use another module whose name is bank. Basically, we can look on module bank as parent class, newbank as child class. In other words, all functions which are implemented in module bank can be used as if they were functions of newbank.

4.3.2) Fortan95 allows programmers give different accessibility to member variables and functions. As the above example shows, there are two variables in the newbank module. They can have different accessibilities. One is private, the other one is public. All member variables and functions which have public accessibility can beaccessed by "child" modules directly. This feature is very important for Fortran95 since programmers can get better control of modules. They can effectively limit the accessibility to specific parts of the modules.

4.3.3) Fortran95 takes advantage of "interface" to implement overloading. In last example, we use interface to declare one function and name is "Report". There are two different implementation of Report which are declared after interface. One is Report_byID; the other one is Report_byName. Fortran95 will automatically invoke corresponding functions by the types of passed parameters.

4.4) Other:
As is mentioned in the above section, Fortran isn't fully object oriented. However – it does offer static polymorphism through generics – overloading, etc – and provides encapsulation. Through this and its memory structure, Fortran provides quick compiling and execution – almost all memory is static (the exception is pointers and allocated structures) – especially on multiprocessor machines. Pointers can only point to object that are flagged as pointer targets. This avoids much of the confusion prevalent in C/C++, and also ensures program and memory integrity. Data abstraction in Fortran is strong through the use of user-defined modules and types.

Another reason Fortran is often used that cannot be overlooked is that it has a truly rich library of code. Fortran is almost fully backwards compatible (one might have to use compilers for previous versions though if it contains deprecated or deleted constructs), thus code written in 1966 can still be used with modern Fortran. Scientific programmers have used Fortran for more than half a decade, and thus the code base is massive. With its scalability and constant modernization, Fortran continues to be a viable tool for numeric computation.

5) Fortran as it Relates to CS 520:

In this section, we will discuss how Fortran immediately embodies various topics discussed in class.

5.1) Memory Management

Historically, Fortran versions prior to Fortran 90 only allowed for static memory allocation. This became a problem when dealing with variable size data. For example, if company ACME were to procure an unknown quantity of widgets each month, the program that kept inventory (using static structures like Fortran previous to Fortran 90) of these objects would have to use a structure representing the largest size of widgets the company can obtain. This leads to a massive waste of memory, since the program will mainly process widgets below the largest size available.

However, with Fortran 90, dynamically allocatable structures and arrays are available. Dynamic arrays were discussed in 4.1.3., but Fortran also allows pointers to provide efficient sub-object aliasing. Pointers provide the utility of allowing for the use of dynamic structures (graphs, linked lists, trees, etc.). However, Fortran's pointers are optimized because they can only point to a target if that target has set it's "target" attribute to accept the pointer. Allocation can be done simply like such:

```
INTEGER, POINTER :: p1        ! p1 declares itself a pointer in its declaration using "pointer"
INTEGER, TARGET :: t1 = 25    ! Keyword "TARGET" allows t1's address made available

p1 => t1                      ! p1 now points to t1
```

There are three basic states to a pointer: *undefined*, *associated*, and *disassociated*. A pointer is *undefined* when it is first declared. It becomes *associated* when it is assigned to a valid target. It is *disassociated* when it is disconnected with its previous target (whether assigned to a new target or a NULLIFY statement is used on it). A pointer can only be used in the *associated* state, or it will return an error (the status can be checked using code like **status = ASSOCIATED (pointerName)**).

To avoid pointer ambiguity in the *undefined* state, a pointer should either be assigned or nullified as soon as it is created.

There is no true garbage collector in the Fortran 95 standard. Rather, Fortran is almost fully static, the only exception is when a programmer explicitly allocates and deallocates memory (the only time a garbage collector could be required) through array operations or using pointers. This does lead to dangling references and memory leaks if the programmer isn't careful to deallocate a structure in the same block. The only way allocated memory can be restored is by deallocating or program termination. Thus, it is advisable to use dynamic structures only in programs, and semi-dynamic structures in subprograms (since they can be variable sized, but become fixed once the subprogram is called).

5.2) Types

Fortran is a static, implicitly typed language. It provides basic and advanced data types to make data manipulation quick and simple. Among the types provided by Fortran are discrete types (characters, integers), scalar types (real - single, double, and quad, complex numbers), and composite types (pointers, user defined types). Arrays of any of these types can be declared (i.e. character array = string) natively, as discussed in 4.1. Fortran uses structural equivalence to determine if two types are equivalent. Since Fortran is implicitly typed, it can use type coercion to assign values between similar types (i.e numerical integers, reals, etc). Different types cannot be directly assigned to each other, rather the results of an operation involving one type can be assigned to another. Thus, the following is possible:

```
REAL :: r1
INTEGER :: i1
i1 = 10
~~~~~
r1 = i1/4
```

However, data can be lost between conversions – as in most languages, an integer can't contain the full part of a real, but will truncate the part after the decimal. When two different, yet coercively compatible types are evaluated with each other (i.e. float multiplied by integer), the lower order types will be converted to the highest order type in the evaluation. For Example, (10.33 – 2.4) + 5 would become (10.33 – 2.40) + 5.00 in the compiler.

User defined types are very similar to records in Ada and structures in C. They can be used to hold any Fortran type, whether an intrinsic type or another derived type. The limit on the size of a defined type is the maximum of its parts. Much of the use for the addition of pointers to Fortran 90 and later variants would be unnecessary if it were not for these user-defined types – as linked lists, trees, etc. would not be possible. A generic function such as '*' can be extended to operate on a user defined type.

Fortran doesn't natively support true polymorphic programs. Rather, Fortran supports generic Polymorphism through generic overloading - similar to Ada's module generics. Fortran programmers basically have to use the data abstraction facilities of Fortran (the Module system as stated in 4.3 and user defined types) to carry out any object oriented designs in the system. Fortran handles Polymorphism better than some languages (such as C and Pascal), but is only object-based, and thus not nearly as graceful as fully object-oriented languages like C++ and Java.

The following is an example of a user defined type, with operator definition, in a module:

```
MODULE TestPoints
  TYPE POINT          ! Defines user defined type POINT
    INTEGER :: X      ! The X position
    INTEGER :: Y      ! The Y position
  END TYPE POINT

  INTERFACE OPERATOR (*)
    MODULE PROCEDURE multPoints !calls multPoints when type point is multiplied
  END INTERFACE

  CONTAINS

  FUNCTION pTwo ( pOne, pTwo )
    TYPE( POINT ) multPoints ! this is the output value
    TYPE( POINT ), intent( in ) :: pOne, pTwo ! read in the two points, set pOne and pTwo

    ! this multiplies each element in pOne to the corresponding element in pTwo, and returns
    ! the Point value contained in multPoints
    multPoints = interval(pOne %X * pTwo %X, pOne %Y * pTwo %Y )
  END FUNCTION multPoints
END MODULE TestPoints
```

The main program looks like:

```
PROGRAM UsePoints
  USE TestPoints
  TYPE (POINT) :: pixel1 (10,10)
  TYPE (POINT) :: pixel2 (20,40)
  TYPE (POINT) pixel3
  pixel = pixel1 * pixel2
END PROGRAM UsePoints
```

In the above example, pixel3 will finish with the values X=200, Y=400.

5.3) Other:

Scope -
Fortran is statically scoped. A variable may not have the same name as any greater than its scope. Fortran subroutines can be nested.

Control Structures -
Fortran contains a rich set of control structures. It has the simple contains the following block control constructs: IF, DO, CASE, and FORALL: (**IF-THEN-ELSE** or**IF -THEN-ELSE IF-THEN……-ELSE**, **DO WHILE**, **SELECT CASE**, etc.). Besides iteration, Fortran allows recursion if a procedure "flags" itself for recursion. Fortran uses the value model for all its variables. Most subprograms use pass-by-value, but pass-by-reference is available.

Exception Handling -
There is no explicit exception handling in Fortran. Users have to trap potential errors manually.

5) The Future of Fortran:

Fortran is an old language. It is amazing that it is still in use more than 50 years after its creation. A lot of its success can be attributed to the fact that Fortran filled an important role historically and has continually updated to maintain its viability to handle that role.

Following Fortran 95, there were two other major updates: Fortran 2003 and Fortran 2008. Each of these languages address major issues which remained from Fortran 90/95 (memory leaks from not deallocating dynamic structures, adding full object oriented programming, etc.). However, Fortran continues to be suited mainly for numerical computing. It is doubtful that Fortran will ever make waves in computer science again – it is not a programming language made for programmers, but for engineers and other scientists. Scientific programmers will still use Fortran for computations on supercomputers because it still maintains competitive and mostly superior execution.

Fortran has a great amount of limitation, which makes it simple and inherently transparent. For scientific programming (considering the majority of Fortran programmers are scientist outside the field of computer science), this is ideal and satisfies Occam's Razor, "All things being equal, the simplest solution tends to be the best one."

Combined with an extensive historical code base and its current updates, Fortran should maintain its viability to fill a small, yet very important niche in numeric computing.

References:

1) "*Fortran 90/95 For Scientists and Engineers*" by Stephen J. Chapman, 1998

2) "*Fortran 95 Handbook Complete ISO/ANSI Reference*" by Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, Jarrold, L. Wagener, 1997

3) Fortran Wikipedia Page, http://en.wikipedia.org/wiki/Fortran