LONI Fortran Workshop @ LSU

Brought to you by:

- HPC@LSU
- LSU ITS
- CCT HPC Enablement
- LONI

Your friendly spirit guides on this quest:

Le Yan, Jim Lupo, Alex Pacheco, Bhupender Thakar, and Kathy Traxler





Survival Tools for HPC

Over the next 4 days, we hope to impart enough knowledge to let you concentrate more on research than programming:

- Fortran Basics
- Fortran Advanced
- Software Development
- Parallel Programming

Hopefully, if you are tossed 100,000 lines of Fortran and told to use the code for research, you won't run screaming for the exits. At least you'll know who to call (*Code Busters*?)



General Logistics

Each day will follow a common pattern:

- 8:00 AM start
- Two morning presentations with a break
- Provided lunch
- Afternoon recap or lecture plus hands-on labs.

Wireless network information is on the board.

Questions are encouraged as we go. Some answers may be put off until the lab sessions if they involve too many technical details.





Transition to Part 1

Fortran – A Survival Guide
Part 1

Dr. Jim Lupo
CCT HPC Enablement
jalupo@cct.lsu.edu



Fortran for programmers

Introduce the basic concepts of the Fortran Language.

Will assume understanding of such concepts as:

- Variables
- Control structures
- Arrays
- Data files



Will cover:

- Syntax and semantics
- I/O control
- Best practices



In the Beginning . . .

*Those who do not read history are doomed to repeat it …**

But Fortran programmers do it naturally.

* *Attributed to Harry Truman*

The Many Flavors of Fortran

- FORTRAN – first released by IBM in Oct, 1956
- FORTRAN II – released by IBM in 1958
- FORTRAN IV – released in 1962, standardized.
- FORTRAN 66 – appeared in 1966 as an ANSI standard.
- FORTRAN 77 – appeared in 1977, structured features
- Fortran 90 – 1992 ANSI standard, free form, modules
- Fortran 95 – a few extensions
- Fortran 2003 – object oriented programming
- Fortran 2008 – a few extensions



Relatively Little Obsolescence

- Most of the changes have been extensions and new features.
- Only some 15 features have been deleted or declared obsolete.
- You may run across "dusty deck" FORTRAN II programs that still compile and run after a few small changes in the code.
- A favorite language of scientific and engineering programmers –"*old FORTRAN never dies*".



There are Standards, and Then There are MY Standards

- "Standard Fortran" requires 3 determinations:
  - Mandated specifications of the standard.
  - Implementation dependent specifications.
  - Additional features added by the compiler vendor.*
- "Language Lawyering" involves:
  - Being aware of the standard.
  - Being acutely aware of implementation artifacts.

*Best Practice Note: Avoid vendor extensions.





Drilling Down

Assume you've been tossed a program that applies a 4-point stencil Jacobi algorithm to the solution of the Laplace Equation for heat diffusion through a slab. There just happens to be such a file in directory:*

```
/work/jalupo/F90_Examples/Laplace_Serial/
```

File name:

```
laplace_solver.f90
```

Feel free to grab a copy and refer to the source as we go along.

*Full tarball: **`/work/jalupo/f90_examples.tbz`**



Program Units

First things to notice:

- Content in single file, which is NOT required.
- Divided into multiple program units.

```
program laplace_main
… stuff …
end program laplace_main

subroutine initialize
… stuff …
end subroutine initialize

subroutine set_bcs
… stuff …
end subroutine set_bcs
```



Unit Scope

Other unit types not shown include:*

- **`function`**
- **`module`**.

Units define the basic scope, or range of program statements, in which definitions and declarations are valid. In general:

- Elements are not visible outside the unit by default.
- Unit names are visible, and we'll see how they are referenced.

*Best Practice Note: one program unit per file.





Source Format

Fortran has two stylistic approaches to formatting the source code:

- Fixed format – columns are significant
- Free format – columns are mostly ignored

They can not be arbitrarily mixed and matched within the same source.

The example uses a fixed format, but we'll look at the details of both format styles.*

*Best Practice Note: use free style within fixed format columns.



Fortran Fixed Source Format

- Fixed source form – throw-back to the days of punched cards, each line follows a strict column-oriented form.
  - **Required** for FORTRAN 77 and earlier
  - **Optional** for Fortran 90 and later
    - Statements may only appear within columns 7 to 72.
    - Blanks are not significant, except in character strings.
    - "**C**" or "*****" in column 1 signifies a comment line, as does a blank line
    - "**;**" may separate multiple statements on a line.
    - A character in column 6 indicates a continuation line – may have up to 19 continuation lines.
    - Digits in columns 1 to 5 represent a *label*.



Fixed Source Example

```
123456789012345678901234567890123456789012345
      program left_right
      real a, b, x(5), y(5)
c     This is a comment
      a = 15.9; b = 42.0
      print 100, x, y
  100 format (f10.1, f10.2, f10.3, f10.4,
     $        f10.5)
      end
```

Fixed column display assumes a fix-space font, like Courier



Side Facts

- Except for character values (i.e. strings), Fortran keywords are case-insensitive.
- Local convention and personal tastes tend to dictate the overall style (various editors can help):
  - All upper/lower-case or mixed case.
  - Stylized comments.
  - Indentation amounts.

Remember those who will come after – make it ***READABLE!***

Did I mention *old Fortran never dies*??



Fortran Free Source Format

- Valid **only** for Fortran90 and later.
- Blanks (white space) outside of character strings are significant
  - multiple blanks are treated as 1.
  - must be used to delineate language keywords. (e.g. `DO, IF`)
- Maximum line length is 132 characters

  (Take a guess at the maximum line length of main-frame chain printers?)
- An "`!`" starts a comment which runs to end-of-line.
  - A blank line is also a comment line.
- An "`&`" at the end-of-line marks a continued line.
- A "`;`" separates statements on a single line.



Free Format Source Example

```
12345678901234567890123456789012345678901234567890123456789012345
program left_right
          real x(5), y(5)
  !  This is a comment
  print 100, x, y  ! Yet another comment
  100 format (f10.1, f10.2, f10.3, f10.4, &
               f10.5)
end program
```





Playing Nice Together*

- Use "**!**" for comments in column 1.
- Use 72 column lines (also a readability feature).
- Start statements in column 7 – use only labels in columns 1-5.
- Use "**&**" in column 73 and again in column 6 to continue a line.

Did I mention *old Fortran never dies?* It is often necessary to update or extend old code, or use pieces in new code. Discipline can make life easier for everyone.

*A Fortran programming best practice.



Bad Mixed Format Example

```
12345678901234567890123456789012345678901234567890123456789012345
program left_right
       real x(5), y(5)
c  This is a comment
  print 100, x, y  ! Yet another comment
  100 format (f10.1, f10.2, f10.3, f10.4, &
              f10.5)
end program
```



Unit Internal Order

Consider these lines taken from the Laplace example:

```
program laplace_main
   . . .
   implicit none
   . . .
   real*8 relerr
   . . .
   parameter ( maxiter = 1000000 )
   . . .
   common / params / nr, nc, niter, iprint, relerr
   . . .
   nr = 0
   . . .
```





Statement Ordering

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE, BLOCK DATA</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER
and
DATA</td><td>derived-type, interface block, type, statement functions, specification</td></tr>
<tr><td>DATA</td><td>executable statements</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">Unit local subprograms or modules</td></tr>
<tr><td colspan="3">END</td></tr>
</table>



Can Be Only One

```
1234567890123456789012345678901234567890123456789012345
      program left_right
        real x(5), y(5)
!       This is a comment
        print 100, x, y
  100   format (f10.1, f10.2, f10.3, f10.4,
     $           f10.5)
       end program left_right
```

Every application must have 1, and only 1, **program** unit, even if the source is in multiple files.



FORTRAN => *Formula Translation*

So, how about them formulas?

- variables – Fortran stores data values in a moderately typed fashion:
  - Implicit or explicit typing.
  - Ability to force explicit typing.
- statements – Fortran is declarative and involves:
  - keywords – Fortran specific language syntax feature
  - operators – methods of manipulating variables.
  - labels – method of referring to other statements.





Fortran Variable Types

Memory storage location and digital representation (i.e. precision) are specified by a variable *type:*

- **`REAL`** – floating point numbers, e.g. **`-1.0e+07`**
- **`INTEGER`** – integer numbers, e.g. **`+12, B"1100", O"14", Z"C"`**
- **`COMPLEX`** – complex numbers, e.g. **`(-37.8, 4.2e-12)`**
- **`CHARACTER`** – character (string) values (**`"a"`**).
- **`LOGICAL`** – true/false values (**`.TRUE./not 0, or .FALSE./0`**).

The number of bytes required for each type is determined by the machine architecture, and may be implementation dependent!





KIND Attribute

Prior to Fortran 90, types could be specified with the number of bytes to be used for storing the value:

- **`real*4`** – uses 4 bytes, roughly $\pm 1 \times 10^{38}$ to $\pm 1 \times 10^{-38}$
- **`real*8`** – uses 8 bytes, roughly $\pm 1 \times 10^{308}$ to $\pm 1 \times 10^{-308}$
- **`complex*16`** – uses 16 bytes, which is two real*8 numbers.

The object was to control the numerical accuracy of a code so that similar results could be obtained on different machines. The use of KIND is preferred:

- **`real (kind=16)`** – real stored in 16 bytes, or what could be called "quad-precision".

If you must deal with KIND, become a close, personal friend of the user manual for the compiler you've selected.



Character Variables

Character variables are used to store text strings. They are a fixed number of characters (bytes) in length, declared with the KIND parameter:

```
character(80) title
title = "A Quick Take on Fortran"
```

This is essentially the same as declaring **`title`** to be an array of 80 characters.



Implicit Automatic Types

Scalar ariables are not required to be defined prior to use. They may assume a type based on the first letter of their name:

- **`I-N`** : variable assumed to be an integer.
- **`A-H, O-Z`** : variable assumed to be real.

Any misspelling creates a new variable name! One can force definition by using:*

```
IMPLICIT NONE
```

*Best Practice Note: always use IMPLICIT NONE





Custom Type (Data Structure)

Organizing several variables into a common entity, or data structure is done in Fortran by declaring a custom type:

```
type student
   integer ID_number
   character(30) name
   integer matriculated
end type student
type (student) j_doe
j_doe%ID_number = 5712342
```

Note that a "%" is used to reference a specific component.





Arrays

Any variable type may be used to create arrays or multiple dimensions. There are two methods for declaring arrays:

```
integer min(10), max(10)
```

or

```
integer, dimension(10) :: min, max
```

Multiple dimensions (max of 7 [F90], 15 [F08]) are given by separating them with a comma:

```
real temperature(10,10,10)
```

And, you can control the range of the indexes used to access array elements:

```
logical faulted(0:9,0:9,0:9)
```



Question

What goes on here?

```
type f1
   integer vin
   character(30) make
   integer year
end type f1
type (f1) stable(5)
stable(3)%make = "Honda"
```

Array Terminology

There are a few bits of terminology that will crop up repeatedly when talking about arrays:

- rank – the number of dimensions.
- shape – the size of an array considering its dimensions in the order specified.
- conformable – quality of two arrays having the same shape.

Prior to Fortran 90, arrays had to be manipulated element by element. Modern Fortrans support array operations on conformable arrays:

```
real x(10), y(10)
x = y
```

Defining Versus Declaring

The ol' variable two-step:

- Declare – specify a variable's attributes.
- Defined – memory is set aside for data storage.

```
real a, b(10)
real, allocatable :: c(:)
common / foo / a

allocate( c(20) )
b(4) = 42
```



Operators

The use of operators is very algebraic in form:

```
ans = 42.5 * 53 / 100.0
```

- assignment is left-hand associative (read: *variable is assigned a value*, not *assign a value to a var*.)
- Operators have an implicit order of precedence, or can be controlled with "( )".*
  - `ans = (42.5 * 53 ) / 100.0`

    not `42.5 * ( 53 /100.0 )`

*Fortran Best Practice: use "( )" freely for clarity.



Operator Types

- Unary (signed, single operand): `+, -`
- Binary (two operands): `+, -, *, /, **`
- Character concatenation: `//`
- Comparative: `.eq., ==, .ne., /=, .gt., >, .ge., >=, .lt., <, .le., <=, .not., .and., .or., .eqv., .neqv.`

Examples:

```
y = +x;  y = x**42;  y = x .gt. z;
title = 'My ' // 'Name ' // 'Is ' // 'Earl'
```



Operation Typing

An operation is performed based on the types of the operands involved. Precedence and grouping apply:

```
real*8 z

z = 10/3
! 10 & 3 are integer, integer result, 3.0d+0 results

z = 10/3.0
! 3.0 is real, 10 promoted, real result, z assigned
! 3.3333330000d+0 (not double precision result)

z = 10.0d+0/3
! Operand is double, 3 promoted to double, z assigned
! full double result (3.333333333333d+0)
```



Using Subroutines

How subroutines are called depends on their definitions:

```
subroutine initialize ( t )
real t
```

This defines a subroutine called initialize which takes one argument of type real. To use it:

```
real x; x = 42.
call initialize( x )
```

Fortran uses “call-by-address”, and not “call-by-value”.

More on this later.



Using Functions

Functions can do all that Subroutines can including returning an invocation value.

```
integer increment( i, j )
  integer i, j
  increment = ( i + j ) ! result assigned to name
  return
end
. . .
integer result, result2, increment
result = increment( 42, 3 ) ! result is 45.
result2 = increment( 2, 3 )**2 ! result2 is 25
```



Where Thar Be Dragons!

```
integer function sample( x, y )
  integer x, y, z
  z = x + y;  y = z
  sample = z
  return
end
. . .
integer a, b, c
a = 41; b = 42; c = sample( a, b ) ! c = 93, b = 93
c = sample( 41, 42 ) ! Segmentation Fault!!
```



Safe Programming

Let the compiler help you avoid problems:

```
integer function sample( x, y, z )
  integer x, y, z
  intent (in) x      !  "out" is also supported
  intent (inout) y
  intent (out) z
  . . .
end function sample
```



Intrinsic Procedures

- While we're on the topic of subroutines and functions, it should be noted that there are many routines defined as part of the Fortran language, and others that may be included by the compiler vendor – have user guide handy for whichever compiler you are using.
- Examples:
  - Trig Functions:* **`sin(), asin(), tan(), atan()`**, etc.
  - Math Functions: **`ceiling(), dble(), random_number()`**
  - Utility: **`reshape(), trim(), btest(), clock()`**

*Trig function angles are in radians.



Global Data

- Common blocks (named and unnamed):

```
real*8 temp, pressure, density
common / state / temp, pressure, density
common x, y, z
```

  Place in a program unit that requires access to the named variables. There can be only one unnamed common block.

- Modules (used for data and code):

```
module global
   real*8, save :: temp, pressure, density
end module
. . .
use global
```

COMMON As Scratch Space

An all-to-common old practice invites trouble:

```
real*8 scratch(1000)
common / work / scratch
. . .
integer*1 foo(8000)
common / work / foo
```

The common **work** is treated as a block of 8000 bytes, and the compiler will store values based on local usage.

Storing integers in one program unit and retrieving reals in another can be very bad for the health of your code.



Memory Placement

Fortran stores data in "column-major" order:

columns (j)

$\leftrightarrow$

$$a(i,j) = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \quad \updownarrow \text{ rows (i)}$$

Index i accesses memory in storage order. Has an impact on a performance measure called "cache line misses".





Logical Control Structures

- **do** loops

```
do i = start, end, increment ! must be integer control variables.
do while ( <logical value )
end do
```

- **if, then, else**

```
if ( <logical value> ) then
else if ( <logical value> ) then
else
end if
```

- **select case**

```
select case ( logical_char_integer_value )
    case( 0 )
    case( 'a' )
    case default
end select
```





Flow Control

- **go to <label>**

```
      go to 100
      . . .
  100 continue
```

- **return <value>**
- **exit <construct names>**

```
  big_loop : do
      . . .
      if ( test ) exit big_loop
      . . .
  end do big_loop
```

- **stop**



Question

How does the Laplace code use flow control?





Stop

Any questions before we take a break?



Transition to Part II



Fortran – A Survival Guide
Part 2

Dr. Jim Lupo

CCT HPC Enablement

jalupo@cct.lsu.edu



Internals

- Call sequences
- Memory organization
- PRINT statement
- Format specifiers
- WRITE/READ statements
- Logical unit control





Call-By-Address

The process by which a subroutine or function is called, arguments passed/returned, and values returned is termed the language *calling sequence*. The process is language specific:

- Fortran – ***call-by-address,*** also termed ***call-by-reference***
- C/C++ – ***call-by-value***
- Assembly – ***call-by-descriptor***

This is directly related to the concept of a pointer. That is, the value of some variable holds the memory address where the value is stored, not the value itself. (*Fortran does pointers too!*)



Simplified Example

```
real ans    ! memory location 0x12345 reserved.
ans = 42   ! Value of 42 written to 0x12345
call report ( ans ) !  0x12345 placed on stack

subroutine report ( x )
  real x, y  ! Address of x is assigned 0x12345
  y = x ! Value of 42 retrieved from 0x12345
  x = 2.0 * y ! The value of 42 is stored in 0x12345
  return
end report

! Back in caller, the value of ans is now 84.
```



Implications

```
real arry(4,4) ! memory location 0x12345 reserved.
call report ( arry ) !  0x12345 placed on stack

subroutine report ( x )
  real x(1), y  ! Address of x(1) is assigned 0x12345
  y = x(16) ! Valid because 16 reals in arry.
  x(17) = 2.0 * x(1) ! May work, but very BAAAAD!
  return
end report
```

This kind of programming is all too common, perfectly legal but all very dangerous. Common cause of *segmentation faults*.





So What Can Go Wrong?

```
integer function sample( x, y )
  integer x, y, z
  z = x + y;  y = z
  sample = z
  return
end
. . .
integer a, b, c, sample
a = 41; b = 42; c = sample( a, b ) ! c = 93, b = 93
c = sample( 41, 42 ) ! Segmentation Fault!!
```



One Proper Way – Assumed Size

```
real arry(4,4) ! memory starting at 0x12345 reserved.
call report ( arry, 4, 4 ) !  0x12345 placed on stack

subroutine report ( x, mx, my )
  integer mx, my
  real y
  real x(mx,my)! Address of x(1,1) is assigned 0x12345
  y = x(4,4) ! Valid because 16 reals in arry.
  x(5,1) = 2.0 * x(1,1) ! Run-time error generated
  return
end report
```

You'd have to enable run-time array bounds checking. It is expensive, but can help locate such issues. Or, do in-line error checking.



Recall Memory Placement

Fortran stores data in "column-major" order:

columns (j) ↔

$$a(i,j) = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

↕ rows (i)

Index i accesses memory in storage order. Has an impact on a performance measure called "cache line misses".



Memory Locality

All modern processor have memory on their silicon called cache. This is high speed, and accessed faster then may memory, but of limited size (1MB per core, maybe). If the processor goes off chip to memory, things slow way down. Assume 256KB of L1 cache, and the following code:

```
real*8 arry(1000,1000) ! requires 8MB of storage.
```

and one of these stanzas:

```
do j = 1, 1000
  do i = 1, 1000
    x = arry(i,j) + 1
  end do
end do
```

```
do i = 1, 1000
  do j = 1, 1000
    x = arry(i,j) + 1
  end do
end do
```





Global, Local, Static – Oh My!

Local variables are allocated on the call stack:

```
subroutine demo
   integer x
   x = x + 27
   ...
end subroutine demo
```

Thus **x** is created when **demo** is called and disappears when **demo** is exited. Value of **x** IS NOT retained between calls! It will be set to 0 each time **demo** is entered.

SAVE Statement/Attribute

A variable can be assigned static memory so its value persists between calls:

```
subroutine demo
   integer, save :: x
   x = x + 27
   ...
end subroutine demo
```

The **save.f90** demo program illustrates this effect.



Input / Output Terminology

- Formatted – data is human readable
- Unformatted – machine readable binary data
- Sequential – data records of random length, one after another
- Fixed – data records of constant length
- Direct – data can be accessed by position (random access)
- Unit – logical unit number connected to a file or device
- Open – associate a unit number with a file or device
- Close – disconnect unit number from file or device
- * – shorthand for "use default format based on variable type"
- * – shorthand LUN for Unix-style stdin, stdout devices.
- Label – a statement label, for I/O it is a format statement.

There are advantages and disadvantages to various combinations.



PRINT Statement

```
print <label | *>, val1, val2, val3 . . .
```

Simplest output command. Sends formatted results to a logical unit number, or the stdout device, using default formats based on the variable type, or as controlled by a format statement.

These days, most vendors use unit 6 as the stdout lun on Unix-like operating systems. Some may still use unit 101.

PRINT Example

```
program main
  print *, 'Hello world!', -42, 22.0/7.0
end program main
```

This outputs a character, integer, and a real value, producing:

```
Hello world!          -42   3.142857
```

Since a real has 7 digits of precision, it shows only 7 significant digits. Spacing varies depending on the combination of values.



A Format Statement

Format statements give the programmer precise control over the appearance of the data being displayed. This is because each variable in the statement list is required to have its own *format specifier*.

```
program main
  print 100, 'Hello world!', -42, 22.0/7.0
  100 format( 1x, a12, 1x, i3, e20.16 )
end program main
```

```
 Hello world! -42  0.3142857074737549E+01
```



Format Specifiers

The general form of a specifier is given by:

```
[rep]<desc><w>[.m]
```

where

```
rep ... optional repetition count
desc .. type specific descriptor flag
w ..... maximum width of output field
m ..... optional minimum digits displayed
```

Commonly Used Descriptors

```
I<w>[.m]  .. Integer
Z<w>[.m]  .. Hexadecimal Integer
A[w]  ..... character
L<w>  ..... logical
F<w>.<d>  .. real, fixed precision
E<w>.<d>  .. real, engineering notation
G<w>.<d>  .. complex engineering notation
/  ........ start a new line (record)
x<w>  ..... blank space
```

There are others possible.



Example Descriptors

```
3.14159 ..... f7.5
-3.14159 ..... f8.5
10 10 10 ..... 3i3, or, 3(i2,1x)
1010101 ..... i7, or, 7i1

0.137e+0 ..... e8.3e2
1.0 2 3.0 4 .. 2(f2.1,i2)
```

Edit descriptors are also used when reading formatted data.



Logical Unit Numbers

A logical unit numbers (LUN) identifies a specific I/O channel. Could be a tape drive, but more commonly a data file.

The **open** statement associates a file with a LUN:

```
open( [unit=]<lun>, [file=]<string_value>,
      [form=formatted|unformatted],
      [status=old|new|unknown|replace|scratch],
      [action=read|write|readwrite],
      [blank=null|zero],[err=label],
      [delim=apostrophe|quote|none],
      [iostat=integer_var], [pad=yes|no],
      [pos=asis|rewind|append],[recl=integer],
      [access=direct|sequential] )
```



Typical Open Usage

```
open( 10, status='scratch' )


open( 10, file='output', status='new' )


open( 20, file='restart', status='new', &
      form='unformatted', err=1010 )
. . .
1010 continue
! Ooops, failed to open restart file
. . .
```



Become a Fortran Lawyer

This presentation barely skims the details regarding I/O specifications. The standard has much more to say, and it may be important to you depending on specific usage.

Consider **`IOSTAT`** in a **`READ/WRITE`** statement:

> *If an IOSTAT= specifier is present, and end-of-file condition occurs, and no error condition occurs, the IOSTAT variable specified becomes defined with a* ***processor-dependent*** *negative value.*

Need the Fortran standard and the vendor compiler user guide!





And the Opposite of Open Is?

Anyone, anyone?

```
close( [unit=]<lun>, [iostat=ierr], [err=label] &
       [status=keep|delete] )
```

The default value of status is **keep**, unless the unit was opened with status **scratch**, it which case the default is **delete**.

A statement we won't cover is **inquire()**, which allows a program to determine the characteristics of a file, or a unit.



The **WRITE** Statement

The **write** statement is a much more powerful form of the **print** statement, since it allows specification of a **unit**.

```
print *, 'Hello world!'
print 100, 'Hello world'

write( *, * ) 'Hello world!'
write( 6, * ) 'Hello world!'
write( 6, '(a)' ) 'Hello world!'
write( 6, 100 )

100 format( 'Hello world!' )
```



WRITE Statement Details

```
write( [unit=]<lun>, [format=]<format>, &
       [[nml=]namelist], [advance=no|yes], &
       [end=label], [eor=label], [err=label], &
       [iostat=integer_var], [rec=recd_num] )
```

The available write specifiers may apply only to specific types of files. For instance, **`rec`** applies to a fixed record sized random access file.



Fortran **READ** Statement

The **print** and **write** statements are handy for getting results out of a program. Getting data in generally requires **read** – almost identical to write, but data moves in opposite direction.

```
read( [unit=]<lun>, [format=]<format>, &
      [[nml=]namelist], [advance=no|yes], &
      [end=label], [eor=label], [err=label], &
      [iostat=integer_var], [rec=recd_num] )
```

If you know how the data was written, it can help correctly read it back in. It can get tedious!





Some Data To Read

Here's a line of data from a file on unit 20, with column markers (in red) to show spacing:

```
12345678901234567890123456789
brakepad      27.95  100429 0
```

Looks like we'll need a character, real, integer, and logical variable to store the data:

```
character(12) partname
real price
integer partnum
logical instock
```

Format Specifiers Needed

```
First 12 spaces are for the part name:          a12
6 spaces and 2 significant digits for price:  f6.2
Assume 8 spaces for the part number:            i8
Skip a blank space:                             1x
Last item is a logical:                         l1

Should be able to use:

   read( 20, '(a12, f6.2, i8, 1x, l1)' ) partname, &
         price, partnum, instock
```



Cruel Example

Here's a line of data from a file on unit 20, with column markers (in red) to show spacing:

```
12345678901234567890123456789
bacadafagaaa0123.45.34.67.8x
```

How many different formats could be used to read this line?

`'(a12,3i1,4f3.1,a)'` ? `'(12a1,2i1,4f3.0,i1,a1)'` ?

Need to know what you're supposed to be reading.





Unformatted I/O

- Formatted I/O intended for human consumption.
- Inefficient w.r.t speed, convenience, and storage space
- Loss of precision on reuse.

Consider:

```
Defined:  0.31415926535897932384626433827950288d+01
Real*4:   0.31415927410125732421875000000000000000e+01
Real*8:   0.31415926535897931159979634685441851620000d+01
```

Space required: real*8 – 50+ characters, hexadecimal – 16 characters, binary – 8 characters (bytes).

Note: binary has **no translation required**.



Example

```
real*8 pi8
real*4 pi4
pi8 = 3.14159265358979323846264338327950288d+0
pi4 = pi8 ! implicit conversion from real*8 to real*4
open( unit=25, form='formatted', type='new' )
open( unit=26, form='unformatted', type='new' )
write( 25, '(e50.40, d50.40)' ) pi4, pi8
write( 26 ) pi4, pi8
close( 25 )
close( 26 )
```

This was used to generate the results presented on previous page.

Unit Control

There are 3 statements which provide device control to the user:

```
backspace(), rewind(), endfile()
```

Their proper usage depends on the I/O specifiers associated with a a unit, and discussion would take us off into the weeds.

Normally found when dealing with record oriented unformatted I/O. Originally intended for tape drive control.





NAMELIST I/O

A **namelist** provides a convenient way to input and output related variables as key-value pairs – program control input, for instance.

```
real *8 x, y, z
integer*4 i, j, k
namelist / input / x, y, z, i, j, k
. . .
write( 7, nml=input )
. . .
read( 7, nml=input)
```

Data for input could appear in a text file as:

```
&input  x=1.0, k=1234, z=-2.0d-27 /
```



In Summary

- Fortran offers features similar to many other languages.
- Some features are left up to the compiler vendor.
- Specific compiler and programming features coming soon.

Remember those who will follow you!

Olde Fortran never dies!

Any questions before we break for lunch?



Transition to Lab

Starting Comments

Do you have any questions about the morning materials before we continue?

We'll cover how to use the compiler and run through some programming examples and exercises.





Working with Editors

vim – it's the bomb

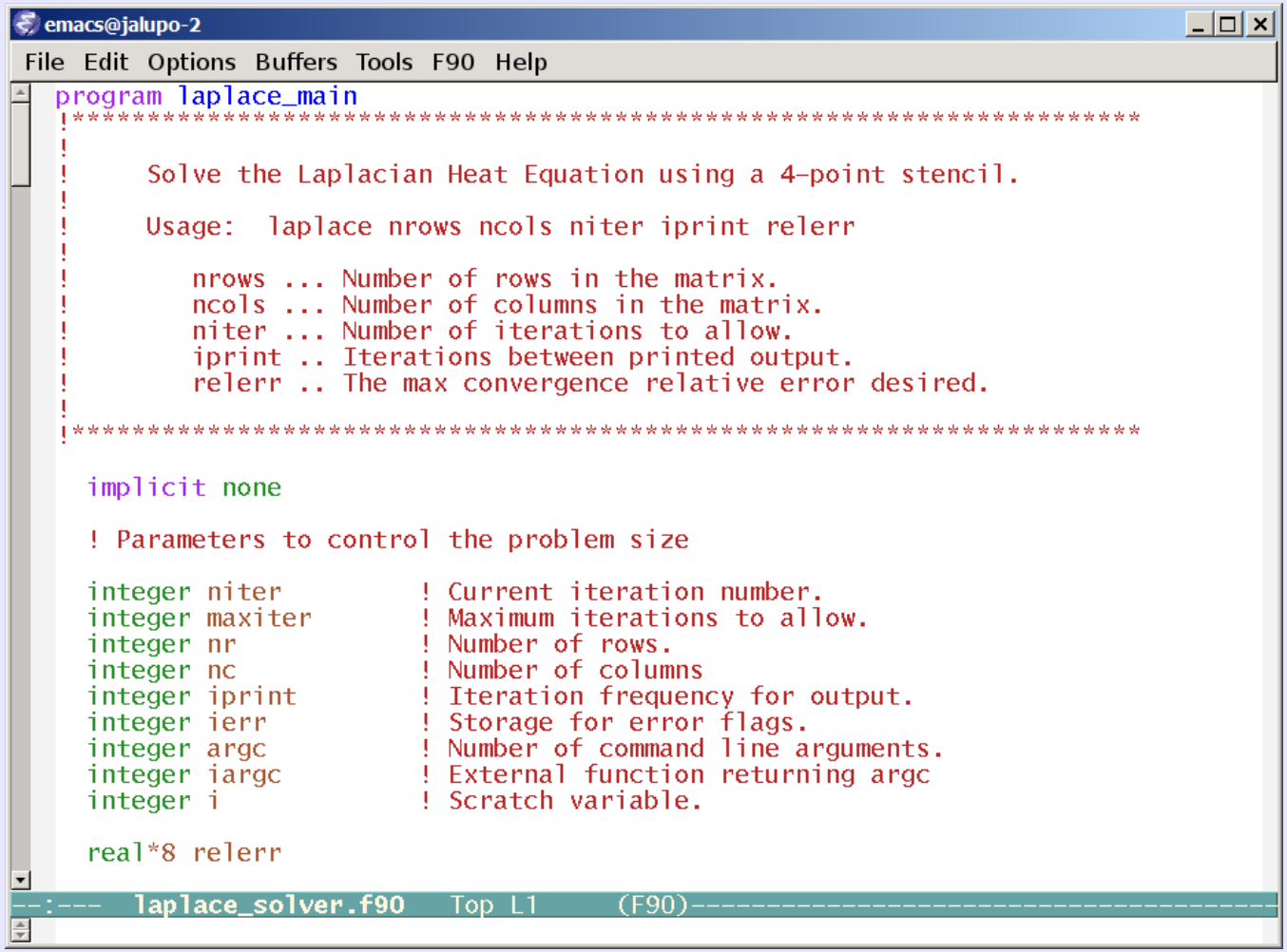
emacs – only real programmers use emacs

It's all about your editor religious beliefs!

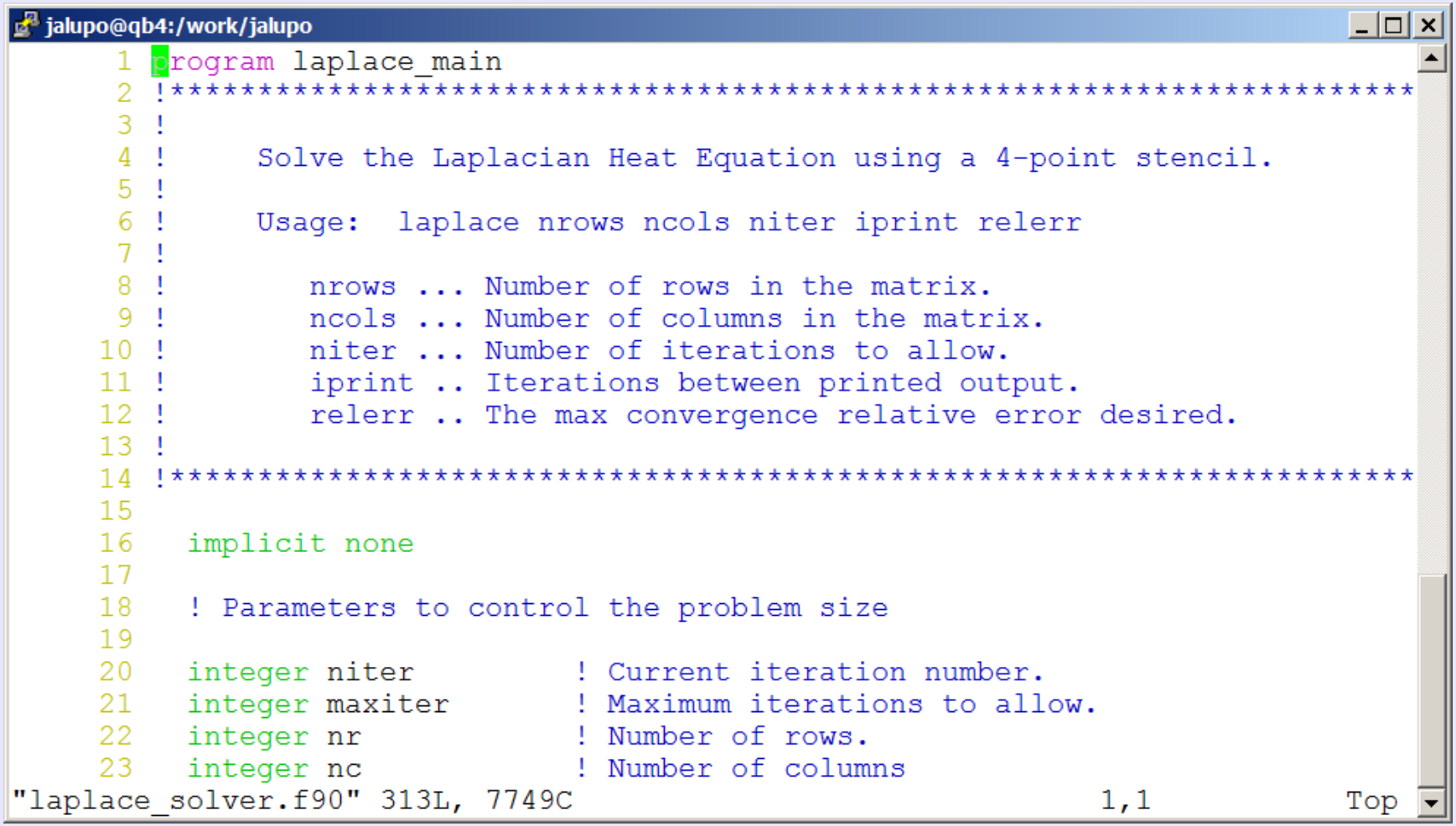
eclipse – wave of the future

notepad – it's for masochists

Context Clues - EMACS



Context Clues - VIM





Set Your Environment

You're shell user environment is controlled by **`softenv`**.
The exercises will assume the default Intel compilers, so your **`.softenv`** file on LONI systems should contain only:

```
@teragrid-basic
@globus-4.0
@teragrid-dev
```

Try this command:

```
$ ifort -V
Intel(R) Fortran Intel(R) 64 Compiler ....
```

Raise your hand if you can't get **`ifort`** to work.





Roll Your Own

Create file **test.f90** with this content:

```
program main
  print *, 'Hello world!'
end program main
```

Compile and execute:

```
$ ifort test.f90
$ ./a.out
 Hello world!
```

My mission is complete – you are a Fortran programmer. The rest is **SMOP** – *simple matter of programming.* Oh, and maybe just a handful of other details . . .



Compiler Command Line

The basic compiler usage is deceptively simple, but can be made arbitrarily complex.

```
$ ifort [options] file1 [file2] . . . [fileN]
$ ifort test.f90
$ ifort -o a.out test.f90
$ ifort -o test -O3 test.f90
$ ifort -c test.f90       # produces test.o
$ ifort -o test test.o
$ ifort -c main.f90 sub1.f90 sub2.f90 sub3.f90
$ ifort -o test main.o sub1.o sub2.o sub3.o
```

The last two show a compile-link two-step process. Future episodes.





Laplace Solver Example

```
$ ifort laplace_solver.f90
$ ./a.out
 Usage: laplace nrows ncols niter iprint relerr
$ ./a.out 100 100 100 50 0.1
Iteration:           50
Max error:        0.669
Iter time:        0.004
Total tim:        0.004
Iteration:          100
Max error:        0.323
Iter time:        0.000
Total tim:        0.004
Iterations:          100
Max error:         0.323
Total time:        0.004
```



Example

Turn this precision example into a real program.

```
real*8 pi8
real*4 pi4
pi8 = 3.14159265358979323846264338327950288d+0
pi4 = pi8 ! implicit conversion from real*8 to real*4
open( unit=25, form='formatted', type='new' )
open( unit=26, form='unformatted', type='new' )
write( 25, '(e50.40, d50.40)' ) pi4, pi8
write( 26 ) pi4, pi8
close( 25 )
close( 26 )
```

What are the default file names?
How would you change the file names?
What is the resulting size difference between the files?





Data File Exercise

Write a program to read and output to the terminal screen, the contents of:

```
/work/jalupo/F90_Examples/Data/inventory.dat
```

Assume: part name, part number, price, quantity-in-stock

Now what if the data looked like:

```
brake pads          101044937      $47.92     126
brake shoes         101044939      $64.29      83
brake cylindar      101044920     $129.97      42
16" wiper refill    102099736      $14.95      10
17" wiper refill    102099737      $14.95      23
```



Impact of SAVE

To see the impact of SAVE, grab of copy of `save.f90`, compile, and run it.

Make note of the change in values.



save.f90

```
module subs

contains

subroutine asub1 (i, control)
   implicit none
   integer, intent (in) :: i
   integer, save :: k
   logical, intent (in) :: control
   if ( control ) k = 0
   k = k + i
   write (*, *) 'i, k=', i, k
end subroutine asub1

subroutine asub2 (i, control)
   implicit none
   integer, intent (in) :: i
   integer k
   logical, intent (in) :: control
   if ( control ) k = 0
   k = k + i
   write (*, *) 'i, k=', i, k
end subroutine asub2

end module subs
```

```
program test_saves

   use subs

   implicit none
   integer i

   print *, "Using ASUB1: value of k " &
            // "retained between calls."
   call asub1( 3, .TRUE. )
   call asub1( 4, .FALSE. )
   call asub1( 5, .FALSE. )

   print *, "Using ASUB2: value of k reset " &
            // "between calls."
   call asub2( 3, .TRUE. )
   call asub2( 4, .FALSE. )
   call asub2( 5, .FALSE. )

end program test_saves
```



Formatted Vs. Binary

- Create a simple program which does the following:
  - Creates some local variables set as show:

```
real*4 xo; real*8 yo
real*4 xi; real*8 yi
xo = 8.125; yo = 1.0203040506070 8d+0
```

  - Write them to a text file; close the file, they read them back in to the variables **`xi`** and **`yi`**.
  - Compute relative errors: **`(xi-xo)/xo`** and **`(yi-yo)/yo`**
  - Repeat, only this time use a binary file.
- Can you draw any conclusions from the results?



NAMELIST I/O

Create a small program using a namelist with at least 5 variables.

1. What does the namelist output look like?

2. Try adjusting the values and read it back in.

3. Note that all namelist members need not be present.



Questions of Personal Interest?



Adiós, until tomorrow