

# Fortran 90 Basics

*I don't know what the programming language  
of the year 2000 will look like, but I know it  
will be called FORTRAN.*

*Charles Anthony Richard Hoare*

# F90 Program Structure

- A Fortran 90 program has the following form:
  - *program-name* is the name of that program
  - *specification-part*, *execution-part*, and *subprogram-part* are optional.
  - Although **IMPLICIT NONE** is also optional, this is required in this course to write safe programs.

```
PROGRAM program-name  
IMPLICIT NONE  
[specification-part]  
[execution-part]  
[subprogram-part]  
END PROGRAM program-name
```

# Program Comments

- Comments start with a **!**
- Everything following **!** will be ignored
- This is similar to **//** in C/C++

```
! This is an example
!  
  
PROGRAM Comment  
.....  
READ(*,*) Year    ! read in the value of Year  
.....  
Year = Year + 1   ! add 1 to Year  
.....  
END PROGRAM Comment
```

# Continuation Lines

- Fortran 90 is not completely format-free!
- A statement must start with a new line.
- If a statement is too long to fit on one line, it has to be *continued*.
- The continuation character is **&**, which is not part of the statement.

```
Total = Total + &  
                Amount * Payments  
! Total = Total + Amount*Payments  
  
PROGRAM &  
    ContinuationLine  
! PROGRAM ContinuationLine
```

# Alphabets

- **Fortran 90 alphabets include the following:**
  - **Upper and lower cases letters**
  - **Digits**
  - **Special characters**

**space**

**' "**

**( ) \* + - / : =**

**\_ ! & \$ ; < >**

**% ? , .**

## Constants: 1/6

- A Fortran 90 constant may be an integer, real, logical, complex, and character string.
- We will not discuss complex constants.
- An **integer constant** is a string of digits with an optional sign: **12345**, **-345**, **+789**, **+0**.

## Constants: 2/6

- A **real constant** has two forms, **decimal** and **exponential**:
  - In the **decimal form**, a real constant is a string of digits with exactly one decimal point. A real constant may include an optional sign. Example: **2.45**, **.13**, **13.**, **-0.12**, **-.12**.

# Constants: 3/6

- A **real constant** has two forms, **decimal** and **exponential**:
  - In the **exponential** form, a real constant starts with an integer/real, followed by a **E/e**, followed by an integer (*i.e.*, the exponent).

Examples:

◆ **12E3** ( $12 \times 10^3$ ), **-12e3** ( $-12 \times 10^3$ ),  
**3.45E-8** ( $3.45 \times 10^{-8}$ ), **-3.45e-8**  
( $-3.45 \times 10^{-8}$ ).

◆ **0E0** ( $0 \times 10^0 = 0$ ). **12.34-5 is wrong!**

## Constants: 4/6

- A logical constant is either **.TRUE.** or **.FALSE.**
- Note that the periods surrounding **TRUE** and **FALSE** are required!

## Constants: 5/6

- A **character string** or **character constant** is a string of characters enclosed between two double quotes or two single quotes. Examples: `"abc"`, `'John Dow'`, `"#$%^"`, and `'( )'`.
- The content of a character string consists of all characters between the quotes. Example: The content of `'John Dow'` is `John Dow`.
- The length of a string is the number of characters between the quotes. The length of `'John Dow'` is 8, space included.

## Constants: 6/6

- A string has length zero (*i.e.*, no content) is an **empty string**.
- If single (or double) quotes are used in a string, then use double (or single) quotes as delimiters. Examples: `"Adam's cat"` and `'I said "go away"'`.
- Two consecutive quotes are treated as one!  
`'Lori''s Apple'` is `Lori's Apple`  
`"double quote"""` is `double quote"`  
`'abc''def"x''y'` is `abc'def"x'y`  
`"abc""def'x""y"` is `abc"def'x"y`

# Identifiers: 1/2

- A Fortran 90 identifier can have no more than 31 characters.
- The first one must be a letter. The remaining characters, if any, may be letters, digits, or underscores.
- Fortran 90 identifiers are CASE INSENSITIVE.
- Examples: **A**, **Name**, **tOTAL123**, **System\_**, **myFile\_01**, **my\_1st\_F90\_program\_X\_**.
- Identifiers **Name**, **nAmE**, **naME** and **NaME** are the same.

## Identifiers: 2/2

- Unlike Java, C, C++, etc, *Fortran 90 does not have reserved words*. This means one may use Fortran keywords as identifiers.
- Therefore, **PROGRAM**, **end**, **IF**, **then**, **DO**, etc may be used as identifiers. Fortran 90 compilers are able to recognize keywords from their “**positions**” in a statement.
- Yes, **end = program + if/(goto - while)** is legal!
- However, avoid the use of Fortran 90 keywords as identifiers to minimize confusion.

# Declarations: 1/3

- Fortran 90 uses the following for variable declarations, where **type-specifier** is one of the following keywords: **INTEGER**, **REAL**, **LOGICAL**, **COMPLEX** and **CHARACTER**, and **list** is a sequence of identifiers separated by commas.

**type-specifier** :: **list**

- Examples:

```
INTEGER :: Zip, Total, counter  
REAL    :: AVERAGE, x, Difference  
LOGICAL :: Condition, OK  
COMPLEX :: Conjugate
```

# Declarations: 2/3

- Character variables require additional information, the *string length*:
  - Keyword **CHARACTER** must be followed by a length attribute (**LEN = *l***), where *l* is the length of the string.
  - The **LEN=** part is optional.
  - If the length of a string is 1, one may use **CHARACTER** without length attribute.
  - Other length attributes will be discussed later.

# Declarations: 3/3

## ● Examples:

■ **CHARACTER (LEN=20) :: Answer, Quote**

Variables **Answer** and **Quote** can hold strings up to 20 characters.

■ **CHARACTER (20) :: Answer, Quote** is the same as above.

■ **CHARACTER :: Keypress** means variable **Keypress** can only hold **ONE** character (*i.e.*, length 1).

## The **PARAMETER** Attribute: 1/4

- A **PARAMETER** identifier is a name whose value cannot be modified. In other words, it is a *named constant*.
- The **PARAMETER** attribute is used after the type keyword.
- Each identifier is followed by a **=** and followed by a value for that identifier.

```
INTEGER, PARAMETER :: MAXIMUM = 10  
REAL, PARAMETER    :: PI = 3.1415926, E = 2.17828  
LOGICAL, PARAMETER :: TRUE = .true., FALSE = .false.
```

## The **PARAMETER** Attribute: 2/4

- Since **CHARACTER** identifiers have a length attribute, it is a little more complex when used with **PARAMETER**.
- Use (**LEN = \***) if one does not want to count the number of characters in a **PARAMETER** character string, where **= \*** means the length of this string is determined elsewhere.

```
CHARACTER (LEN=3), PARAMETER :: YES = "yes" ! Len = 3
CHARACTER (LEN=2), PARAMETER :: NO = "no" ! Len = 2
CHARACTER (LEN=*), PARAMETER :: &
                                PROMPT = "What do you want?" ! Len = 17
```

# The **PARAMETER** Attribute: 3/4

- Since Fortran 90 strings are of *fixed* length, one must remember the following:
  - If a string is longer than the **PARAMETER** length, the right end is truncated.
  - If a string is shorter than the **PARAMETER** length, spaces will be added to the right.

```
CHARACTER (LEN=4), PARAMETER :: ABC = "abcdef"  
CHARACTER (LEN=4), PARAMETER :: XYZ = "xy"
```

ABC = 

a	b	c	d
---	---	---	---

XYZ = 

x	y		
---	---	--	--

## The **PARAMETER** Attribute: 4/4

- By convention, **PARAMETER** identifiers use all upper cases. However, this is not mandatory.
- For maximum flexibility, constants other than 0 and 1 should be **PARAMETER**ized.
- A **PARAMETER** is an alias of a value and is not a variable. Hence, one cannot modify the content of a **PARAMETER** identifier.
- One can may a **PARAMETER** identifier anywhere in a program. It is equivalent to replacing the identifier with its value.
- The value part can use expressions.

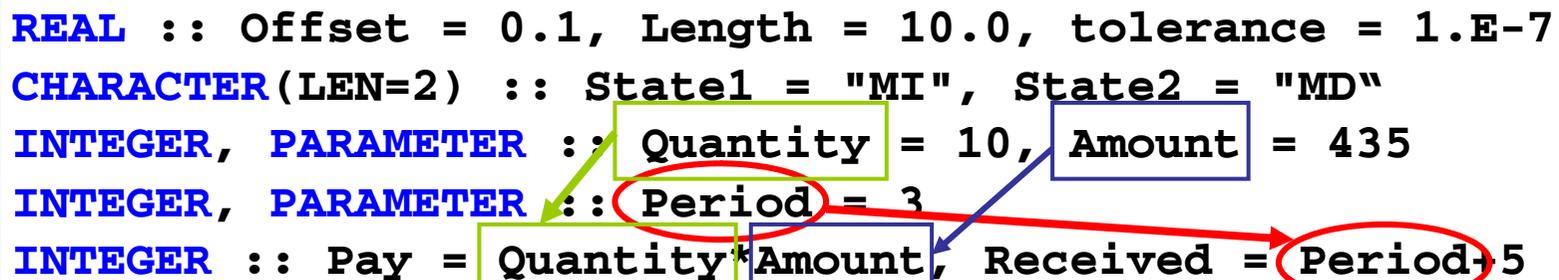
# Variable Initialization: 1/2

- A variable receives its value with
  - ***Initialization***: It is done once before the program runs.
  - ***Assignment***: It is done when the program executes an assignment statement.
  - ***Input***: It is done with a **READ** statement.

## Variable Initialization: 2/2

- Variable initialization is very similar to what we learned with **PARAMETER**.
- A variable name is followed by a **=**, followed by an expression in which all identifiers must be constants or **PARAMETERS** defined *previously*.
- Using an un-initialized variable may cause unexpected, sometimes disastrous results.

```
REAL :: Offset = 0.1, Length = 10.0, tolerance = 1.E-7
CHARACTER(LEN=2) :: State1 = "MI", State2 = "MD"
INTEGER, PARAMETER :: Quantity = 10, Amount = 435
INTEGER, PARAMETER :: Period = 3
INTEGER :: Pay = Quantity*Amount, Received = Period+5
```



# Arithmetic Operators

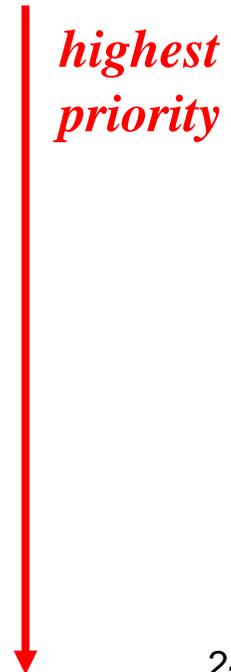
- There are four types of operators in Fortran 90: arithmetic, relational, logical and character.
- The following shows the first three types:

<i>Type</i>	<i>Operator</i>						<i>Associativity</i>
<b>Arithmetic</b>	**						<u>right to left</u>
	*			/			left to right
	+			-			left to right
<b>Relational</b>	<	<=	>	>=	==	/=	none
<b>Logical</b>	.NOT.						<u>right to left</u>
	.AND.						left to right
	.OR.						left to right
	.EQV.			.NEQV.			left to right

# Operator Priority

- **\*\*** is the highest; **\*** and **/** are the next, followed by **+** and **-**. All relational operators are next.
- Of the 5 logical operators, **.EQV.** and **.NEQV.** are the lowest.

<i>Type</i>	<i>Operator</i>						<i>Associativity</i>
Arithmetic	**						<i>right to left</i>
	*			/			left to right
	+			-			left to right
Relational	<	<=	>	>=	==	/=	none
Logical	.NOT.						<i>right to left</i>
	.AND.						left to right
	.OR.						left to right
	.EQV.			.NEQV.			left to right



# Expression Evaluation

- Expressions are evaluated from left to right.
- If an operator is encountered in the process of evaluation, its priority is compared with that of the next one
  - if the next one is **lower**, evaluate the current operator with its operands;
  - if the next one is **equal** to the current, the associativity laws are used to determine which one should be evaluated;
  - if the next one is **higher**, scanning continues

# Single Mode Expression

- A *single mode* arithmetic expression is an expression all of whose operands are of the same type.
- If the operands are **INTEGERS** (*resp.*, **REALS**), the result is also an **INTEGER** (*resp.*, **REAL**).

```
1.0 + 2.0 * 3.0 / ( 6.0*6.0 + 5.0*44.0) ** 0.25
--> 1.0 + 6.0 / (6.0*6.0 + 5.0*44.0) ** 0.25
--> 1.0 + 6.0 / (36.0 + 5.0*44.0) ** 0.25
--> 1.0 + 6.0 / (36.0 + 220.0) ** 0.25
--> 1.0 + 6.0 / 256.0 ** 0.25
--> 1.0 + 6.0 / 4.0
--> 1.0 + 1.5
--> 2.5
```

# Mixed Mode Expression: 1/2

- If operands have different types, it is *mixed mode*.
- **INTEGER** and **REAL** yields **REAL**, and the **INTEGER** operand is converted to **REAL** before evaluation. Example: **3.5\*4** is converted to **3.5\*4.0** becoming single mode.
- Exception: **x\*\*INTEGER**: **x\*\*3** is **x\*x\*x** and **x\*\*(-3)** is **1.0/(x\*x\*x)**.
- **x\*\*REAL** is evaluated with **log()** and **exp()**.
- Logical and character cannot be mixed with arithmetic operands.

# Mixed Mode Expression: 2/2

- Note that  $a^{**}b^{**}c$  is  $a^{**}(b^{**}c)$  instead of  $(a^{**}b)^{**}c$ , and  $a^{**}(b^{**}c) \neq (a^{**}b)^{**}c$ .  
This can be a big trap!

```
5 * (11.0 - 5) ** 2 / 4 + 9
--> 5 * (11.0 - 5.0) ** 2 / 4 + 9
--> 5 * 6.0 ** 2 / 4 + 9
--> 5 * 36.0 / 4 + 9
--> 5.0 * 36.0 / 4 + 9
--> 180.0 / 4 + 9
--> 180.0 / 4.0 + 9
--> 45.0 + 9
--> 45.0 + 9.0
--> 54.0
```

$6.0^{**}2$  is evaluated as  $6.0 * 6.0$   
rather than converted to  $6.0^{**}2.0$ !

red: type conversion

# The Assignment Statement: 1/2

- The assignment statement has a form of **variable = expression**
- If the type of **variable** and **expression** are identical, the result is saved to **variable**.
- If the type of **variable** and **expression** are not identical, the result of **expression** is converted to the type of **variable**.
- If **expression** is **REAL** and **variable** is **INTEGER**, the result is truncated.

# The Assignment Statement: 2/2

- The left example uses an initialized variable **Unit**, and the right uses a **PARAMETER PI**.

```
INTEGER :: Total, Amount
INTEGER :: Unit = 5

Amount = 100.99
Total = Unit * Amount
```

```
REAL, PARAMETER :: PI = 3.1415926
REAL :: Area
INTEGER :: Radius

Radius = 5
Area = (Radius ** 2) * PI
```

This one is equivalent to `Radius ** 2 * PI`

# Fortran Intrinsic Functions: 1/4

- Fortran provides many commonly used functions, referred to as *intrinsic functions*.
- To use an intrinsic function, we need to know:
  - Name and meaning of the function (*e.g.*, **SQRT ( )** for square root)
  - Number of arguments
  - The type and range of each argument (*e.g.*, the argument of **SQRT ( )** must be non-negative)
  - The type of the returned function value.

# Fortran Intrinsic Functions: 2/4

## ● Some mathematical functions:

<i>Function</i>	<i>Meaning</i>	<i>Arg. Type</i>	<i>Return Type</i>
<b>ABS (x)</b>	absolute value of <b>x</b>	<b>INTEGER</b>	<b>INTEGER</b>
		<b>REAL</b>	<b>REAL</b>
<b>SQRT (x)</b>	square root of <b>x</b>	<b>REAL</b>	<b>REAL</b>
<b>SIN (x)</b>	sine of <b>x</b> radian	<b>REAL</b>	<b>REAL</b>
<b>COS (x)</b>	cosine of <b>x</b> radian	<b>REAL</b>	<b>REAL</b>
<b>TAN (x)</b>	tangent of <b>x</b> radian	<b>REAL</b>	<b>REAL</b>
<b>ASIN (x)</b>	arc sine of <b>x</b>	<b>REAL</b>	<b>REAL</b>
<b>ACOS (x)</b>	arc cosine of <b>x</b>	<b>REAL</b>	<b>REAL</b>
<b>ATAN (x)</b>	arc tangent of <b>x</b>	<b>REAL</b>	<b>REAL</b>
<b>EXP (x)</b>	exponential $e^x$	<b>REAL</b>	<b>REAL</b>
<b>LOG (x)</b>	natural logarithm of <b>x</b>	<b>REAL</b>	<b>REAL</b>

**LOG10 (x)** is the common logarithm of **x**!

# Fortran Intrinsic Functions: 3/4

## ● Some conversion functions:

<i>Function</i>	<i>Meaning</i>	<i>Arg. Type</i>	<i>Return Type</i>
<b>INT (x)</b>	truncate to integer part <b>x</b>	<b>REAL</b>	<b>INTEGER</b>
<b>NINT (x)</b>	round nearest integer to <b>x</b>	<b>REAL</b>	<b>INTEGER</b>
<b>FLOOR (x)</b>	greatest integer less than or equal to <b>x</b>	<b>REAL</b>	<b>INTEGER</b>
<b>FRACTION (x)</b>	the fractional part of <b>x</b>	<b>REAL</b>	<b>REAL</b>
<b>REAL (x)</b>	convert <b>x</b> to <b>REAL</b>	<b>INTEGER</b>	<b>REAL</b>

### Examples:

```
INT(-3.5) → -3  
NINT(3.5) → 4  
NINT(-3.4) → -3  
FLOOR(3.6) → 3  
FLOOR(-3.5) → -4  
FRACTION(12.3) → 0.3  
REAL(-10) → -10.0
```

# Fortran Intrinsic Functions: 4/4

## ● Other functions:

<i>Function</i>	<i>Meaning</i>	<i>Arg. Type</i>	<i>Return Type</i>
MAX( <i>x</i> <sub>1</sub> , <i>x</i> <sub>2</sub> , ..., <i>x</i> <sub><i>n</i></sub> )	maximum of <i>x</i> <sub>1</sub> , <i>x</i> <sub>2</sub> , ... <i>x</i> <sub><i>n</i></sub>	INTEGER	INTEGER
		REAL	REAL
MIN( <i>x</i> <sub>1</sub> , <i>x</i> <sub>2</sub> , ..., <i>x</i> <sub><i>n</i></sub> )	minimum of <i>x</i> <sub>1</sub> , <i>x</i> <sub>2</sub> , ... <i>x</i> <sub><i>n</i></sub>	INTEGER	INTEGER
		REAL	REAL
MOD( <i>x</i> , <i>y</i> )	remainder $x - \text{INT}(x/y) * y$	INTEGER	INTEGER
		REAL	REAL

# Expression Evaluation

- Functions have the highest priority.
- Function arguments are evaluated first.
- The returned function value is treated as a value in the expression.

```
REAL :: A = 1.0, B = -5.0, C = 6.0, R
```

```
R = (-B + SQRT(B*B - 4.0*A*C)) / (2.0*A)
```

R gets 3.0

```
(-B + SQRT(B*B - 4.0*A*C)) / (2.0*A)
--> (5.0 + SQRT(B*B - 4.0*A*C)) / (2.0*A)
--> (5.0 + SQRT(25.0 - 4.0*A*C)) / (2.0*A)
--> (5.0 + SQRT(25.0 - 4.0*C)) / (2.0*A)
--> (5.0 + SQRT(25.0 - 24.0)) / (2.0*A)
--> (5.0 + SQRT(1.0)) / (2.0*A)
--> (5.0 + 1.0) / (2.0*A)
--> 6.0 / (2.0*A)
--> 6.0 / 2.0
--> 3.0
```

## What is **IMPLICIT NONE**?

- Fortran has an interesting tradition: all variables starting with **i, j, k, l, m** and **n**, if not declared, are of the **INTEGER** type by default.
- This handy feature can cause serious consequences if it is not used with care.
- **IMPLICIT NONE** means all names must be declared and there is no implicitly assumed **INTEGER** type.
- All programs in this class must use **IMPLICIT NONE**. *Points will be deducted if you do not use it!*

## List-Directed **READ**: 1/5

- Fortran 90 uses the **READ(\*,\*)** statement to read data into variables from keyboard:

```
READ(*,*) v1, v2, ..., vn
```

```
READ(*,*)
```

- The second form has a special meaning that will be discussed later.

```
INTEGER           :: Age  
REAL              :: Amount, Rate  
CHARACTER(LEN=10) :: Name  
  
READ(*,*) Name, Age, Rate, Amount
```

# List-Directed **READ**: 2/5

## ● Data Preparation Guidelines

- **READ ( \* , \* )** reads data from keyboard by default, although one may use input redirection to read from a file.
- If **READ ( \* , \* )** has *n* variables, there must be *n* Fortran constants.
- Each constant must have the type of the corresponding variable. Integers can be read into **REAL** variables but not vice versa.
- Data items are separated by spaces and may spread into multiple lines.

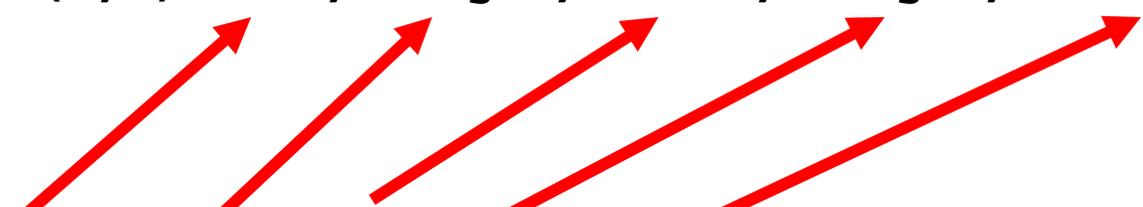
## List-Directed **READ**: 3/5

- *The execution of **READ (\*, \*)** always starts with a new line!*
- Then, it reads each constant into the corresponding variable.

```
CHARACTER (LEN=5) :: Name
REAL              :: height, length
INTEGER          :: count, MaxLength

READ(*,*) Name, height, count, length, MaxLength
```

**Input:** "Smith" 100.0 25 123.579 10000



The diagram illustrates the mapping of input values to variables. Five red arrows originate from the input line and point to the variable names in the READ statement above. The first arrow points from "Smith" to Name, the second from 100.0 to height, the third from 25 to count, the fourth from 123.579 to length, and the fifth from 10000 to MaxLength.

# List-Directed **READ**: 4/5

- Be careful when input items are on multiple lines.

```
INTEGER :: I, J, K, L, M, N
```

**Input:**

```
READ(*,*) I, J ← 100 200
READ(*,*) K, L, M ← 300 400 500
READ(*,*) N ← 600
```

```
INTEGER :: I, J, K, L, M, N
```

```
READ(*,*) I, J, K ← 100 200 300 400
READ(*,*) L, M, N ← 500 600 700 800
900
```

*ignored!*

**READ(\*,\*)** always starts with a new line

# List-Directed **READ**: 5/5

- Since **READ (\*, \*)** always starts with a new line, a **READ (\*, \*)** without any variable means skipping the input line!

```
INTEGER :: P, Q, R, S
```

<code>READ (*, *) P, Q</code>	←	100	200	300
<code>READ (*, *)</code>	←	400	500	600
<code>READ (*, *) R, S</code>	←	700	800	900

## List-Directed WRITE: 1/3

- Fortran 90 uses the **WRITE (\*, \*)** statement to write information to screen.
- **WRITE (\*, \*)** has two forms, where **exp1**, **exp2**, ..., **expn** are expressions  
**WRITE (\*, \*) exp1, exp2, ..., expn**  
**WRITE (\*, \*)**
- **WRITE (\*, \*)** evaluates the result of each expression and prints it on screen.
- **WRITE (\*, \*)** *always starts with a new line!*

# List-Directed WRITE: 2/3

- Here is a simple example:

```
INTEGER :: Target
REAL    :: Angle, Distance
CHARACTER(LEN=*) , PARAMETER ::
  Time = "The time to hit target ",
  IS = " is ",
  UNIT = " sec."
```

means length is determined by actual count

```
Target = 10
Angle = 20.0
Distance = 1350.0
WRITE(*,*) 'Angle = ', Angle
WRITE(*,*) 'Distance = ', Distance
WRITE(*,*)
WRITE(*,*) Time, Target, IS,
  Angle * Distance, UNIT
```

continuation lines

Output:

```
Angle = 20.0
Distance = 1350.0
The time to hit target 10 is 27000.0 sec.
```

print a blank line

## List-Directed WRITE: 3/3

- The previous example used **LEN=\*** , which means the length of a **CHARACTER** constant is determined by actual count.
- **WRITE (\*, \*)** without any expression advances to the next line, producing a blank one.
- A Fortran 90 compiler will use the *best* way to print each value. Thus, indentation and alignment are difficult to achieve with **WRITE (\*, \*)**.
- One must use the **FORMAT** statement to produce good looking output.

# Complete Example: 1/4

- This program computes the position ( $x$  and  $y$  coordinates) and the velocity (magnitude and direction) of a projectile, given  $t$ , the time since launch,  $u$ , the launch velocity,  $a$ , the initial angle of launch (in degree), and  $g=9.8$ , the acceleration due to gravity.
- The horizontal and vertical displacements,  $x$  and  $y$ , are computed as follows:

$$x = u \times \cos(a) \times t$$
$$y = u \times \sin(a) \times t - \frac{g \times t^2}{2}$$

## Complete Example: 2/4

- **The horizontal and vertical components of the velocity vector are computed as**

$$V_x = u \times \cos(a)$$

$$V_y = u \times \sin(a) - g \times t$$

- **The magnitude of the velocity vector is**

$$V = \sqrt{V_x^2 + V_y^2}$$

- **The angle between the ground and the velocity vector is**

$$\tan(\theta) = \frac{V_x}{V_y}$$

# Complete Example: 3/4

- Write a program to read in the launch angle  $a$ , the time since launch  $t$ , and the launch velocity  $u$ , and compute the position, the velocity and the angle with the ground.

```
PROGRAM Projectile
  IMPLICIT NONE
  REAL, PARAMETER :: g = 9.8           ! acceleration due to gravity
  REAL, PARAMETER :: PI = 3.1415926   ! you know this. don't you?
  REAL :: Angle                       ! launch angle in degree
  REAL :: Time                         ! time to flight
  REAL :: Theta                       ! direction at time in degree
  REAL :: U                           ! launch velocity
  REAL :: V                           ! resultant velocity
  REAL :: Vx                          ! horizontal velocity
  REAL :: Vy                          ! vertical velocity
  REAL :: X                           ! horizontal displacement
  REAL :: Y                           ! vertical displacement
  ..... Other executable statements .....
END PROGRAM Projectile
```

# Complete Example: 4/4

- Write a program to read in the launch angle  $a$ , the time since launch  $t$ , and the launch velocity  $u$ , and compute the position, the velocity and the angle with the ground.

```
READ(*,*) Angle, Time, U

Angle = Angle * PI / 180.0           ! convert to radian
X      = U * COS(Angle) * Time
Y      = U * SIN(Angle) * Time - g*Time*Time / 2.0
Vx     = U * COS(Angle)
Vy     = U * SIN(Angle) - g * Time
V      = SQRT(Vx*Vx + Vy*Vy)
Theta  = ATAN(Vy/Vx) * 180.0 / PI ! convert to degree

WRITE(*,*) 'Horizontal displacement : ', X
WRITE(*,*) 'Vertical displacement   : ', Y
WRITE(*,*) 'Resultant velocity      : ', V
WRITE(*,*) 'Direction (in degree)   : ', Theta
```

# CHARACTER Operator //

- Fortran 90 uses `//` to concatenate two strings.
- If strings **A** and **B** have lengths  $m$  and  $n$ , the concatenation **A** `//` **B** is a string of length  $m+n$ .

```
CHARACTER(LEN=4) :: John = "John", Sam = "Sam"
CHARACTER(LEN=6) :: Lori = "Lori", Reagan = "Reagan"
CHARACTER(LEN=10) :: Ans1, Ans2, Ans3, Ans4

Ans1 = John // Lori           ! Ans1 = "JohnLori[]"
Ans2 = Sam // Reagan          ! Ans2 = "SamReagan"
Ans3 = Reagan // Sam          ! Ans3 = "ReaganSam"
Ans4 = Lori // Sam            ! Ans4 = "LoriSam"
```

# CHARACTER Substring: 1/3

- A consecutive portion of a string is a *substring*.
- To use substrings, one may add an *extent specifier* to a **CHARACTER** variable.
- An extent specifier has the following form:  
( **integer-exp1** : **integer-exp2** )
- The first and the second expressions indicate the start and end: ( **3 : 8** ) means 3 to 8,
- If **A = "abcdefg"** , then **A(3 : 5)** means **A**'s substring from position 3 to position 5 (*i.e.*, **"cde"** ).

## CHARACTER Substring: 2/3

- In `(integer-exp1:integer-exp2)`, if the first `exp1` is missing, the substring starts from the first character, and if `exp2` is missing, the substring ends at the last character.
- If `A = "12345678"`, then `A(:5)` is `"12345"` and `A(3+x:)` is `"5678"` where `x` is `2`.
- As a good programming practice, in general, the first expression `exp1` should be no less than `1`, and the second expression `exp2` should be no greater than the length of the string.

# CHARACTER Substring: 3/3

- Substrings can be used on either side of the assignment operator.
- Suppose `LeftHand = "123456789"` (length is 10).
  - `LeftHand(3:5) = "abc"` yields `LeftHand = "12abc67890"`
  - `LeftHand(4:) = "lmnopqr"` yields `LeftHand = "123lmnopqr"`
  - `LeftHand(3:8) = "abc"` yields `LeftHand = "12abc□□□90"`
  - `LeftHand(4:7) = "lmnopq"` yields `LeftHand = "123lmno890"`

## Example: 1/5

- This program uses the **DATE\_AND\_TIME ( )** Fortran 90 intrinsic function to retrieve the system date and system time. Then, it converts the date and time information to a readable format. This program demonstrates the use of concatenation operator **//** and substring.
- System date is a string **ccyyymmdd**, where **cc** – century, **yy** = year, **mm** = month, and **dd** = day.
- System time is a string **hhmms.ss**, where **hh** = hour, **mm** = minute, and **ss.ss** = second.

## Example: 2/5

- The following shows the specification part.  
Note the handy way of changing string length.

```
PROGRAM DateTime
  IMPLICIT NONE
  CHARACTER(LEN = 8)   :: DateINFO           ! ccyymmdd
  CHARACTER(LEN = 4)   :: Year, Month*2, Day*2
  CHARACTER(LEN = 10)  :: TimeINFO, PrettyTime*12 ! hhmmss.sss
  CHARACTER(LEN = 2)   :: Hour, Minute, Second*6

  CALL DATE_AND_TIME(DateINFO, TimeINFO)
  ..... other executable statements .....
END PROGRAM DateTime
```

This is a handy way of changing string length

## Example: 3/5

- Decompose **DateINFO** into year, month and day. **DateINFO** has a form of **ccyyymmdd**, where **cc** = century, **yy** = year, **mm** = month, and **dd** = day.

```
Year  = DateINFO(1:4)
Month = DateINFO(5:6)
Day   = DateINFO(7:8)
WRITE(*,*) 'Date information -> ', DateINFO
WRITE(*,*) '          Year -> ', Year
WRITE(*,*) '          Month -> ', Month
WRITE(*,*) '          Day -> ', Day
```

**Output:**

```
Date information -> 19970811
          Year -> 1997
          Month -> 08
          Day -> 11
```

# Example: 4/5

## ● Now do the same for time:

```
Hour          = TimeINFO(1:2)
Minute        = TimeINFO(3:4)
Second        = TimeINFO(5:10)
PrettyTime    = Hour // ':' // Minute // ':' // Second
WRITE(*,*)
WRITE(*,*) 'Time Information -> ', TimeINFO
WRITE(*,*) ' Hour          -> ', Hour
WRITE(*,*) ' Minute        -> ', Minute
WRITE(*,*) ' Second        -> ', Second
WRITE(*,*) ' Pretty Time   -> ', PrettyTime
```

**Output:**

```
Time Information -> 010717.620
                Hour   -> 01
                Minute -> 07
                Second -> 17.620
                Pretty Time -> 01:07:17.620
```

## Example: 5/5

- We may also use substring to achieve the same result:

```
PrettyTime = " " ! Initialize to all blanks
PrettyTime( :2) = Hour
PrettyTime(3:3) = ':'
PrettyTime(4:5) = Minute
PrettyTime(6:6) = ':'
PrettyTime(7: ) = Second

WRITE(*,*)
WRITE(*,*) ' Pretty Time -> ', PrettyTime
```

# What **KIND** Is It?

- Fortran 90 has a **KIND** attribute for selecting the precision of a numerical constant/variable.
- The **KIND** of a constant/variable is a positive integer (more on this later) that can be attached to a constant.
- Example:
  - **126\_3** : **126** is an integer of **KIND 3**
  - **3.1415926\_8** : **3.1415926** is a real of **KIND 8**

## What **KIND** Is It (**INTEGER**)? 1/2

- Function **SELECTED\_INT\_KIND**( $k$ ) selects the **KIND** of an integer, where the value of  $k$ , a positive integer, means the selected integer **KIND** has a value between  $-10^k$  and  $10^k$ .
- Thus, the value of  $k$  is approximately the number of digits of that **KIND**. For example, **SELECTED\_INT\_KIND**(10) means an integer **KIND** of no more than 10 digits.
- If **SELECTED\_INT\_KIND**() returns **-1**, this means the hardware does not support the requested **KIND**.

## What **KIND** Is It (**INTEGER**)? 2/2

- **SELECTED\_INT\_KIND()** is usually used in the specification part like the following:

```
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND(2)  
INTEGER(KIND=SHORT) :: x, y
```

- The above declares an **INTEGER PARAMETER SHORT** with **SELECTED\_INT\_KIND(2)**, which is the **KIND** of 2-digit integers.
- Then, the **KIND=** attribute specifies that **INTEGER** variables **x** and **y** can hold 2-digit integers.
- In a program, one may use **-12\_SHORT** and **9\_SHORT** to write constants of that **KIND**.

## What **KIND** Is It (**REAL**)? 1/2

- Use **SELECTED\_REAL\_KIND**(*k*, *e*) to specify a **KIND** for **REAL** constants/variables, where *k* is the number of significant digits and *e* is the number of digits in the exponent. Both *k* and *e* must be positive integers.
- Note that *e* is optional.
- **SELECTED\_REAL\_KIND**(7, 3) selects a **REAL KIND** of 7 significant digits and 3 digits for the exponent:  $\pm 0.xxxxxxx \times 10^{\pm yyy}$

# What **KIND** Is It (**REAL**)? 2/2

- Here is an example:

```
INTEGER, PARAMETER ::                                &  
    SINGLE=SELECTED_REAL_KIND(7,2),                   &  
    DOUBLE=SELECTED_REAL_KIND(15,3)  
REAL(KIND=SINGLE) :: x  
REAL(KIND=DOUBLE) :: Sum  
  
x      = 123.45E-5_SINGLE  
Sum    = Sum + 12345.67890_DOUBLE
```

## Why **KIND**, etc? 1/2

- Old Fortran used **INTEGER\*2**, **REAL\*8**, **DOUBLE PRECISION**, etc to specify the “precision” of a variable. For example, **REAL\*8** means the use of 8 bytes to store a real value.
- This is not very portable because some computers may not use bytes as their basic storage unit, while some others cannot use 2 bytes for a short integer (*i.e.*, **INTEGER\*2**).
- Moreover, we also want to have more and finer precision control.

## Why **KIND**, etc? 2/2

- Due to the differences among computer hardware architectures, we have to be careful:
  - The requested **KIND** may not be satisfied. For example, **SELECTED\_INT\_KIND(100)** may not be realistic on most computers.
  - Compilers will find the best way good enough (*i.e.*, larger) for the requested **KIND**.
  - If a “larger” **KIND** value is stored to a “smaller” **KIND** variable, unpredictable result may occur.
- Use **KIND** carefully for maximum portability.

**The End**