A Brief Overview of Fortran

Raymond J. Spiteri

Lecture Notes for CMPT 898:
Numerical Software

University of Saskatchewan

April 4, 2013

Objectives

- A brief overview of Fortran 77
- A brief overview of Fortran 95

Fortran: Overview

From Wikipedia,

> *Fortran (previously FORTRAN) is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing.*

Fortran was originally developed at IBM between 1954 and 1957 by a team at IBM led by John Backus.

The name derives from *The IBM Mathematical* **For***mula* **Tran***slating System*.

It was the first of the high-level programming languages above assembly language.

Fortran has had several incarnations since then, including Fortran 77 and Fortran 95.

It has progressively moved away from its original sleek design that led to executable code that was close to machine code.

Fortran: Overview

So the latest versions of Fortran have many modern features of programming languages such as dynamic memory allocation but at the cost of reduced performance.

Despite being the F-word in many university computer science departments, it remains one of the most popular languages in high-performance computing.

Moreover, recent surveys among industry practitioners reveal that knowledge of Fortran is valued.

It is undeniable that there is a great deal of legacy code that is written in Fortran; so it is relatively certain that serious use of numerical software will intersect with code written in Fortran.

It has been said that whatever programming language is used in the future, it will be called Fortran!

Fortran 77

Fortran 77 is a positional language: each line of the program file is divided into subfields.

Columns 1-5 are reserved for statement labels, which are unsigned integers having at most 5 digits.

Statements and declarations may be written in columns 7-72 (inclusive).

Card numbers, which were relevant when punch cards were used, may appear in columns 73-80.

Accordingly, characters beyond column 72 will not be read by a Fortran compiler.

This is a very common source of error (and consternation!).

Each new line represents a new statement unless there is a nonblank continuation character in column 6.

In this case, the statement from the preceding line is continued to this line.

Classically, a comment is denoted by a C in column 1: the rest of the line is ignored by the compiler.

Each comment must have a C in column 1: comments may not be continued with a continuation mark in column 6.

Modern Fortran 77 compilers may allow comments to be denoted by `*` or `!`, where the latter can be used in any column to denote that the remainder of the line is to be treated as a comment.

Whitespace may be used freely in Fortran 77.

So, it is possible to write programs so that their physical layout reflects their logical structure.

In the following, we adopt the convention that items appearing inside [] are optional while items appearing in {} may appear 0 or more times.

Fortran 77: Program structure

A Fortran 77 program has the following structure:

```
[PROGRAM]
{declarations}
{statements}
[STOP]
END
[{subprogram[s]}]
```

where a `subprogram` takes the form of either a

```
SUBROUTINE name [ ( id { , id } ) ]
{declarations}
{statements}
[RETURN, STOP]
END
```

or a

```
FUNCTION name ( id { , id } )
{declarations}
{statements}
[RETURN, STOP]
END
```

Fortran 77: Program structure

`id` represents an identifier.

The parameter list for a subroutine may be empty, in which case the parentheses following the name of the subroutine are optional.

However, parentheses are always required for a function even if the parameter list is empty.

As is common, the value that the function is to return is assigned to the function name.

A function name must be declared in any program or subprogram that uses it.

If a subprogram is passed as a parameter to another subprogram, then the name of the subprogram being passed should be declared to be EXTERNAL in the program or subprogram that contains the passing call.

Subprograms are not recursive in Fortran 77![1]

[1] Some compilers may allow recursion; but then portability is impacted.

Fortran 77: Program structure

Subroutines are called using the `CALL` statement

`CALL name [ ( expr {, expr } ) ]`

where `expr` is an expression, i.e., a combination of one or more operands, zero or more operators, and zero or more pairs of parentheses.

Functions are called by using their name followed by an appropriate argument list of the same form as in the CALL statement in an expression.

The argument list of a subprogram must have as many arguments as there are parameters to the subprogram; they are matched on a one-to-one basis.

The types of arguments must match appropriately, but this is not enforced rigorously.

If a value is to be returned through an argument, then that argument must be a variable or an array element.

Fortran 77: Identifiers

Identifiers in Fortran 77 must begin with a letter and may contain up to six letters and digits.

Using such short names effectively requires practice!

Many compilers allow longer names albeit at the expense of portability.

Generally one has to be aware that only the first six letters and digits define the identifier; letters and digits after the first six may be ignored.

Fortran 77: Type

Each variable used in a program should be declared in a type statement of the form

`type variable {, variable }`

where `type` is `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`, `EXTERNAL`, or `CHARACTER*n` for a positive integer `n`.

EXTERNAL is used to declare a subprogram name in a program segment that passes that subprogram name as an argument to another subprogram.

There is no double-precision complex type in standard Fortran 77, but many compilers provide one (at the expense of portability).

Fortran 77: Type

A variable may be a simple identifier or an array identifier declared as

`identifier` (`range` $\{$, `range` $\}$)

where `range` is of the form

`n0` or `n1 : n2`

In the first case, `n0` ≥ 1, and the array has elements indexed $1, 2, \ldots,$ `n0`.

In the second case, `n1` $\leq$ `n2`, and the array has elements indexed `n1`, `n1`$+1, \ldots,$ `n2`.

An array element is referenced in the program as

`identifier` (`expression` $\{$, `expression` $\}$)

where each expression must lie within the declared range of that dimension.

When an array index falls outside the range of the array, the program crashes with a segmentation fault.

Fortran 77: PARAMETER statement

A `PARAMETER` statement has the form

`PARAMETER ( name = expr {, ... } )`

where `...` is another instance of `name = expr` and `expr` is an expression.

The expressions may contain the arithmetic operators +, − , *, /, and **; the exponents used with ** must be integers.

The value assigned to the identifier is fixed throughout the program.

Note: The PARAMETER statement does not declare a variable; it must be declared in a type statement.

Otherwise, the variable will be given a default type, and this may cause errors if this type is not suitable.

Fortran 77: DATA statement

`DATA` statements are used to initialize variables.

A `DATA` statement has the form

`DATA nlist / clist /` $\{$`, nlist / clist /` $\}$

Each `nlist` is a list of variable names, array names, array element names, or implied DO lists.

Each `clist` is a list of constants or symbolic names of constants and may be preceded by an `r*`, where `r` is an unsigned positive integer indicating `r` occurrences of the following constant in the list.

The values in the `clist` are assigned to the variables in the `nlist` on a one-for-one basis.

Example:

```
DATA a, b, c / 0.0 2*1.0 /
```

Fortran 77: COMMON blocks

A `COMMON` block has the form

```
COMMON / blockname / list
```

where `list` is a list of identifiers or array elements.

No subprogram argument or function name may appear in the list.

Fortran has no global variables per se.

Common provides a means of sharing variables between program segments.

The `blockname` must be the same in each segment.

The identifiers in the list are matched one-for-one by position proceeding from left to right, irrespective of their names or types in different program segments.

Fortran 77: Assignment

An assignment statement has the form

```
variable = expression
```

The left side may be either a simple variable or an array element.

Arrays cannot be assigned all at once: they must be assigned an element at a time in a DO loop.

If the type of the left and right sides do not match, the value of the expression is converted to the type of the left side variable (if possible) after the expression has been evaluated.

Fortran 77: GO TO statement

A `GO TO` statement is of the form

`GO TO label`

and transfers control to the statement with label `label` in columns 1-5.

`GO TO` statements are one of the reasons programming purists despise Fortran.

Indeed, their indiscriminate use can lead to terrible-looking (and badly behaving) programs.

They can be used to construct well-structured loops.

Fortran 77: CONTINUE statement

A `CONTINUE` statement does nothing per se, but it serves as a convenient point to which to attach a label, in particular with respect to loops.

Fortran 77: Logical constants, expressions

A logical constant is either `.TRUE.` or `.FALSE.`

A logical expression is of the form

```
logical constant
logical variable
comparison
.NOT. logical expression
logical expression .AND. logical expression
logical expression .OR. logical expression
```

where `comparison` takes the form

```
expression comparator expression
```

and `comparator` is one of `.LT.`, `.LE.`, `.GT.`, `.GE.`, `.EQ.`, or `.NE.`

Expressions in brackets are evaluated first, then comparisons, then `.NOT.s` applied, then `.AND.s` performed, followed by `.OR.s`.

Fortran 77: Conditional statements

Fortran is capable of logic using `IF` constructs.

```
IF ( logical expression ) statement
```

The parentheses around the logical expression are required in all `IF` statements in Fortran 77.

If `logical expression` evaluates to `.TRUE.`, the statement is executed; otherwise it is not.

In this construct, only one executable `statement` may be included.

To execute more than one statement, the construct is

```
IF ( logical expression ) THEN
{ statements }
END IF
```

If `logical expression` evaluates to `.TRUE.`, the statements between the `IF` and the `END IF` are executed; otherwise they are not.

Fortran 77: Conditional statements

`ELSE` clauses are possible in Fortran via

```
IF ( logical expression ) THEN
{ statements }
ELSE
{ statements }
END IF
```

If `logical expression` evaluates to `.TRUE.`, the statements between the `IF` and the `ELSE` are executed; otherwise the statements between the `ELSE` and the `END IF` are executed.

Fortran 77: Conditional statements

```
IF ( logical expression ) THEN
{ statements }
{ ELSE IF ( logical expression ) THEN
{ statements } }
[ ELSE
{ statements } ]
END IF
```

The `statements` after the first `logical expression` that evaluates to `.TRUE.` are executed; all the others are skipped.

If no `logical expression` evaluates to `.TRUE.` and there is an `ELSE` part, then the `statements` after the `ELSE` are executed.

In any case, execution resumes after the `END IF`.

`IF` constructs may be nested.

An `ELSE` clause is always associated with the closest preceding `IF` statement.

Fortran 77: Loops

Fortran 77 has no general loop ... end loop structure.

One can be built using `IF` and `GO TO` statements.

```
C LOOP
label1 CONTINUE
{ statements }
C EXIT WHEN logical expression satisfied
IF (logical expression) GO TO label2
{ statements }
GO TO label1
C END LOOP
label2 CONTINUE
```

Fortran 77: DO Loops

Fortran 77 does support indexed loops (`DO` loops):

```
DO label identifier = start, limit [, step]
{ statements }
label CONTINUE
```

where `start`, `limit`, and `step` are expressions that are evaluated before the loop is executed.

The loop is executed with `identifier` initialized to `start` and incremented by `step` (which has a default value of 1) until the value of `identifier` is greater than `limit`.

The statements within the loop are not executed if `start` is greater than `limit` and `step` is positive or `start` is less than `limit` and `step` is negative.

It is usually good practice for all counters to be integers, but other types are allowed.

Fortran 77: I/O

Input and output are performed using the `READ` and `PRINT` commands, respectively.

We illustrate how to use `READ`; simply replace with `PRINT` where desired.

```
READ * [, variable {, variable } ]
```

or

```
READ label [, variable {, variable } ]
label FORMAT ( format item {, format item } )
```

An `expression` may be any valid expression or an implied `DO` list.

A `variable` is a simple identifier, an array element, or an implied `DO` list.

Fortran 77: I/O

An implied `DO` list has the form

```
( dlist, identifier = start, limit [,step] )
```

where `dlist` is a list of permissible input or output items, and `start` , `limit`, and `step` are expressions.

The implied `DO` works just like a `DO` loop and is frequently used to read and write arrays; e.g.,

```
READ *, (a(i), i=1,10)
```

reads elements `a(1)`, `a(2)`,. . . , `a(10)` of the array `a`.

The `*` form uses default formats.

This is the recommended form in which to use `READ`.

However, with `PRINT`, it is often useful to use a customized output format rather than the default.

Fortran 77: I/O

In a `FORMAT` statement, a `format item` can be

- `nX`: skip the next `n` columns.
- `nIw`: `n` integers, right-justified in fields of width `w`.
- `nFw.d`: `n` real or double-precision values without exponents right-justified in fields of width `w` with `d` digits to the right of the decimal point.
- `nEw.d`: `n` real or double-precision values with an exponent, right-justified in fields of width `w` with a leading `0`, followed by a decimal point, followed by `d` digits.
- `nDw.d`: like `nEw.d` except that the exponent is marked by `D` rather than `E`.
- `nAw`: `n` groups of `w` characters.

Fortran 77: I/O

Fortran uses the first character of each output line for carriage control.

The first character should be one of the following.

- ' ' (blank) start a new line.
- `'1'`: start a new page.
- `'0'`: skip a line then start a new line (double space).
- `'+'`: go back to the beginning of the current line (overprint).

Fortran 77: Arithmetic constants and expressions

Integer constants are of the form

`[ sign ] digit { digit }`

Real constants are of the form

`[ sign ] { digits } . { digits } { E [ sign ] digit { digit } }`

where at least one of the two groups of `digits` surrounding the decimal point must be nonempty.

A few valid real numbers are

`1. .33 1.5E4 0.333E+10 4.2E-10`

A double-precision number looks similar to a real number but the exponent `E` is replaced by a `D`.

The `D` is not optional!

Without it, the number is a single-precision real.

Fortran 77: Arithmetic constants and expressions

Expressions are formed using the arithmetic operators `+`, `-` , `*`, `/`, `**`, where `**` represents exponentiation.

If `a` and `b` are integers, then `a/b` returns the integer quotient of `a` divided by `b` (e.g., $3/2 = 1$).

Otherwise, the operators are as one would expect.

The precedence of these operators is `**` highest, `*` and `/` next, and `+` and `-` lowest.

Operators of equal precedence are evaluated from left to right, except for `**`, which is evaluated right to left.

`A**B` is computed by repeated multiplication if `B` is an integer, but it is computed as `exp(B*log(A))` otherwise.

Hence, if `A` is negative and B is non-integer, an exception occurs even though the expression may be mathematically valid.

Fortran 77: Character constants

A character constant is any string of characters enclosed in single quotes not containing a single quote.

A quote may be included in a character constant by using two single quotes in a row.

Some examples are

`FRED, X = 24, MR. OREILLY`

Fortran 77: Intrinsic functions

Fortran 77 has many useful intrinsic functions, including

- `MOD(m,n)` the remainder of `m` divided by `n`
- `ABS(x)` $= |x|$
- `MAX(x1,...,xn)` the maximum of `x1`, . . . , `xn`
- `MIN(x1,...,xn)` the minimum of `x1`, . . . , `xn`
- `SQRT(x)` $= \sqrt{x}$
- `EXP(x)` $= e^x$
- `LOG(x)`, `LOG10(x)` $= \log_e(x)$, $\log_{10}(x)$
- `SIN(x)` $= \sin(x)$ (`x` in radians)
- `ASIN(x)` $= \arcsin(x)$ (result in radians)

Sample program

Here is a simple program to calculate $n!$.

```
c     Sample program to compute and print n!
c     for n = 0,...,limit

      integer n, limit
      double precision fac
      parameter ( limit = 20 )
      do 10 n = 0,limit
         print 5, n, fac(n)
 5       format('n = ',i2,3x,'n! = ',d15.5)
 10   continue
      stop
      end

      double precision function fac(n)
c     This function computes n!.
c     It assumes without checking
c     that n .ge. 0
      integer i,n
      fac = 1.d0
      do 10 i = 1,n
         fac = fac * dble(i)
 10   continue
      return
      end
```

Sample program

Here is the output:

```
n =  0   n! =     0.10000D+01
n =  1   n! =     0.10000D+01
n =  2   n! =     0.20000D+01
n =  3   n! =     0.60000D+01
n =  4   n! =     0.24000D+02
n =  5   n! =     0.12000D+03
n =  6   n! =     0.72000D+03
n =  7   n! =     0.50400D+04
n =  8   n! =     0.40320D+05
n =  9   n! =     0.36288D+06
n = 10   n! =     0.36288D+07
n = 11   n! =     0.39917D+08
n = 12   n! =     0.47900D+09
n = 13   n! =     0.62270D+10
n = 14   n! =     0.87178D+11
n = 15   n! =     0.13077D+13
n = 16   n! =     0.20923D+14
n = 17   n! =     0.35569D+15
n = 18   n! =     0.64024D+16
n = 19   n! =     0.12165D+18
n = 20   n! =     0.24329D+19
```

Fortran 95

In many ways, subsequent versions of Fortran, like Fortran 90, 95, 2003, certainly are reminiscent of Fortran 77, but there are some key differences.

Consider the following code:

```
! A Hello, World! program
PROGRAM HELLO

  PRINT*, 'Hello, World!' ! the classic message

END PROGRAM HELLO
```

Fortran 95 is free source form; i.e.,

- statements can begin in any column
- statements on one line can be separated by `;`
- `!` denotes the start of a comment no matter where
- statements can be continued by appending &

Fortran 95: Identifiers and declarations

Identifiers in Fortran 95 are subject to the following constraints:

- up to 31 characters
- first character must be a letter
- case-insensitive

Variables and their types are declared at the beginning of the program.

The intrinsic data types are `character`, `logical`, `real` (single precision), `double precision`, and `complex`.

Fortran 95: Identifiers and declarations

The general form a declaration in F90/95 is

```
<type> [, <attribute_list>] :: &
[, <variable>[= <value>]]
```

where < attribute_list > contains attributes like PARAMETER, SAVE, INTENT, POINTER, TARGET, DIMENSION, etc.

Any object may be given any number of attributes, provided they are compatible with each other.

Fortran 95: Identifiers and declarations

Declarations for `integers` look like

```
INTEGER I,J,K
```

or

```
INTEGER :: I,J,K
```

Normally, when variables are declared, their values are undefined.

In Fortran, they are often just set to 0.

In Fortran 95, an `integer` can be declared and initialized as

```
INTEGER :: I=1
```

Fortran 95: Identifiers and declarations

`PARAMETER`s can be defined as

```
INTEGER BIG
PARAMETER (BIG=6)
```

or directly as

```
INTEGER, PARAMETER :: BIG=6
```

The `::` form is required when more than one attribute is ascribed to a variable, e.g., `INTEGER` and `PARAMETER`, or when a variable is declared and initialized in one go.

Fortran 95: Strings

For characters strings, we can specify the length (and optionally initialize) as

```
CHARACTER(LEN=7) :: LOGIN
CHARACTER*7 LOGIN
```

With hard-coded `lens`, strings are padded with spaces or truncated to make the string the declared length.

Wildcards can be used to avoid this; e.g.,

```
CHARACTER(LEN=8) :: LOGIN, PASSWORD*12
```

A string is a scalar not an array of characters. So, it is possible to declare a 10×10 matrix whose elements are 6-character long strings:

```
character(len=6), dimension(10,10) :: A
```

Strings can be split across lines by adding & at the end of the current line and the beginning of the next; e.g.,

```
 "I love com&
&puting"
```

Fortran 95: Implicit declarations

In Fortran, implicit declarations are allowed.

In other words, we can just use variables `I`, `J`, `X`, `Y` without declaring them.

The Fortran compiler automatically declares `I`, `J` as `INTEGER`s and `X`, `Y` as `REAL`s.

The convention is that undeclared variables that start with `I`, `J`, `K`, `L`, `M`, or `N` are considered `INTEGER`s, and the rest are `REAL`s.

The automatic declarations based on implicit types are called implicit declarations.

Implicit declarations are permitted but frowned upon.

Their use can mask programming errors and negatively impact future development and maintenance.

For example, a misspelling of a variable name will result in a new variable declaration, which can be further assigned, etc., with the programmer being unaware.

An example (from A.C. Marshall) is

```
DO30I = 1.100
<statements>
30 CONTINUE
```

Instead of a DO loop, because of the typo, we end up with a new real variable, `DO30I`.

We generally recommend disabling the implicit declarations by placing the command `IMPLICIT NONE` as the first line after any `USE` statements (i.e., before the declarations sequence).

Then, the existence of variables that are not explicitly declared will lead to a compilation error.

This may be less of an issue in Fortran 77 if you can respect the naming convention.

The convention can be arbitrarily changed; e.g.,

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

declares all variables that are non-integer by default to be `DOUBLE PRECISION` instead of `REAL`.

Fortran 95 vs. Fortran 77

`DO` loops can be performed in Fortran 90/95 without reference to a line number.

```
DO <DO_var> = <n_expr1>, <n_expr2>[, <n expr3>]
<exec_stmts>
END DO
```

The loop can be named and the body can contain `EXIT` or `CYCLE` statements.

Fortran 95 vs. Fortran 77

arrays

procedures

Fortran 95: Derived types

Compound entities (like `structs` in C) are called derived types in Fortran 90/95.

For example,

```
TYPE POINT
REAL :: X,Y,Z
END TYPE POINT
```

packs the co-ordinates of a point into one variable.

An object of type Point can be declared in a type declaration statement

```
type(Point) :: A, B
```

To select individual components of a derived type object, we use the % operator; thus

```
A%x = 1.0
A%y = 2.0
A%z = 3.0
```

assigns the values 1, 2, 3 to the coordinates of `A`.

Fortran 95: Derived types

Alternatively, it is possible to use a derived type constructor to assign values to the whole object.

```
A = POINT( 1.0, 2.0, 3.0 )
```

Assignment between two objects of the same derived type is intrinsically defined; e.g.,

```
B = A
```

Fortran 90/95 does not imply any form of storage association; so objects of type `POINT` may not occupy 3 contiguous `REAL` storage locations.

Fortran 95: Derived types

A new derived type can contain another derived type as one of its components; the derived type of the components must have already been declared or must be the type currently being declared; e.g.,

```
TYPE SPHERE
  TYPE(POINT) :: CENTER
  REAL :: RADIUS
END TYPE SPHERE
TYPE(SPHERE) :: BUBBLE
BUBBLE%RADIUS = 1.0
BUBBLE%CENTER%X = 0.2
BUBBLE%CENTER%Y = 0.4
BUBBLE%CENTER%Z = 0.6
BUBBLE = SPHERE( POINT(0.2,0.4,0.6), 1.0 )
```

Derived objects can be used in I/O statements similarly to intrinsic objects; so the statement

```
PRINT*, BUBBLE
```

is equivalent to

```
PRINT*, BUBBLE%CENTER%X, BUBBLE%CENTER%Y, BUBBLE%CENTER%Z, &
              %BUBBLE%RADIUS
```

Fortran 95: Derived types

We can have arrays of derived-type objects; e.g.,

```
TYPE(POINT), DIMENSION(4) :: TETRAHEDRON
```

Derived types can contain array components; e.g.,

```
TYPE PNT
  REAL, DIMENSION(3) :: X
END TYPE PNT
TYPE VOLUME
  TYPE(POINT), DIMENSION(4) :: TETRAHEDRON
  INTEGER :: LABEL
END TYPE VOLUME
TYPE(VOLUME), DIMENSION(100) :: DIAMOND
```

`DIAMOND` is an object of type `VOLUME`.

Geometrically, a diamond has many facets, and we can conceptually create it by adjoining tetrahedra.

Each tetrahedron is described by its four corner points, and each corner point is described by its set of Cartesian coordinates (x_1, x_2, x_3).

Fortran 95: Derived types

We can refer to a specific coordinate of one of the node points; e.g.,

```
diamond(5)%tetrahedron(2)%x(1)
```

refers to tetrahedron 5, node point 2, coordinate 1.

We can also refer to a subsection of the array component, provided that there is only one non-scalar index in the reference; e.g.,

```
diamond(:)%tetrahedron(2)%x(1)
diamond(5)%tetrahedron(:)%x(1)
diamond(5)%tetrahedron(2)%x(:)
```

are all correct; however,

```
diamond(:)%tetrahedron(:)%x(1)
diamond(5)%tetrahedron(:)%x(:)
diamond(:)%tetrahedron(2)%x(:)
```

are incorrect because we can only section at most one component at a time.

Fortran 95: Derived types

Derived type objects can be passed as arguments pretty much as intrinsic objects but with some caveats.

They can be given attributes (`OPTIONAL`, `INTENT`, `DIMENSION`, `SAVE`, `ALLOCATABLE`, etc.).

But two types cannot be declared in two different places (e.g., in the main program and in a function) even if they look the same because they would have different scopes.

For further information, see the `USE` statement.

Placing a `USE` statement in the module definition allows it to be used from the main program.

Fortran 95: Pointers and targets

Unlike C pointers, F90/95 pointers are much less flexible but more highly optimized.

The space to which a Fortran pointer points is called a target.

Some things to note:

- Pointers are strongly typed.
- Any variable that is pointed at must have the TARGET attribute.
- Pointers are automatically dereferenced (pointer and target refer to same memory location).
- Target address cannot be printed.

Pointers in Fortran provide a more flexible alternative to allocatable arrays and allow the creation and manipulation of linked lists.

Fortran 95: Pointers and targets

Sample pointer declarations:

```
REAL, POINTER :: PtoR, PtoR2
REAL, DIMENSION(:,:), POINTER :: PtoA
```

Sample target declarations:

```
real,target :: x, y
real, dimension(5,3),target :: a, b
real, dimension(4,7),target :: c, d
```

`x`, `y` may become associated with `PtoR`, whereas `a`, `b`, `c` ,`d` may become associated with `PtoA`.

TARGETs are used only for optimization. The compiler assumes any non-pointer object not explicitly declared as a TARGET is only referred to by its original name.

Fortran 95: Pointers and targets

Pointer assignment takes place between a pointer variable and a target variable, or between two pointer variables.

```
PtoR => y
```

Pointer `PtoR` is associated with the target `y`; i.e., `PtoR` becomes an alias for `y`.

```
PtoA => b
```

Pointer `PtoA` is associated with the target `b`.

```
PtoR2 => PtoR
```

Pointer `PtoR2` is associated with the target of the pointer `PtoR`; i.e., it is associated with `y`. So now both `PtoR` and `PtoR2` are aliases for `y`.

This statement is OK because all pointer variables implicitly have the `TARGET` attribute (`PtoR` is a target).

Fortran 95: Pointers and targets

Note the difference between pointer assignment (=>), which makes the pointer and the target variables reference the same space, and (ordinary) assignment (=), which alters the value in the space referred to by the LHS.

For example,

```
x=3.0
PtoR => y ! pointer assignment
PtoR = x ! y = x
```

The last statement effectively sets `y` to 3.

Pointers in an (ordinary) assignment are automatically dereferenced; thus, `PtoR` is effectively an alias for `y`.

Fortran 95: Elements of OOP

Routines in Fortran 95 can be called with keyword arguments and can use default arguments.

That is, some arguments can be given with keywords instead of their position

Furthermore, some arguments do not have to be given at all, in which case a standard or default value is used.

Fortran 95: Elements of OOP

```
MODULE D3
!
TYPE COORDS
PRIVATE
REAL :: X,Y,Z
END TYPE COORDS
!
CONTAINS
!
TYPE(COORDS) FUNCTION INIT_COORDS(X,Y,Z)
REAL, INTENT(IN), OPTIONAL :: X,Y,Z
INIT_COORDS = COORDS(0.0,0.0,0.0)
IF(PRESENT(X)) INIT_COORDS%X = X
IF(PRESENT(Y)) INIT_COORDS%Y = Y
IF(PRESENT(Z)) INIT_COORDS%Z = Z
END FUNCTION INIT_COORDS
!
SUBROUTINE PRINT_COORDS(C)
TYPE(COORDS), INTENT(IN) :: C
PRINT*, C%X,C%Y,C%Z
END SUBROUTINE PRINT_COORDS
!
end module d3
```

Note that the components of a `COORDS` type object are not visible to the user; they can only be accessed through the functions contained in the module `D3`.

Furthermore, the components can only be printed from the module procedure `PRINT_COORDS`.

Fortran 95: Elements of OOP

Procedures can be (unambiguously!) overloaded.

The compiler decides (at compile time) which procedure to use based on the signature (type, number, etc.) of the non-optional arguments.

```
MODULE GI
INTERFACE PLUS1
  MODULE PROCEDURE IPLUS1
  MODULE PROCEDURE RPLUS1
  MODULE PROCEDURE DPLUS1
END INTERFACE ! PLUS1
CONTAINS
INTEGER FUNCTION IPLUS1(X)
  INTEGER, INTENT(IN) :: X
  IPLUS1 = X + 1
END FUNCTION IPLUS1
REAL FUNCTION RPLUS1(X)
  REAL, INTENT(IN) :: X
  RPLUS1 = X + 1.0
END FUNCTION RPLUS1
DOUBLE PRECISION FUNCTION DPLUS1(X)
  DOUBLE PRECISION, INTENT(IN) :: X
  DPLUS1 = X + 1.0D0
END FUNCTION DPLUS1
END MODULE GI
```

Thus, the call `PLUS1(2)` returns an `INTEGER` result, whereas `PLUS1(2.0)` returns a `REAL` result.

Fortran 95: Measuring performance

Fortran 95 has the intrinsic subroutine

```
SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])
```

that gives the clock ticks since some point in the past, modulo `COUNT_MAX`, at `COUNT_RATE` ticks per second.

Summary

- A brief overview of Fortran 77
- A brief overview of Fortran 95