Programming Language

FORTRAN

submitted By:

ANKIT JAIN – 14
ASHWANI AGARWAL - 03
A Brief History of FORTRAN

WHAT IS FORTRAN?

- **Formula Translation language.** A high-level programming language that is used primarily for scientific, engineering, and mathematical applications.
- A widely used high-level programming language well suited to problems that can be expressed in terms of algebraic formulas.
- Fortran (also FORTRAN) is a statically typed, compiled, programming language originally developed in the 1950s (IBM) and still heavily used for scientific computing and numerical computation half a century later. The name is a portmanteau of Formula Translator/Translation. Early versions of the language were known as FORTRAN, but the capitalization has been dropped in newer revisions beginning with Fortran 90. The official language standards now refer to the language as "Fortran".

The development of FORTRAN I

The first FORTRAN compiler was a milestone in the history of computing, at that time computers had very small memories (on the order of 15KB, it was common then to count memory capacities in bits), they were slow and had very primitive operating systems (if they had them at all). At those days it seemed that the only practical way is to program in assembly language.

The pioneers of FORTRAN didn’t invent the idea of writing programs in a High Level Language (HLL) and compiling the source code to object code with an optimizing compiler, but they produced the first successful HLL. They designed an HLL that is still widely used, and an optimizing compiler that produced very efficient code, in fact the FORTRAN I compiler held the record for optimizing code for 20 years!

This wonderful first FORTRAN compiler was designed and written from scratch in 1954-57 by an IBM team lead by John W. Backus and staffed with super-programmers like Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert
Nelson, Irving Ziller, Richard Goldberg, Lois Haibt and David Sayre. By the way, Backus was also system co-designer of the computer that run the first compiler, the IBM 704. The new invention caught quickly, no wonder, programs computing nuclear power reactor parameters took now hours instead of weeks to write, and required much less programming skill. Another great advantage of the new invention was that programs now became portable. Fortran won the battle against Assembly language, the first in a series of battles to come, and was adopted by the scientific and military communities and used extensively in the Space Program and military projects. The phenomenal success of the FORTRAN I team, can be attributed in part to the friendly non-authoritative group climate. Another factor may be that IBM management had the sense to shelter and protect the group, even though the project took much more time than was first anticipated.

**FORTRAN II, III, IV and FORTRAN 66**

**FORTRAN II (1958)** was a significant improvement, it added the capability for separate compilation of program modules, assembly language modules could also be 'linked loaded' with FORTRAN modules.

**FORTRAN III (1958)** was never released to the public. It made possible using assembly language code right in the middle of the FORTRAN code. Such "inlined" assembly code can be more efficient, but the advantages of an HLL are lost (e.g. portability, ease of use).

**FORTRAN IV (1961)** was a 'clean up' of FORTRAN II, improving things like the implementation of the COMMON and EQUIVALENCE statements, and eliminating some machine-dependant language irregularities.

A FORTRAN II to FORTRAN IV translator was used to retain backward compatibility with earlier FORTRAN programs.

On May 1962 another milestone was traversed, an ASA committee started developing a standard for the FORTRAN language, a very important step that made it
worthwhile for vendors to produce FORTRAN systems for every new computer, and made FORTRAN an even more popular HLL. The new ASA standard was published in 1966, and was known accordingly as FORTRAN 66, it was the first HLL standard in the world.

**FORTRAN 77 standard**
Formally outdated many years ago, compilers for FORTRAN 77 are still used today, mainly to re-compile legacy code.

**FORTRAN 77 added:**
1. DO loops with a decreasing control variable (index).
2. Block if statements IF ... THEN ... ELSE ... ENDIF.
3. Before F77 there were only IF GOTO statements.
4. Pre-test of DO loops. Before F77 DO loops were always executed at least once, so you had to add an IF GOTO before the loop if you wanted the expected behaviour.
5. CHARACTER data type. Before F77 characters were always stored inside INTEGER variables.
6. Apostrophe delimited character string constants.
7. Main program termination without a STOP statement.

The next Fortran standard (fortran 90) was published too many years after Fortran 77 was out, allowing other programming languages to evolve and compete with Fortran. For example, the system-programming language C, and its evolved variant C++, became more popular in the traditional strongholds of Fortran: the scientific and engineering worlds, in spite of being non-computationally oriented.

**Fortran 90 standard**
A new standard has been designed and widely implemented in recent years. It is unofficially called Fortran 90, and adds many powerful extensions to FORTRAN 77. The language in its present form is competitive with computer languages created later (e.g. C).
Fortran 90 added:
1. Free format source code form (column independent)
2. Modern control structures (CASE & DO WHILE)
3. Records/structures - called "Derived Data Types"
4. Powerful array notation (array sections, array operators, etc.)
5. Dynamic memory allocation
6. Operator overloading
7. Keyword argument passing
8. The INTENT (in, out, inout) procedure argument attribute
9. Control of numeric precision and range
10. Modules - packages containing variable and code

FORTRAN STILL DOMINATES in the numerical computing world, but it seems to lose ground. The following points may help:

FORTRAN tends to meet some of the needs of scientists better.
Most notably, it has built in support for:

Variable-dimension array arguments in subroutines. A feature required for writing general purpose routines without explicitly specifying the array dimensions passed to them.

A rich set of useful generic-precision intrinsic functions. Such functions can be highly optimized (written in assembly language with optimized cache utilization), and they make programs standard at a higher level (and more portable).

Builtin complex arithmetic (arithmetic involving complex numbers represented as having real and imaginary components).

Array index-ranges may start and end at an arbitrary integer, the C convention of [0,N-1] is usually inconvenient. Better I/O routines, e.g. the implied do facility gives
flexibility that C's standard library can't match. The Fortran compiler directly handles the more complex syntax involved, and as such syntax can't be easily reduced to argument passing form, C can't implement it efficiently.

A compiler-supported infix exponentiation operator which is generic with respect to both precision and type, AND which is generally handled very efficiently, including the commonly occurring special case floating-point**small-integer.

Fortran 90 supports an array notation that allows operations on array sections, and using vector indices.

The new intrinsic functions allow very sophisticated array manipulations.

The new array features are suitable for parallel processing.

Fortran 90 supports automatic selection of numeric data types having a specified precision and range, and makes Fortran programs even more portable.

Fortran extensions for parallel programming are standardized by the High Performance Fortran (HPF) consortium.

Fortran 90 supports useful features of C (column independent code, pointers, dynamic memory allocation, etc) and C++ (operator overloading, primitive objects).

**The design of FORTRAN allows maximal speed of execution:**

FORTRAN 77 lacks explicit pointers, which is one reason that it is more amenable to automatic code optimization. This is very important for high-performance computing.

Fortran 90 allows explicit pointers restricted to point only to variables declared with the "target" attribute, thus facilitating automatic optimizations.

Fortran was designed to permit static storage allocation, saving the time spent on creating and destroying activation records on the stack every procedure call/return.
Recursive procedures are impossible with static allocation, but can be simulated efficiently when needed (very rare). Frtran implementations may pass all variables by reference, the fastest method.

Fortran disallows aliasing of arguments in procedure-call statements (CALL statements and FUNCTION references), all passed argument lists must have distinct entries.

Fortran disallows also aliasing between COMMON (global) variables and dummy arguments.

These restrictions allows better compiler optimizations.

There is a vast body of existing FORTRAN code (much of which is publicly available and of high quality). Numerical codes are particularly difficult to port, scientific establishments usually do not have large otherwise idle programming staffs, etc. so massive recoding into any new language is typically resisted quite strongly.

FORTRAN 77 tends to be easier for non-experts to learn than C, because its 'mental model of the computer' is much simpler. For example, in FORTRAN 77 the programmer can generally avoid learning about pointers and memory addresses, while these are essential in C. More generally, in FORTRAN 77 the difference between (C notation) x, &x, and often even *x is basically hidden, while in C it's exposed. Consequently, FORTRAN 77 is a much simpler language for people who are not experts at computer internals.

Because of this relative simplicity, for simple programming tasks which fall within its domain, (say writing a simple least-squares fitting routine), FORTRAN 77 generally requires much less computer science knowledge of the programmer than C does, and is thus much easier to use.

Fortran 90 changes the picture somewhat, the new language is very rich and complex, but you don't have to use or even know about all this complexity. The C standard requires only a basic double-precision mathematical library, and this is
often what you get. The FORTRAN standard, on the other hand, requires single &
double precision math, many vendors add quad-precision (long double, REAL*16)
and provide serious math support.

Single-precision calculations may be faster than double-precision calculation even on
machines where the individual machine instructions takes about the same time
because single-precision data is smaller and so there are less 'memory cache
misses'. Quad-precision (long double) calculations are sometimes necessary to
minimize roundoff errors.

If you have only double-precision mathematical routines, the basic mathematical
primitives will take up unnecessary CPU time when used in single-precision
calculations and will be inexact if used with 'long double'.

**FORTRAN is designed to make numerical computation easy, robust
and well-defined:**

1) The order of evaluation of arithmetical expressions is defined precisely, and an be
controlled with parentheses.

2) The implicit type declaration feature saves time/typing (however it makes your
program vulnerable to annoying and hard to detect bugs).

3) Case insensitivity eliminates bugs due to 'miscased' identifiers.

4) The lack of reserved words in the language gives the programmer complete
freedom to choose identifiers.

5) The one statement per line principle (of course continuation lines are allowed with
a special syntax) makes programs more robust.

6) Added blanks (space characters) are insignificant (except in character constants)
this also to the robustness of FORTRAN programs.
7) Linking with the mathematical library doesn't require any compiler option (in C you to have to use "-lm").

Last but not least, FORTRAN compilers usually emit much better diagnostic messages.

In summary, we can say that the difference between FORTRAN and C is the difference between a language designed for numerical computations, and a language designed for other purposes (system programming).

**GENERAL PROGRAMMING RULES**

Programming rules are an attempt summarize the programming experience of programmers, and the theoretical considerations raised by computer scientists. Some programming rules apply to all programming language, they stem from general principles like:

A) Modern computers are very powerful, memory became relatively cheap, and modern compilers automatically optimize code better than the average programmer.

So, saving memory or CPU time is no longer of prime importance: what is important is creating programs that are easy to VALIDATE (check that the results are correct) and MAINTAIN (change to suit different future needs).

B) The building units of program code, procedures and functions, should be made as GENERAL (suitable for many similar applications) and FLEXIBLE (able to handle different types of input and computing requirements) as possible.

This principle promotes CODE REUSABILITY (using parts of old programs in new ones), and also helps improve coding since programs that don't rely on special assumptions tend to be more reliable.
C) Programs should have a well organized internal structure. The MODULARITY PROGRAMMING PARADIGM requires partitioning every program into relatively small units, each performing a well defined task.

The interaction between code units can be described by a PROCEDURE CALLING TREE, the calling tree makes clear also the internal structure of the program.

A subprogram call has a performance cost, so sometimes you don't partition the program to many subprograms but just arrange it in well defined blocks, optimizing compilers does such things automatically (procedure inlining).

(Some new programming paradigms like OBJECT ORIENTED PROGRAMMING and VISUAL PROGRAMMING are not yet supported directly by the current Fortran languages and dialects. It seems that implementing these paradigms degrade performance.)

D) Programs should be fully documented. Documentation consists of:

1) Self-explaining names for variables, procedures etc.

2) Short comments explaining the algorithms used, variables' roles and computation steps.

3) Program header containing: A general description, bugs and problems, planned improvements, and redistribution information.

4) Procedure headers similar to the program header.

5) A text file explaining how to compile, link, install and use the program (and/or a Makefile).

E) Programs should be portable. To achieve this goal you should:

1) Strictly adhere to the FORTRAN STANDARD
2) Avoid using operating system routines
3) Avoid using any processor dependent features

Languages standards are technical specifications describing a programming language in detail. Compiler writers check their compilers against the standard, and so ensure that a program that compiled correctly on one standard conforming compiler will compile correctly on another such compiler.

The problem is that most compilers offer nice LANGUAGE EXTENSIONS, tat sometimes are quite tempting to use, and may even be included in future revisions of the standard. However, if you use language extensions, your program might not compile on other machines without some rewriting, and so might be less useful.

F) A program source is really just a text file; as such it can and should be as READABLE and EASY TO EDIT as possible. Readable programs are easier to maintain and validate.

G) A program should not depend on assumptions about the correctness of user's input, guesses about expected data etc, this is called DEFENSIVE PROGRAMMING.

Make your program check user's input and key computation results, if wrong, try to recover - go back to the last trusted point, or just give a detailed message and abort the program.

It is very important to carefully select appropriate algorithms.

In the general programming rules section, it was emphasized how important it is to adhere to the standard of FORTRAN, and avoid fancy but unportable language extensions.
FORTRAN DATA TYPES

There are two kinds of FORTRAN constants:

1) **Named constants**, are declared with the PARAMETER statement and are used by the name it assigned. Named constants can be assigned a data type with the ordinary data type declarations, but are not variables, in the sense that memory storage is allocated differently.

The data type of the constant is determined either by an explicit declaration, or by an implicit one, not by the constant's form.

2) **Unnamed constants**, are used explicitly in the source code. The compiler infers the data type from the constant's form, e.g. "1.0E-02".

Some compilers use context-dependent considerations in inferring the data type, e.g. if the constant is assigned to a variable, it gets the data type of this variable.

Another useful classification is to **typed and type less constants** -

1) Ordinary constants are typed, i.e. have a well-defined data type.

2) Type less constants are specified directly in terms of the bit patterns representing them in the hardware and are therefore not portable.

There are **two kinds of type less constants**: bit and Hollerith

DATA TYPES

**The star notation**

The 'star' (*n) notation is standard Fortran for character strings, using the star notation for non-character (numeric) data-types is common, but not standard Fortran, either FORTRAN 77 or Fortran 90.

The Fortran standards can't allow the star notation because they must avoid any restriction on the internal representation of variables, otherwise Fortran programs will not be portable to machines with different architectures.

The number after the star in character strings is the number of "character storage units", whose size is purposely left undefined and is never compared with the "numeric storage unit" used to describe numeric types. The number of "character storage units" is the number of characters in the string, not the number of bytes.

Almost all "English speaking" computers use one-byte representations for characters, so the number after the star coincide with the number of bytes used in the internal
representation. In the non-standard use of the star notation for numeric data types, the number after the star also denotes the number of bytes used.

**Basic data types**

The data types provided by a language have little to do with the underlying machine, there were perfectly usable Fortran compilers on 8-bit computers, which simulated 32-bit floating point hardware.

The basic data types of FORTRAN 77 are: **integers, floating-point numbers** of two kinds (REAL and DOUBLE PRECISION), **logical and character strings**.

**CHARACTER STRINGS**

A character is a graphic sign that may be represented differently in different representation schemes developed and used on different machines.

Individual characters (CHARACTER*1) are always represented in practice as small integers, usually in the range 0-255 so they can be stored in one byte.

The exceptions to the 'one byte per character rule' are:

1) Unicode systems - characters are consistently 16-bits per character.

2) Asian language systems - the number of bytes per character is variable.

Because characters are represented, in practice, always as small integers, and representations are what computers use and work with, people sometimes tend to think that the integer representation is the primary entity.

**BYTE**

A NONSTANDARD fortran data type, equivalent to C 'signed char'.

Can be thought of as a small subtype of INTEGER, formally equivalent to LOGICAL*1, which is also nonstandard fortran.

Should be used only if you have to do fast byte manipulations on data.

- Size: 1 byte (8 bits)
- Value range: -128 to 127.

**INTEGER**

The standard INTEGER data type was implemented as INTEGER*2 on old 16 bit machines, it is now usually INTEGER*4, but on the new 64 bit machines (and some older ones) it may be an INTEGER*8.

FORTRAN integer types are signed:
INTEGER*2 (non-standard)

------------------------
Size: 2 bytes (16 bits, bit 15 = sign)
Value range: -32768 to 32767

INTEGER*4

Size: 4 bytes (32 bits, bit 31 = sign)
Value range: -2147483648 to 2147483647

LOGICAL

The FORTRAN standard requires logical variables to be the same size as INTEGER/REAL variables (see the chapter on memory management) although only one bit is really needed to implement this type.

The values used to implement the logical constants .TRUE. and .FALSE. differ:

<table>
<thead>
<tr>
<th></th>
<th>VMS</th>
<th>Sun</th>
<th>IRIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>.TRUE.</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>.FALSE.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Unix machines naturally adopted the C convention, VMS has a seemingly strange value for .TRUE., however on a closer look you will see that if .FALSE. is "all bits 0", .TRUE. should be "all bits 1", in two's complement signed integers the number with all bits set to 1 is -1.

ARRAYS

An array is an allocated area in memory, together with the information needed to interpret it correctly. The information needed for that consists of:

1) The data type of the elements. All elements have the same data type, so it's natural to attribute the same data type to the array itself. In other words there is one data type per array, and it applies to all elements.
2) Number of dimensions
3) The start-index and end-index of each dimension
4) Specification of the element order in memory

The compiler needs all this information in order to generate references to array elements, or implement whole array operations. Since a FORTRAN compiler compiles separately each procedure, this info must be available in the called procedure, when passing an array to a procedure.

Possible mechanisms for passing array configuration info are combinations of the following:
1) Explicit declarations in the called procedure
2) Using the procedure arguments or common block variables
3) Ignoring some of the info, if possible
4) A special data structure passed with the array itself

For example, a formula for a one-dimensional array may be similar to:

**Memory-address = Base-address + Element-size * (Array-index - Start-index)**

The default start address in FORTRAN is of course 1.

**C Main program for both examples**

```fortran
PROGRAM TEST1
   INTEGER N
   PARAMETER (N = 10)
   REAL Y(N)
   Y(1) = 0.123456
   CALL A(N,Y)
   CALL B(Y)
END
```

**Adjustable size array method**

```fortran
SUBROUTINE A(N,Y)
   INTEGER N
   REAL Y(N)
   WRITE (*,*)
   WRITE (*,*) ' Adjustable array: ', Y(1)
END
```

**Assumed size method**

```fortran
SUBROUTINE B(Y)
   REAL Y(*)
   WRITE (*,*)
   WRITE (*,*) ' Assumed size array: ', Y(1)
END
```

Both routines will write the first element of the array Y.