Any Questions?
Topics for this week

- The Clock
- Logical Operators
- Conditional Statements
- DO loops
- Whole Array Operators
- Multi-Dimensional Arrays
- Subroutines and Functions
The Machine Clock

- There is a little electronic heartbeat in the processor called the clock, that switches on and off.
- Each time the clock goes on, the processor gets another chunk of 1's and 0's from memory and sends them down the pipeline.
  - Effectively, the computer can do one operation per cycle.
  - If you are pulling from a slower part of memory, the machine might sit idle for several cycles waiting.

- The speed of your processor is the speed of your clock - 2 GHz = On/Off $2 \times 10^9$ (2 billion) s$^{-1}$
A FLOP is a Floating point Operation

Because Floating point numbers have two parts and extreme precision requirements, multiplication and division with floats can take several cycles to do one operation.

Supercomputers mostly do floating point math, so their performance is measured in FLOPs instead of cycles.

Recently, computers were created that produce PETA-FLOP speed (10^{15} flops/s – 1 quadrillion, or 1,000 trillian operations each second)
Another type of operation that we can do is decision making.

The IF statement evaluates a logical statement and determines whether or not the body statements will be evaluated.

The Logical Type

```fortran
IF (logical expression) THEN
    ....  (body)
ENDIF
```

See Example: IfExample.f90

Each If statement requires at least 2 cycles
Relational and logical operators

- .TRUE., .FALSE.
- 0 is false, all other values are true
- == Equal (.eq.), /= Not equal (.neq.)
- >= Greater than or equal (.ge.)
- <= Less than or equal (.le.)
- < Less than (.lt.), > Greater than (.gt.)
- .AND., .OR. and .NOT.
- .eqv., .neqv.
New Order of Operations

- (), **, */+, +-
- .EQ., .NE., .LT., .LE., .GT., .GE.
- .NOT.
- .AND.
- .OR.
- .EQV. and .NEQV.
Try some of these out...

- $2^{a \cdot b} = 2^{(a \cdot b)}$
- $T \land T \lor F$
- $a > b \equiv a \geq b$
- $\neg T \lor 5 < 3$
- $3 + 4 \neq 12 - 5 \land 2 > 1$
**IF/THEN/ELSE**

- An efficient way to branch control between two sets of operations.

- Very similar to an IF statement, but with a second option, only one body or the other will be executed.

```c
IF (logical expression) THEN
    ....  (body1)
ELSE
    ....  (body2)
ENDIF
```
A less efficient and more complex method to branch between multiple bodies of operations.

```
IF (logical expression) THEN
   ....  (body1)
ELSE IF (logical expression) THEN
   ....  (body2)
ELSE
   ....  (body2)
ENDIF
```

This structure can be dangerously overused.
CASE Statements

Could be more efficient, because the initial expression is only evaluated once.

Less likely to accidentally evaluate code due to poorly formed secondary tests.

Much easier to read, understand and debug.

UNFORTUNATELY, the value of the case selector must be a single integer, character (of any length), or logical value, which limits use.
CASE Statements

\[
\text{SELECT CASE (expression)} \\
\quad \text{CASE (case selector)} \\
\quad \quad \ldots \\
\quad \quad \text{CASE (case selector)} \\
\quad \quad \quad \ldots \quad \text{(body2)} \\
\quad \text{CASE DEFAULT} \\
\quad \quad \ldots \quad \text{(body2)} \\
\text{END SELECT}
\]

See Examples
Computers DO Operations Repeatedly

- The DO loop lets us do the same series of operations over and over again

  DO Counter = Start, End, Increment
  .... (Operations)
  ENDDO

- See Example OscarCount.f90
DO Loops, A Few Rules

- The statements in the body of the loop are repeated until the counter no longer equals the “End” value.

- Note that the loop IS executed at the start and end values - unlike IF statements.

- The “Increment” is optional, if omitted, the compiler assumes it is “+1”.

- You must be able to reach “End” from “Start” (so an Increment of 0 is not allowed), if this is not possible, the loop is not executed.

- Need to be careful, plenty of infinite loops are not caught by the compiler.
Nesting DO Loops

This can be done as many times as necessary, but be careful, because your statements will be evaluated exponentially more times for each loop!

DO Counter = Start, End, Increment
   DO Counter2 = Start, End, Increment
      .... (Operations)
   ENDDO
ENDDO

To keep it understandable, Counter and Counter2 should be different variables

Two nested loops are order $n^2$, three is $n^3$, etc
More on Arrays

- Rank, bounds, extent, size, conformable

Subsections: \texttt{xarr(start:end)}

Fortran Triplets: \texttt{xarr( start:end:stride )}

Whole Array arithmetic - the arrays must be conformable

\[ C = A + B \quad E = C \times D \quad \text{foo} = \text{bar}^{\times 0.5} \quad \text{results} = 0 \]

- adds or multiplies each element of each array to the other
Multi Dimensional Arrays

- `type, dimension(dim1,dim2,...) :: name`

- `REAL,
dimension(lon,lat,height,time) :: temp`

- Array element ordering - indices vary slower as the dimension gets larger `{(1,1)(2,1)(3,1)(1,2)(2,2)(3,2)(1,3)(2,3)(3,3)}`

- Higher dimensional arrays are usually stored contiguously in memory, in ROW MAJOR order `(lon, lat)`
# Multi-Dimensional

For more dimensions:

- $(1,1,1)(2,1,1)(3,1,1)$
- $(1,2,1)(2,2,1)(3,2,1)$
- $(1,3,1)(2,3,1)(3,3,1)$
- $(1,1,2)(2,1,2)(3,1,2)$
- $(1,2,2)(2,2,2)(3,2,2)$

<table>
<thead>
<tr>
<th></th>
<th>Dimension 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Array Transformation

Reshape function is pretty cool

Matrix = RESHAPE( Source, Shape )
A = RESHAPE( B, (/3,2/) )

Another way to index your array elements uses ‘mod’ and integer division

lon = array(MOD(i,num_lons))
lat = array(i/num_lats)

Can store data as (lat, lon) as well, but it will be counter-intuitive and less efficient to transform
WHERE statements

- An easy way to initialize or set sections of arrays

WHERE (array expression)
array assignment block
ELSEWHERE
array assignment block 2
END WHERE

- This is called “masking”
This statement indicates to the compiler that the operations can be performed in parallel (no operations depend on the value of the operation on other elements in the array)

\[
\text{FORALL (triplet)}
\]
\[
\text{variable = expression}
\]
Subroutines

In our constant effort to avoid re-inventing the wheel, we can abstract our code by packaging certain parts to be reusable.

A subroutine is just a sub-section of code that can be reused in the main program.

Subroutines can be included in the same file as the main program or in other files (but we’ll talk about that next week)

In Fortran, arguments are passed by reference, so subroutines can change the value of variables back in the main program.
Subroutines

```fortran
SUBROUTINE name ([arguments])
  Implicit None
  argument declaration with INTENT
  .... (body)
END SUBROUTINE name
```

- The arguments are generally optional, but are the safest way to pass information.
- In your main program, call the subroutine as:
  ```fortran
call name ([arguments])
```
- See example OscarCount2.f90
Subroutines

Intent lets the compiler know what you are doing with the variables

It is not required, but will help your compiler help you

- INTENT(IN) - the parameter has a value when the function is called, and that value is not changed
- INTENT(OUT) - the parameter does not have a given value, but is assigned to when the subroutine or function is run
- INTENT(INOUT) - the parameter is given a value which could be changed by the subroutine.
Functions

Exactly the same as subroutines, except Functions can return a value.

Like all of the built-in functions we’ve been using, a call to a function returns the value which can be stored in a variable (but the types must match).

\[ \text{result} = \text{funct}(\text{arg1}, \text{arg2}, \ldots) \]

Example OscarCount3.f90
Functions

- You can declare a function in the same file as your main program (after it)

```fortran
TYPE FUNCTION name ([arguments])
  Implicit None
  argument declaration with INTENT
  .... (body)
END FUNCTION name
```
What did we cover today?

- The clock
- Logical execution control
- Relational and logical operators
- DO loops
- Whole array manipulators
- Multi-Dimensional Arrays
- Where and ForAll
- Subroutines and Functions