

Goals for This Lecture:

- Understand how to set aliases
- Understand globbing and wildcards in the shell
 - Matching multiple filenames with patterns
- Understand shell variables
- Setting up your bash shell
- Invoking the Compiler
- Compilers: what are they and what do they do?
 - Invoking the compiler
- Compile our “hello world” program
- Discuss the basics of digital data representation

Filename expansion & Wildcards

- Suppose we have a collection of files named `file1.dat`, `file2.dat`, `file3.dat`, `file4.dat` It would be convenient to have a way to refer to the entire set of these files in a shorthand manner.
- Fortunately, the shell provides a mechanism to accomplish this: filename expansion with wildcards, a.k.a. *globbing*

Rather than typing:

```
>ls file1.dat file2.dat file3.dat file4.dat  
file1.dat file2.dat file3.dat file4.dat
```

we can type:

```
>ls file*.dat  
file1.dat file2.dat file3.dat file4.dat
```

- The asterix "*" is called a wildcard.
- The shell expands the asterisk to match any filenames that begin with "file" and end with ".dat"
- The expressions `*.dat` or `file*` would have also matched these same files
- Wildcards are handy but ***can also be dangerous!***

For example, suppose you meant to type:

```
>rm *.dat
```

But instead typed:

```
>rm * .dat
```

Result: Disaster!

Globbering & Wildcards

- Lesson: be very, very careful utilizing globbing (wildcard file matching)!
- Always look twice to make sure that you don't have typos in your wildcard expression
- Make sure that you understand what files are being matched by the expression
- If in doubt test the pattern first using an innocuous command like ls
- Be triply careful if you are doing anything that will alter or delete files based on wildcards

More Globbing...

- The asterix “*” expansion wildcard matches any string of characters of any length in a filename
- There are other ways of matching characters in a filename via shell expansion
- Lists of characters can be used:

```
> ls file[1,3].dat  
file1.dat file3.dat
```
- One can also specify ranges

```
>ls file[1-3].dat  
file1.dat file2.dat file3.dat
```

Still More Globbing...

- Ranges can be numerical or alphabetical:
0-9,a-z,A-Z
- For example, if we have the following files in our current working directory: `file.a,file.b,file.c,file.d,file.e,...,file.z`
> `ls file.[b-e]`
`file.b file.c file.d file.e`
- Expansion wildcards can be combined

Suppose we have the following files in our working directory:

`f_new_1.dat, f_new_2.dat, f_new_3.dat, f_new_4.dat,`
`f_old_1.dat, f_old_2.dat, f_old_3.dat, f_old_4.dat`

Then the pattern `f_*_[2-3].dat` matches the files:

`f_new_2.dat f_new_3.dat f_old_2.dat f_old_3.dat`

Shell variables

- The shell has the capability of holding strings of characters in *shell variables*
- This capability is often used to provide information to the shell or an application.
- Variables can easily be created and set. For example:

```
> x="file1.dat"
```

Creates a variable named X that holds the value "file1.dat"

The value of the variable X can be referenced by placing a "\$" in front of the variable name, e.g. \$X

```
>echo $X  
file1.dat
```

Exporting shell variables into the Environment

- Shell variables defined by commands of the form of “variable=value” are local variables
 - They are known only to the shell
- The shell variables can be made accessible to applications if they are exported into the environment using the export command

```
>export X
```

Makes X accessible to applications.

- The environment contains an entire set of variable that hold values that control the shell and applications
- Try typing the `env` command to see a list of the current shell variables
- The setting and export of a shell variable can be combined into one line of the form:

```
>export variable=value
```

Automatically setting shell variables

- Shell variables can be automatically set and exported when you log in
- This is possible by setting them in the shell configuration file named `.bashrc` (Yes there is a “.” at the beginning of the name!)
- A very important shell variable that is often set in the `.bashrc` file is the `PATH` variable
- This `PATH` variable controls where the shell looks for executable commands.
- Examine the value of the path:
- `echo $PATH`
- Modify your `.bashrc` file to add the following two lines to the file
- `export PATH=$PATH:/opt/intel_fc_80/bin`
- `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel_fc_80/lib`
- These two variables will allow the Intel FORTRAN compiler to work
- To check that your shell can find the compiler execute the following command:
 `>which ifort`
 `/opt/intel_fc_80/bin/ifort`

Exporting shell variables into the Environment

- Shell variables defined by commands of the form of “variable=value” are local variables
 - They are known only to the shell
- The shell variables can be made accessible to applications if they are exported into the environment using the export command
 - >**export X**
 - Makes X accessible to applications.
- The environment contains an entire set of variable that hold values that control the shell and applications
- Try typing the **env** command to see a list of the current shell variables

What's the big deal about FORTRAN?

- FORTRAN today is not your Father's/Mother's FORTRAN!
 - Modern FORTRAN 95 does not look like FORTRAN IV
- What's good about FORTRAN?
 - FORTRAN is very natural language for expressing numerical computations
 - FORTRAN handles arrays really well (extremely important for numerical computing)
 - Think of arrays as enumerated lists of data until we learn about them in a few weeks.
 - Natural syntax for handling subsets or parts of a an array of data
 - Produces extremely fast code.
 - The language has a very restricted set of operations that allows the compiler to find, in many cases, an optimal order in which to execute those operations
 - Languages such as C, C++ allow use of pointers (references to data by it's address in memory) which make finding an optimal order of execution difficult in many cases.
 - C++ code can sometimes be made to run as fast as FORTRAN but it can take some work on the part of the programmer
 - Usually the language recommended by most vendors of supercomputing systems
 - Highly standardized language
 - Makes porting code to a new computer easy
- What's bad about FORTRAN?
 - Not flexible for input/output of data compared to C++
 - Fixed in FORTRAN 2003
 - Not good for system level programming
 - Can be done in some cases but it is not easy
 - Don't try to write an editor in FORTRAN!
 - Expressing modern programming concepts can be difficult in earlier versions of FORTRAN
 - FORTRAN 90 lessens this problem
 - FORTRAN 2003 fully resolves this problem

A “Hello World” FORTRAN program

```
program hello_world  
write(*,*) “Hello World!”  
stop  
end program hello_world
```



Four lines of
FORTRAN code

- Each line of code is called a statement
- Order of statements in program matters

Top-down execution of program

```
program hi_bye  
write(*,*) "Hello World!"  
write(*,*) "Goodbye Cruel World!"  
stop  
end program hi_bye
```



Five lines of
FORTRAN code

- Some statements are executable statements “write” statement & “stop” statement
 - Executed by the computer
- Other statements are non-executable
 - “program” statement & “end” statement
 - Provide information to the compiler
- Executable statements execute in order from the top of program to the bottom of the program

Compiling & executing the “Hello World” program

- We can now compile our “Hello World” program:

```
>ifort -o hello_world hellow_world.f90
```

- There is now an executable file named hello_world in the current working directory

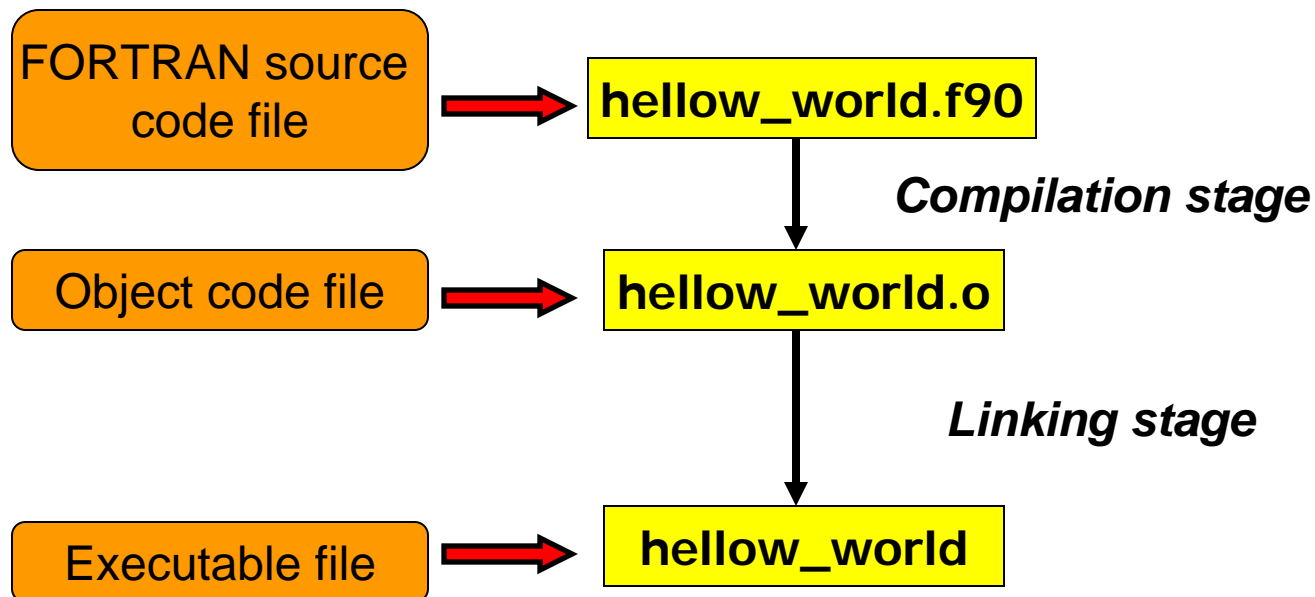
- Executing the program is simple:

```
>hello_world
```

```
Hello World!
```

What is the compiler and what does it do?

- The compiler translates the FORTRAN statements in our program into machine executable instructions
- It does this in two stages: compilation & linking

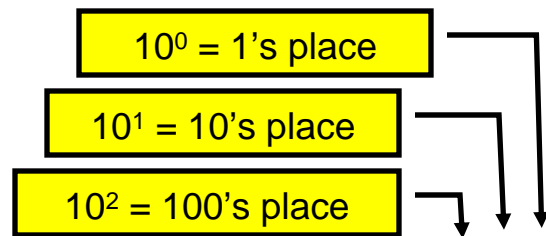


Representing data digitally

- Memory is composed of millions/billions of individual switches which are all in a state of being either ON or OFF
- A switch can represent only the numbers 0 or 1
- Each switch is called a bit
- To store larger numbers we group switches together and store numbers in binary form where each digit is either 1 or 0
- Usually the smallest grouping of numbers is 8 bits which is known as a byte
- Usually bytes are grouped together into 16 bit (2 bytes), 32 bit (4 bytes), 64 bit (8 bytes), or 128 bit (16 byte) words
 - Some computers don't use byte groupings and directly form groups 64 bits together into a word
 - Occasionally one finds odd sorts of machines that use groupings like 36 bit words

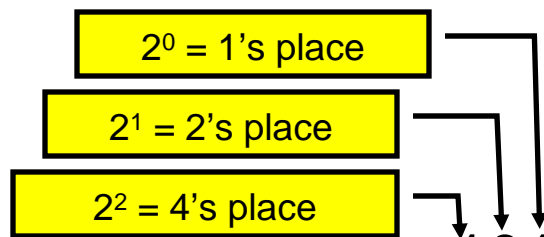
Binary Data Representation

- Base ten (decimal) representation: each digit multiplies a power of 10:



$$137 = 1 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

Base two (binary) representation: each digit multiplies a power of 2:



$$101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

(= 5 in base 10 representation)

Byte Integers

- One byte of data can be used to positive store integers up to:

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 $= 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 127$ (in base 10)



Sign bit; 0 if positive integer
1 if negative integer

- Negative integers are stored via two's complement representation
- Complement the absolute value of the number (change zero's to one's and one's to zero)
- Add one to the complemented number to find the two's complement
- Therefore the value of -127 is stored in two's complement form as:

1	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

- While the two's complement form may seem a bit weird it offers a big advantage in the design of ALU circuits:
 - To subtract one number from another one can form the two's complement of the first number and add it to the second number.
- In general a N-bit integer can store numbers in the range of $-2^{N-1} - 1$ to 2^{N-1}
- In the case of a two byte integer this range is -32768 to 32767
- In most cases, FORTRAN integers are 4-bytes (unless otherwise specified) which gives a integer range of $-2^{31} - 1$ to 2^{31}

Assignments

- Read Sections 1.1-1.6, 2.1-2.3 of Chapman (FORTRAN 90/95)
- Work through University of Utah Un*x tutorial
 - Link on course web page
- Work through University of Surrey Tutorial 1
 - Link on course web page