

Goals for This Lecture:

- Understand the relational operators
- Understand logical operators
- Understand character variables
- Understand character substrings
- Understand the character concatenation operator
- Relational operators with character variables
- Initialization of variables
- Debugging strategies

Important Homework Clarification

- Submit the FORTRAN source code file via email using the mail command
- **DO NOT** send the executable file!
- Reminder:
 - Secure your files by executing the command
> `chmod og-rwx .`
In your home directory to turn off read, write, and execute permissions for everyone but yourself

Setting Your PATH to include the current working directory

- When trying to run your program perhaps you are seeing “*command not found*” errors
- This means the current working directory is not included in your path

- To see the value of the PATH variable

```
>echo $PATH
```

```
/bin:/usr/bin:/usr/local/bin:/opt/intel_fc_bin
```

- To add the current working directory to your PATH:

- Make sure that you are using the bash shell

```
>echo $SHELL
```

```
/usr/local/bin/bash
```

- Add “.” to the beginning of your PATH variable

```
>export PATH=.:$PATH
```

- Check your PATH variable again

```
>echo $PATH
```

```
./bin:/usr/bin:/usr/local/bin:/opt/intel_fc_bin
```

Relational operators

- Relational operations combine two numerical or two character values to yield a logical result or `.true.` or `.false.`

- Example:

`3.0 > 2.1`

- The operators are used in the form:

`a1 op a2`

- Example:

```
logical :: tval
```

```
tval = 31.4 < 32.5
```

Operator	New Style	Old Style
Equal to	<code>==</code>	<code>.eq.</code>
Not equal to	<code>/=</code>	<code>.ne.</code>
Greater than	<code>></code>	<code>.gt.</code>
Less than	<code><</code>	<code>.lt.</code>
Greater than or equal to	<code>>=</code>	<code>.ge.</code>
Less than or equal to	<code><=</code>	<code>.le.</code>

Relational Operator Examples

Operation	Result
<code>4 > 3</code>	<code>.true.</code>
<code>4 <= 3</code>	<code>.false.</code>
<code>4 == 3</code>	<code>.false.</code>
<code>4 < 3</code>	<code>.false.</code>
<code>4 /= 3</code>	<code>.true.</code>
<code>4 .gt. 3</code>	<code>.true.</code>
<code>4 .le. 3</code>	<code>.false.</code>
<code>4 .eq. 3</code>	<code>.false.</code>
<code>4 .lt. 3</code>	<code>.false.</code>
<code>4 .ne. 3</code>	<code>.true.</code>

- Hierarchy of operations:
 - Relational operations are evaluated after numerical operators

Logical operators

- Logical conditions can be evaluated with binary operators

Operator	Function	Definition
<code>L1.and.L2</code>	Logical AND	.true. If both L1 and L2 are true, .false. otherwise
<code>L1.or.L2</code>	Logical OR	.true. If L1 or L2 are true, .false. otherwise
<code>L1.EQV.L2</code>	Logical equivalence	.true. If L1 = L2, .false. otherwise
<code>L1.NEQV.L2</code>	Logical nonequivalence	.true. If L1 is different from L2, .false. otherwise

- Logical values can be negated with the unary operator `.NOT.`

Operator	Function	Definition
<code>.NOT.L1</code>	Logical negation	.true. If L1 is false, .false. If L1 is true

Character variables

- Character variables are declared with declaration statements of the form:

```
character(len=<len>) :: var1, var2, ...
```

```
character(<len>) :: var1, var2, ...
```

```
character :: var1, var2, ...
```

- The number of characters in the variable is specified by the (len=<len>) or <len> attributes.
- If specification is omitted from the declaration statement the length defaults to 1
- Examples:

```
character(len=15) :: first_name, last_name
character(10) :: title
character :: label
```

Character Assignment

- Character variables can be assigned values with assignment operator.

```
character(len=1) :: tag
tag = 'A'
```

- If the character variable is longer than the character expression the value assigned to the variable is padded on the end with blanks

```
character(3) :: name
name = "a"
```

– **name** gets the value “a”

- If the character variable is shorter than the character expression the value assigned to the variable is truncated

```
name = "abcdef"
```

– **name** gets the value “abc”

Character Substrings

- It is possible to refer to a portion of a character variable called a substring
- Example:

```
character(len=10) :: cstring
cstring = "abcdefghij"
```
- `cstring(3:5)` contains the value “**cde**”
- `cstring(9:9)` contains the value “**i**”
- Substring references are of the form:
 - `variable(begin:end)`
 - *begin* is the location of the beginning of the substring within the variable
 - *end* is the location of the end of the substring within the variable
 - If *begin* > *end* the string has zero length

Character string concatenation

- Character strings can be combined with the *concatenation operator* “//”
- The concatenation operator can be used to form character expressions
 - Example of use:

```
character(len=10) :: first_name, last_name
character(len=20) :: full_name
first_name = 'Douglas'
last_name = 'Swesty'
full_name = first_name // last_name
```

Relational operators & character values

- Relational operators can be used to compare character values
- Be careful doing this!
 - Results depend on the collating sequence which may be machine dependent
- If two strings of different lengths are compared and they agree up to the length of the shorter one, then the longer of the two is considered greater
- Examples for ASCII character set:

Operation	Result
<code>'a' > 'b'</code>	<code>.true.</code>
<code>'A' <= 'a'</code>	<code>.false.</code>
<code>'a' == 'A'</code>	<code>.false.</code>
<code>'A' < 'C'</code>	<code>.true.</code>
<code>'4' > '3'</code>	<code>.true.</code>

Initialization of Variables

- Variables must always be initialized before they can be used in expressions, function arguments, or on the right side of an assignment statement.
- *Uninitialized variables* can cause an error
 - You may be unaware of this error!
- Do not assume that an uninitialized variable has a value of zero!
- Variables can be initialized in four ways:
 1. Declaration statement
`type :: var1=value, var2=value,...`
 2. Assignment statements
 3. Read statements
 4. Subroutine calls (we will come to this later in the semester)

More Debugging Tips

- Compile your programs as you write them
 - Start with:
`program`
`end program`
and work up from there
- Break long expressions up into sub-expressions
- Use lots of parenthesis to clarify order of evaluation
- Use lots of write statements
- Echo your read statements with write statements to make sure the correct values were read in
- Make sure that you are using consistent units
- Avoid the *Mars Orbiter* disaster
- Watch out for uninitialized variables
- Watch out for integer or mixed-mode arithmetic

STDIN & STDOUT

- On Un*x systems programs can input and output data to & from many places
- Two are very important:
- Standard input or STDIN
 - Usually the keyboard
- Standard output or STDOUT
 - Usually the terminal screen
 - The first * in `READ(*,*)` tells the compiler to use STDIN for input
 - The first * in `WRITE(*,*)` tells the compiler to use STDOUT for output
- Many Un*x commands write to STDOUT and read from STDIN

Redirecting STDOUT & STDIN

- On Un*x systems the STDOUT can be redirected to a file
- This is accomplished with the “>” character on the shell command line
- Example:
`>ls > file_list.txt`
redirects the output of the ls command to the file named file_list.txt
- You can do the same thing with your FORTRAN programs
- Try it with your hello world program
- STDIN can be redirected on the shell command line with the “<” character
- Example:
`mail phy277@mail.astro.sunysb.edu < hello_world.f90`
Mail command then takes it's input from the file named hello_world.f90
– Easy way to email a program quickly

Reading Assignment

- Read Sections 3.1-3.3