

PROGRAMMING IN MODERN FORTRAN

Fortran 2003/2008

COMPILERS VERSUS INTERPRETERS

- A **compiler** produces a stand-alone binary for a given *platform* (cpu+operating system). The output of a compiler is an *object file*, represented with a `.o` suffix on Unix.
- A **linker** takes the `.o` files and any external *libraries* and links them into the executable. Normally the linker is invoked through the compiler.
- An **interpreter** interprets line by line. The binary that is run is the interpreter itself. Programs for interpreters are often called *scripts*. Scripts are frequently cross platform, but the interpreter itself must be appropriate to the platform.

COMPILED LANGUAGES

- Compiled languages are:
 - Generally stricter about typing (static typing) and memory allocation.
 - Generally produce faster and more efficient runs.
- Interpreted languages are:
 - Generally looser about typing (dynamic typing).
 - Generally have dynamically sized data structures built in.
 - Often run very slowly.

WHY FORTRAN

- Fortran is still very widely used in many fields of science and engineering, particularly Earth sciences, chemistry, and mechanical engineering.
- Features of modern Fortran make it very well suited to numerically-intensive programming.
- Many scientific programmers have to know both C/C++ and Fortran

WHY DOES FORTRAN HAVE A REPUTATION?

- As the oldest higher-level language, many old code using obsolete constructs still exists and is in use.
- Computers for which the first languages were written were extremely limited in memory, speed, and disk space.
 - Typical memory in the 1960s was measured in kilobytes
 - A supercomputer circa 1994 had 8GB of RAM

ALGOL 60

- Of similar age and similar design
- Sample code (from Wikipedia)

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);  
  value n, m; array a; integer n, m, i, k; real y;  
  comment The absolute greatest element of the matrix a, of  
  size n by m, is transferred to y, and the subscripts of this  
  element to i and k;  
  begin integer p, q;  
    y := 0; i := k := 1;  
    for p := 1 step 1 until n do  
      for q := 1 step 1 until m do  
        if abs(a[p, q]) > y then begin  
          y := abs(a[p, q]); i := p; k := q  
        end  
  end Absmax
```

NEWER FORTRAN

- The language has changed dramatically since 1957 but the name has never changed.
- Algol's descendants include Pascal and C (but note how different C is)

STRENGTHS AND WEAKNESSES

C++ (not C)

- Limited mathematical built-ins
- True multidimensional arrays not possible without add-on libraries (Blitz++, Boost)
- Pretty good string handling (compared to C)
- Straightforward implementation of classes (but no modules)

Fortran

- (2003/8) Many math function built-ins
- Multidimensional arrays a first-class data structure, array operations supported
- Does not support true strings yet, just character arrays
- Classes somewhat clunky. Modules fill much of this role.

QUOTE FROM APPROXIMATELY 1985

- I do not know what the scientific programming language of the year 2000 will look like but it will be called Fortran.
 - Apocryphal, sometimes attributed to John Backus (inventor of Fortran) or Seymour Cray (inventor of the supercomputer).

SETTING UP YOUR ENVIRONMENT

INTEGRATED DEVELOPMENT ENVIRONMENTS

- An Integrated Development Environment (IDE) combines an editor and a way to compile and run programs in the environment.
- A well-known IDE for Microsoft Windows is VisualStudio. Available through Microsoft Store, not free for individuals.
- Mac OSX uses Xcode as its native IDE. Xcode includes some compilers, particularly for Swift, but it can manage several other languages. Available at App Store, free.
- A full-featured cross-platform IDE is Eclipse (www.eclipse.org). Free.
- A lighter-weight IDE for Windows and Linux is Code::Blocks (www.codeblocks.org). Free.
- We will use a very lightweight IDE called Geany since it is free, easy to install and use, and works on all three platforms.

LINUX

- For users of the University of Virginia's cluster, first load a compiler module.

```
module load gcc
```

brings a newer gcc, g++, and gfortran into the current environment

```
module load geany
```

```
geany &
```

- Geany is also available for all popular Linux distributions and can be installed through the distribution's package manager.

WINDOWS AND MAC

- Geany can be installed on Windows and Mac through the usual software installation methods.
- Geany does not install a compiler suite. This must be performed independently.
- Macs with Xcode include gcc and g++ but not gfortran.
- Windows does not include a default compiler suite, but VisualStudio includes Microsoft C and C++.
- Geany can be downloaded for Mac or Windows starting from its home page
 - www.geany.org

INSTALLING COMPILERS ON MACS

- Install Xcode from the App Store.
- If you are going to use Fortran, download a binary for your version of OSX from
- <https://gcc.gnu.org/wiki/GFortranBinaries>

INSTALLING COMPILERS ON WINDOWS

- MinGW provides a free distribution of gcc/g++/gfortran
- Executables produced by the standard MinGW package will be 32 bits
- Also install MSYS for a minimalist Unix system.
 - Download from www.mingw.org
 - Run installer
 - Choose packages to install, then click Apply.
 - After the installation, follow instructions for "After Installing" at http://www.mingw.org/wiki/Getting_Started
 - Be sure to modify your path environment variable

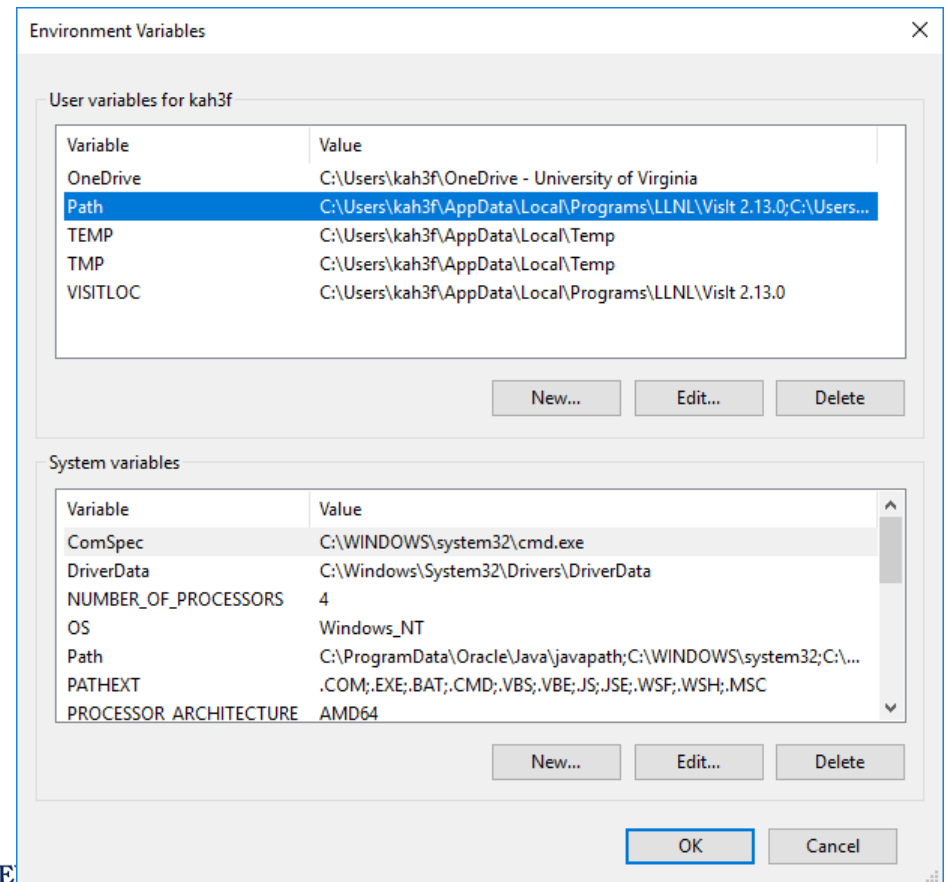
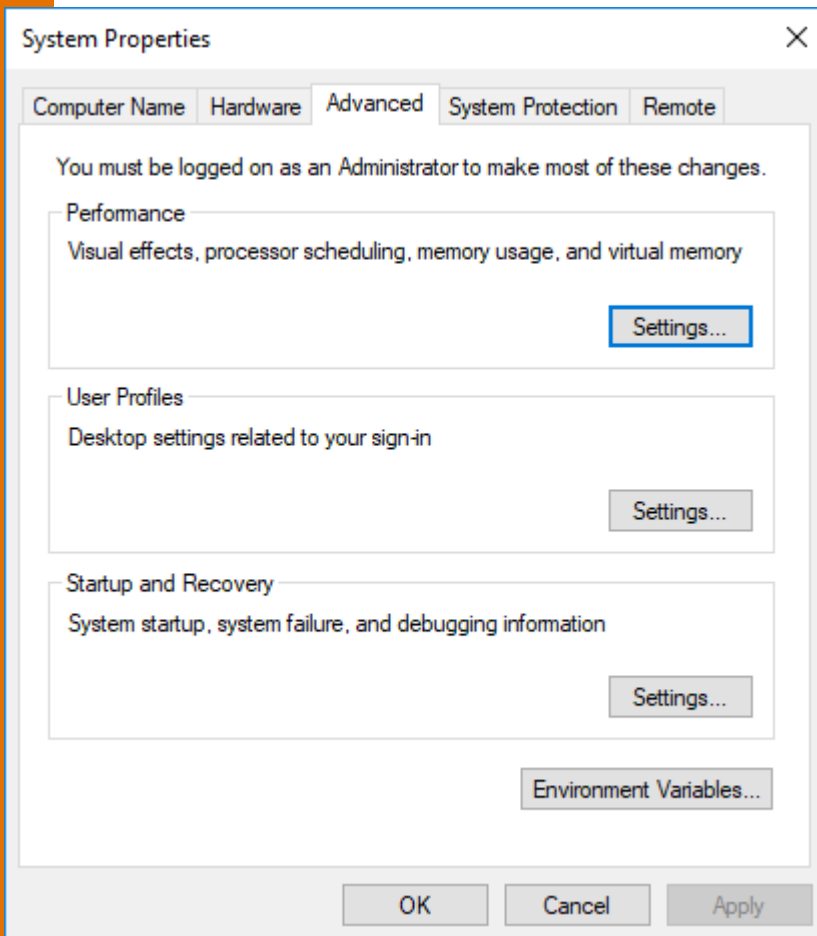
The screenshot shows a window titled "MinGW Installation Manager Setup Tool". At the top, it displays "mingw-get version 0.6.2-beta-20131004-1". Below this is a logo consisting of a red cube and a blue cube. The text "Written by Keith Marshall" is centered, followed by "Copyright © 2009-2013, MinGW.org Project" and the URL "http://mingw.org". A large paragraph of text states: "This is free software; see the product documentation or source code, for copying and redistribution conditions. There is NO WARRANTY; not even an implied WARRANTY OF MERCHANTABILITY, nor of FITNESS FOR ANY PARTICULAR PURPOSE." Below this, another paragraph says: "This tool will guide you through the first time setup of the MinGW Installation Manager software (mingw-get) on your computer; additionally, it will offer you the opportunity to install some other common components of the MinGW software distribution." A third paragraph explains: "After first time setup has been completed, you should invoke the MinGW Installation Manager directly, (either the CLI mingw-get.exe variant, or its GUI counterpart, according to your preference), when you wish to add or to remove components, or to upgrade your MinGW software installation." At the bottom, there are three buttons: "View Licence", "Install" (which is highlighted with a blue border), and "Cancel".

[illegible]

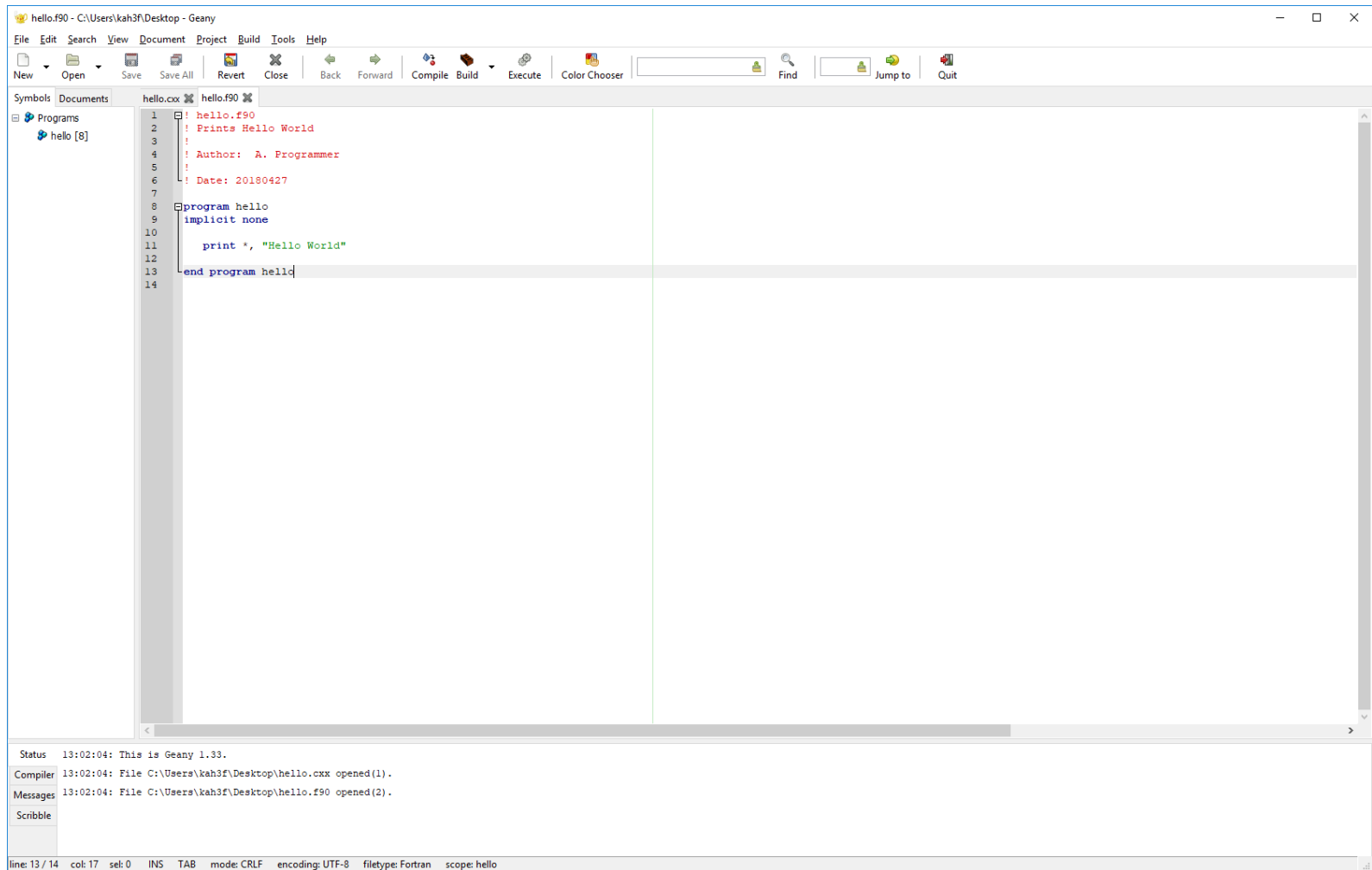
ENVIRONMENT VARIABLES IN WINDOWS

Control Panel->System and Security->Advanced system settings->Environment Variables

Once you open Path, click New to add to the Path



GEANY ON WINDOWS



VARIABLES IN FORTRAN

VARIABLES IN FORTRAN

- Unlike most languages, Fortran is *not* case sensitive. Variables `Mean`, `mean`, and even `mEan` are the same to the compiler.
- Like most compiled languages, Fortran is *statically typed*. All variables must be *declared* to be of a specific type before they can be used. A variable's type cannot be changed once it is declared.
- Fortran is (nearly) strongly typed. Mixed-mode expressions are limited and most conversions must be explicit.

NUMERIC TYPES: INTEGER

- Integer
 - Quantities with no fractional part
 - Represented by sign bit + value in *binary*
 - *Remember that computers do not use base 10 internally*
 - Default integers are of size 32 bits
 - Maximum integer is $2^{32}-1$
 - All Fortran integers are signed
 - Compiler extension (in nearly all compilers)
 - `INTEGER*8` (old declaration style) is a 64-bit integer
 - Will show another method when we learn about `KIND`

NUMERIC TYPES: SINGLE PRECISION

- Floating point single precision
 - Called `REAL` in Fortran
 - Sign, exponent, mantissa
 - 32 bits
 - IEEE 754 defines representation and operations
 - Approximately 6-7 decimal digits of precision, *approximate* exponent range is 10^{-126} to 10^{127}

NUMERIC TYPES: DOUBLE PRECISION

- Double precision floating point
 - Sign, exponent, mantissa
 - 64 bits
 - Number of bits NOT a function of the OS type! It is specified by the IEEE 754 standard!
 - Approximately 15-17 decimal digits of precision, approximate exponential range 10^{-308} to 10^{308}
 - In Fortran the default *literal* is single precision. Double precision literals *must* include a d/D exponent indicator.
 - Forgetting to write DP literals with D rather than E often causes a significant loss of precision that is hard to find.

NUMERIC TYPES: COMPLEX

- A complex number consists of 2 reals enclosed in parentheses
 - Single-precision type is `COMPLEX`
 - $z = (r, i)$
 - Most compilers provide the
`DOUBLE COMPLEX`
extension as a variable type

NON-NUMERIC TYPES: LOGICAL

- Booleans are called `logical` in Fortran.
- Values can be `.true.` or `.false.` (periods required)
 - Are not necessarily represented by integers; internal representation is up to the compiler.
 - Cannot even be cast to an integer.

NON-NUMERIC TYPES: CHARACTER

- Character
 - 1 byte (8 bits) per single character
- A character has a fixed length that must be declared at compile time

```
character(len=8) :: mychar
```
- In subprograms a character of unspecified length may be passed

```
character(len=*) :: dummy
```

 - Fortran 2008 has a variable character length but this is beyond our scope.
- Note that character in Fortran really means a fixed-length string. The default length is 1, however.

```
character :: letter
```

LITERALS

- Literals aka constants

- Specified values e.g.

- 3

- 3.2

- 3.213e0 (Fortran single precision)

- 3.213d0 (Fortran double precision)

- 3.213_rk (Determined by kind parameter rk)

- "This is a string"

- "Isn't it true?"

- 'Isn't it true?'

- .true.

- (1.2, 3.5) (Fortran complex)

- Literals have a type but it is determined from the format rather than a declaration.

VARIABLE DECLARATIONS

- Variables are declared by indicating the type followed by a comma-separated list of variables.
- In older code no separator was used.

```
INTEGER  i, j, k
```

- In newer code (including all new code you write) use the double colon to separate the type from the variable list

```
INTEGER  ::  i, j, k
```

- If there are other attributes on the line the `::` will be *required*.

DECLARATIONS

- First statement should be

```
PROGRAM myname
```

- Then follow it immediately with

```
IMPLICIT NONE
```

Declare variables with double-colon syntax

```
INTEGER           ::  I, J
REAL              ::  R, S, T
DOUBLE PRECISION  ::  D
DOUBLE COMPLEX    ::  Z
LOGICAL           ::  FLAG
CHARACTER (len=20) ::  C
```

- Line up declarations neatly.
- All caps are **not** required but I use them to emphasize the keywords.

INITIALIZING AT COMPILE TIME

- Variables can be declared and initialized at the same time:

```
real    :: x=1.e-8, y=42.
```

- When variables are initialized in this manner it happens only *once*, at compile time. If this takes place in a subprogram it will not happen again upon repeated invocations.

- It is equivalent to the older DATA statement

```
data x,y/1.e-8,42./
```

- In Fortran 2003 it became possible to initialize using intrinsic functions:

```
real    :: pi = 4.0*atan(1.0)
```

EXAMPLE

- Start Geany, Code::Blocks, or another editor. Type

```
program first
```

```
! My first program
```

```
! Author:   Your Name
```

```
implicit none
```

```
    real      :: x,y
```

```
    integer   :: i,j=11
```

```
    x=1.0
```

```
    y=2.0
```

```
    i=j+2
```

```
    print *, "Reals are ",x,y
```

```
    print *, "Integers are ",i,j
```

```
end program
```

PARAMETER

- In compiled languages, programmers can declare a variable to have a fixed value that cannot be changed.
- In Fortran this is indicated by the `PARAMETER` attribute.

```
real, parameter :: pi=3.14159
```

- Attempting to change the value of a variable declared to be a parameter will result in a fatal compiler error.
- In older code the declaration and parameter statement will be on different lines

```
real pi  
parameter (pi=3.14159)
```


TYPE CONVERSIONS

- Most compilers will automatically cast numeric variables to make mixed expressions consistent. The variables are promoted according to their rank. Lowest to highest the types are integer, float, double, complex.
- Use explicit casting to be clear, or in circumstances such as argument lists where the compiler will not do it.
- The new way to cast numbers is via `KIND`. Older conversion functions such as `db1e` can still be used and will be in older code.
- Logicals cannot be cast to anything.
- Strings may be cast to numbers and vice versa by a fairly idiosyncratic method.

EXAMPLES

- Explicit casting among numeric types, default kind.

```
R=real (I)
```

```
I=int (R)
```

```
Z=cmplx (r1, r2)
```

```
D=dbl (R)
```

CHARACTER ↔ NUMERIC

- Fortran has a peculiar way to do this called internal read/write.
- Convert numeric to character:

```
character(len=4)    :: age
integer             :: iage
    iage=39
    write(age, '(i4)') iage
```

- Convert character to numeric

```
age='51'
    read (age, '(i4)') iage
```

ARITHMETIC OPERATIONS

- Operators are defined on integers, floats, and doubles
- $+$ $-$ add subtract
- $*$ $/$ multiply divide
- $**$ exponentiation
- Operator Precedence is:
 - $**$ $(* /)$ $(+ -)$
- Evaluation is left to right by precedence unless told otherwise with parentheses

INTEGER OPERATORS

- In Fortran $2/3$ is always zero! Why?
 - Because 2 and 3 are both integers. Nothing will be promoted to a float, so $/$ is an integer operation that yields an integer result
- Remainder comes from `mod(n, d)` or `modulo(n, d)`
 - `mod` and `modulo` are NOT THE SAME for negative numbers
 - `mod` is most frequently used though `modulo` is closer to other languages' $\%$ operator. Use for negatives is uncommon in all languages.

CHARACTER (STRING) OPERATORS

- Strings/Characters

- There are many (some of which require function calls)
- Concatenation //
- Substring extraction

`S (1 : 3)`

The first character is counted as 1 and the last one in the substring is the actual upper bound. This expression extracts characters 1 to 3 *inclusive*.

- Fortran counts from 1 and the upper bound is included in the range.

CONDITIONAL OPERATORS

- Numeric
 - Fortran has two sets, one with letters and one with symbols. Note that /= has a / for “not.”

.eq. ==

.ne. /=

.lt. <

.gt. >

.le. <=

.ge. >=

LOGICAL OPERATORS

- Negation

`.not.`

`.not. flag`

- AND

`.and.`

- OR

`.or.`

CONDITIONAL OPERATOR PRECEDENCE

- `>`, `>=`, `<`, `<=` outrank `==` or `/=`
- `==`, `/=` outranks `.and.`
- `.and.` outranks `.or.`
- As always, use parentheses to change grouping or to improve clarity.

CHARACTER COMPARISON INTRINSICS

`lge(stringA, stringB)`

- Returns `.true`. If `stringA` is lexically greater than or equal to `stringB`, otherwise returns `.false`.

`lgt(stringA, stringB)`

- Returns `.true`. If `stringA` is lexically greater than `stringB`, otherwise returns `.false`.

`lle(stringA, stringB)`

- Returns `.true`. If `stringA` is lexically less than or equal to `stringB`, otherwise returns `.false`.

`llt(stringA, stringB)`

- Returns `.true`. If `stringA` is lexically less than or equal to `stringB`, otherwise returns `.false`.

EXPRESSIONS AND STATEMENTS

EXPRESSIONS IN FORTRAN

- Fortran expressions are much like those of other languages.

`a+3*c`

`8.d0*real(i,dp)+v**3`

`phase+cmplx(0.,1.)`

`sqrt(abs(a-b))`

`A .or. B`

`y > 0.0 .and. y < 1.0`

`myfunc(x,y)`

STATEMENTS

- A Fortran peculiarity: statements are *executable* or *non-executable*. Non-executable statements are instructions to the compiler (variable declarations, interfaces, etc.) Executable statements perform some action. All non-executable statements must *precede* the first executable statements in a program unit.
- Indentation is not required but *should be* used!
- No semicolons should be used at the end of the line.
- Multiple statements (keep them short) may be written on the same line if they are separated by semicolons.

FIXED FORMAT VERSUS FREE FORMAT

- Prior to the Fortran 90 standard, Fortran code was required to conform to rigid column rules based on the layout of punched cards.
 - This may be another reason that computer scientists sneer at it 😊
 - Statements began in column 7 and could extend to column 72. Column 6 was reserved for continuation marks. Columns 1-5 were for statement labels. Columns 73-80 were ignored (and were used to number cards)
- In Fortran 90 and up, there are no column rules. This is called free format.

COMMENTS, CONTINUATIONS, ETC.

- Fixed format comment:
 - C or c in the first column meant the entire line was a comment.
- Free format comment:
 - Anything from ! to the end of the line is ignored.
- Fixed format continuation:
 - Number or printable character in the 6th column.
- Free format continuation:
 - Ampersand & at the end of the line to be continued.

STATEMENTS FOR THE MAIN PROGRAM

- A program may optionally begin with a `PROGRAM` statement which is optionally followed by its name.

```
PROGRAM myprogram
```

- The program must end with an `END` statement. Optionally it may be

```
END PROGRAM <name>
```

- I strongly recommend use of the longer, more descriptive forms.
- Execution may be stopped with the `STOP` statement. `STOP` is required only for abnormal termination. It can optionally be followed by a message, which it will print to standard output.

```
STOP "Attempt to divide by zero."
```

IMPLICIT

- For historical reasons, Fortran can use implicit typing. By default, variable names beginning in A–H and O–Z are REAL (single precision) while variable names beginning in I–N (the first two letters of INteger) are integers.
- IMPLICIT statements change this behavior. Older code often changes the default float to double

```
IMPLICIT DOUBLE PRECISION (a-h, o-z)
```

- New code should always cancel implicit typing with

```
IMPLICIT NONE
```

This requires that all variables be declared and will catch many “typo” bugs.

- The IMPLICIT statement must appear in each program unit.

STATEMENT LABELS

- In fixed format code, *statement labels* were often used.
- In fixed format statement labels must be integers and must occupy a maximum of five digits, at least one of which must be nonzero.
- In free format there is less need for labels and they do not always need to be integers.
- Any statement that is not part of a compound (semicolon-separated) statement can be labeled with an integer.

MISCELLANEOUS

- The no-op is `continue`
 - It was often used in old code since do loops required a labeled statement as the terminator.

```
do 100 i=1,n
    statements
100 continue
```
 - We use `end do` now for this purpose. However, `continue` is a convenient target for labels in input/output statements.
- Fortran has a `go to` (or `goto`) statement.

```
go to <label>
```

 - Despite computer science hatred for it, it is still sometimes useful and in some situations it is far more readable than the contortions required to replace it!
 - As a rule `goto` should always direct the flow downward, never upward.

HELLO WORLD

```
program hello
implicit none
    integer    :: i
    real       :: pi=0.25*atan(1.0)
    i=42
    print *, 'Hello! The answer is',
&          i, ' and pi is
',pi
end program
```

EXERCISE

- Write a program that will set variables as indicated and will print the expressions indicated. Use print * to print to the console. Invoke implicit none and declare all your variables.

```
x=17.  
Xs=11.  
num_1=10  
num_2=14
```

```
print *, x  
print *, Xs/x  
print *, int(Xs/x)  
print *, int(Xs)/int(x)  
print *, Xs/x + x  
print *, Xs/(x+x)  
print *, x/num_1  
print *, num_1/num_2  
print *, num_2/num_1
```

EXERCISE

- Declare character variables large enough to hold the indicated strings. Make `newtitle` at least 5 characters longer than you think necessary.

```
title="This is a string"
subtitle="Another string"
print *, len(title)
print *, title//": "//subtitle
newtitle=title//": "//subtitle
print *, len(newtitle)
print *, len_trim(newtitle)
print *, newtitle(2:4)
!Change "This" to "That" in newtitle
```

EXERCISE

Exercises with conditionals.

Be sure to declare variables appropriately.

```
a=11.; b=9.; c=45.; n=3
```

```
print *, a>b
```

```
print *, a<b and c==n
```

```
print *, a<b or c==n
```

```
print *, a>b or c==n and a<b
```

```
print *, (a>b or c==n) and a<b
```

```
is_equal= a==b
```

```
print *, is_equal
```


KIND

KIND

Variables in modern Fortran may have a *kind* associated with them. The programmer requests at least a certain number of decimal digits of precision and at least a certain exponent range, and the system matches the request as best it can.

In practice, most systems have single and double precision. A few offer quad precision (`REAL*16` in older nomenclature) but it is usually done in software and is *very* slow.

Compilers still support `REAL` and `DOUBLE PRECISION` (`REAL*8` was never standard, but is fairly universally supported). The advantage to `KIND` is that it becomes easy to change precision, especially when using a module.

OBTAINING KIND INFORMATION

- NEVER assume that the `KIND` type integer equals the number of bytes. Use intrinsics to obtain or use `KIND`. The range is 10^{range}

```
ik=selected_int_kind(range)
```

```
rk=selected_real_kind(prec, range)
```

`kind(x)` returns the kind type parameter of `x`

```
ik=kind(1)
```

```
rk=kind(1.0)
```

```
dk=kind(1.0d0)
```

DECLARING KIND

- For the IEEE 754 standard, the two kinds supported in hardware can be selected with:

```
INTEGER, PARAMETER :: rk=kind(1.0)
```

```
INTEGER, PARAMETER :: rk=kind(1.d0)
```

The kind variable needs to be a `parameter`.

- Variables are then declared as

```
REAL(rk)    :: r, s, t
```

- Literals can be written as

```
1.0_rk
```

- Switching between single and double precision is then as easy as replacing the `SELECTED_REAL_KIND` or `KIND` statement. It is best to use a module for this purpose.

TYPE CONVERSIONS WITH KIND

The `kind` argument is optional except for `real` converting a floating-point type to another FP type.

```
aint(a,kind)    !truncates a float
aint(a,kind)    !nearest integer
ceiling(a,kind)
cmplx(x,y,kind)
floor(a,kind)
int(a,kind)     !truncates, casts to integer
nint(a,kind)    !nearest integer, casts to integer
real(a,kind)    !converts integer to float or &
                between float types
```

- Example: explicit casting with kind
 - Given that `dp` has been declared to match double:
`x=real(w,dp)`
converts `w` from single to double precision.

FORTRAN LOOPS AND CONDITIONALS

CONDITIONALS

`elseif/else if` and `else` are optional. The parentheses around the conditional are required.

```
if ( comparison ) then
    code
elseif ( comparison) then
    more code
else
    yet more code
endif
```

SELECT CASE

- Many else ifs can become confusing.
 - Expression must be character, integer, or logical
 - Ranges only applicable for numeric or character expressions

```
SELECT CASE (expression)
    CASE (:value0)      ! Expression <= value0
        code
    CASE (value1)
        code
    CASE (value2)
        code
    CASE (value3:)      ! Expression >=value3
        code
    CASE (value4,value5,value6) !Multiples OK
        code
    CASE (value7:value9) !Inclusive
        code
    CASE DEFAULT        ! Optional
        code
END SELECT
```


SELECT EXAMPLE

```
select case (x)
  case (:0)
    y=-x
  case (1)
    y=x+3.
  case (2:9)
    y=float(x)/2.
  case (10:20)
    y=float(x)/3.
  case default
    y=0.
end select
```

DO LOOP

- DO is the equivalent of FOR in languages like C/C++.
- DO executes a fixed number of iterations unless explicitly terminated.
- DO can iterate only over integer sequences.

```
INTEGER    :: L, U, S
```

```
INTEGER    :: I
```

```
DO I=L,U,S
```

```
...
```

```
END DO
```

I: Loop variable

L: Lower bound

U: Upper bound

S: Stride. Equal to 1 if not present

S can be negative, in which case L must be greater than U.

QUIZ

- The standard requires that loop variables be integers. How would I implement loop variables that are real?
- How might real loop variables be a problem?

IMPLIED DO

- The implied do is used in a few circumstances, specifically input/output and array construction.

`(var(iterator), iterator=lbound, ubound, s)`

The parentheses are required.

`(a(i), i=1, 20)`

- Implied do loops can be nested.

`((r(i, j), j=1, M), i=1, N)`

EARLY EXIT

`exit`: leave loop

- `exit` is able to break out of *only* the loop level *in which it appears*. It cannot break from an inner loop all the way out of a nested set of loops. This is a case where `goto` is better than the alternatives. Equivalent to Python/C/C++ `break`

`cycle`: skip rest of loop and go to next iteration. Equivalent to Python/C/C++ `continue`

WHILE LOOPS

```
do while (<logical expression>)  
    statement  
    statement  
    ...  
end do
```

- Remember that your logical expression must become false at some point.

EXAMPLE

```
integer  :: x, y, z
x=-20
y=-10
do while (x<0 .and. y<0)
    x=10-y
    y=y+1
    z=0
enddo
z=1
```

EXIT/CYCLE

```
x=1.  
do while (x>0.0)  
    x=x+1.  
    if (x>=10000.0) exit  
    if (x<100.0) cycle  
    x=x+20.0  
enddo
```


REPEAT-UNTIL

```
do
    statement
    statement
    ...
    if (<logical expression>) exit
end do
```

do while always tests at the *top* of the loop.
The do ... if/exit form can test anywhere.

EXAMPLE

- Reading a file of unknown length

```
nlines=0
```

```
do
```

```
    read(unit=iunit, end=10) var
```

```
    nlines=nlines+1
```

```
enddo
```

```
10 continue
```

EXAMPLE

```
program second
implicit none
  integer :: x, y, z
  x=-20
  y=-10
  do while (x<0 .and. y<0)
    x=10-y
    y=y+1
    z=0
  enddo
  z=1
  print *, x, y, z
end program
```

EXERCISE

- Loop from 0 to 20 by increments of 2. Make sure that 20 is included. Print the loop variable at each iteration.
- Start a variable n at 1. As long as n is less than 121, do the following:
 - If n is even, add 3
 - If n is odd, add 5
 - Print n for each iteration. Why do you get the last value?
- Set a real value $x=0$. Loop from 1 to N inclusive by 1.
 - If the loop variable is less than M , add 11. to x .
 - If $x > w$ and $x < z$, skip the iteration.
 - If $x > 100.$, exit the loop.
 - Experiment with different values for the variables. Start with $N=50$, $M=25$, $w=9.$, $z=13$.

ARRAYS

TERMINOLOGY

- A *scalar* is a single item (real/float, integer, character/string, complex, etc.)
- An *array* contains data of the **same type** with each scalar element addressed by *indexing* into the array.
- An array has one or more *dimensions*. The *bounds* are the lowest and highest indexes. The *rank* is the number of dimensions.
- Fortran arrays are similar to NumPy arrays and also carry metadata (shape, size, bounds, some other data) about themselves.

FORTRAN ARRAYS

- Arrays must be declared by type and either by size or by some indication of the number of dimensions.
 - We will do variable dimensions later

```
REAL, DIMENSION(100) :: A
```

By default the index starts at 1. However, it can start at any integer less than the upper bound:

```
REAL, DIMENSION(-1:101, 0:3) :: A0
```

- Arrays may have zero size.
- Maximum (standard) dimensions <=F2003 is 7. Increases to 15 in F2008.

ORIENTATION

- Array elements are *adjacent* in memory (this is one of their advantages) and are arranged linearly no matter how many dimensions you declare. If you declare a 3x2 array the order in memory is

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

- “Orientation” refers to how the array is stored *in memory*, not to any mathematical properties.
- Fortran is *column-major* oriented. Most other languages are *row-major* oriented.
- Loop indices should reflect this whenever possible (when you need loops).
- Fortran: outermost first. Go right to left.

$A(i, j, k)$ loop order is `do k/do j/do i`



ARRAY ELEMENTS AND SLICES

- Each element can be addressed by its index or indices, enclosed in *parentheses*.

`A (3)`

`X (i , j)`

- Remember, starts at 1 by default

- Slices (subarrays)

`REAL, DIMENSION (100) :: A`

`REAL, DIMENSION (12) :: B`

`INTEGER, DIMENSION (20,10) :: N`

`INTEGER, DIMENSION (20) :: C`

`B=A (1:12)`

`C=N (:, i) !ith column of N`

ARRAY INITIALIZATION

- Arrays can be initialized to the same quantity by an array operation

`A=0.0 !equivalent to A(:)=0.0`

- For small arrays, an array constructor can be written.

`I=[1,0,0,0]`

- The constructor can use a construct called an implied do (which is somewhat similar to a Python list comprehension):

`X=[(real(i), i=1, 100)]`

ARRAY OPERATIONS

- Most of the mathematical functions are *overloaded* to accept array arguments. They operate on the array(s) *elementwise*.

`T=3.0`

`A=3.14159*I`

`B=sin(A)`

`C=A/B !Watch out for zero elements`

ARRAY INTRINSICS

- Modern Fortran has many intrinsic functions that operate on arrays.
- Array intrinsics can be classified as inquiry, construction and manipulation, and transformation/reduction.

ARRAY CONSTRUCTION INTRINSICS

`reshape (source, shape [, pad] [, order])`

`merge (array1, array2, mask)`

`pack (array, mask [, vector])`

`unpack (vector, mask, field)`

`spread (source, dim, ncopies)`

ARRAY INQUIRY

INTRINSICS

- `allocated(array)`
 - returns `.true.` or `.false.`
- `lbound(array), ubound(array)`
- `shape(array)`
 - returns a rank-one array containing the dimensions
- `size(array, [dim])`
 - returns the size (the total number of elements). If the optional argument `dim` is not present it returns the total number of elements; if `dim` is present it returns the number of elements on that dimension.

ARRAY TRANSFORMATION INTRINSICS

- `transpose(matrix)`
 - matrix must be square and rank 2
- `dot_product(V1,V2)`
 - both must be rank 1
- `matmul(A,B)`
 - A and B must conform, must return a rank-2 array even if it's (1,1)
 - Note: depending on compiler and version, might be slow
- `minloc(array [,mask])` !first one it finds
- `minloc(array, dim [,mask])`
- `maxloc` (as above)
- `minloc/maxloc` return a rank-1 array of *indices*.

ARRAY REDUCTION INTRINSICS

- `minval(A [,dim])`
 - `maxval(A [,dim])`
 - `all(mask [,dim])`
 - `any(mask [,dim])`
 - `count(mask)`
 - `product(A [,dim])`
 - `sum(A [,dim])`
 - **Example:** A has shape (4, 5, 6)
`sum(A, 2)` has shape (4, 6) and elements
`sum(A(i, :, j))`
- >Same behavior as similar NumPy built-ins.

EXAMPLE

- Open a new file (you can call it arrays.f90) and type

```
program arrays
```

```
! Array demo
```

```
implicit none
```

```
real, dimension(100)    :: A
```

```
real, dimension(10,10)  :: B
```

```
integer, dimension(2)   :: shaper=[10,10]
```

```
integer                 :: i
```

```
A=[ (0.1*real(i), i=1,100) ]
```

```
B=reshape(A, shaper)
```

```
print *, B(1:3,1:4)
```

```
end program
```

ALLOCATABLE ARRAYS

- Arrays may be sized at runtime by making them *allocatable*. They are declared with an `allocatable` attribute and a colon for each dimension.

```
REAL, ALLOCATABLE, DIMENSION(:) :: A, B
```

- If any dimension is `allocatable`, all must be.
- They must be allocated before they are used, so their size must be known at runtime.

```
ALLOCATE (A (NMAX) , B (MMAX) )
```

- Check whether an array is allocated with `allocated(A)`

```
if (allocated(A)) then  
    do something  
endif
```

EXERCISE

- In arrays.f90
 - Print the size of the array A
 - Change the fourth element to 1.1
- Declare a new real array W of rank 2 (two dimensions) and make it allocatable. Allocate the array to size 10x10 and set all values to 1.0 using an array operation.
- Use a nested loop to set each element of W(i,j) to the sum of its row and column indices. Fortran loops should run over indices from right to left, e.g.

```
do j=1,N
  do i=1,M
    A(i,j)=something
  enddo
enddo
```

MASKING ARRAY OPERATIONS

- Masks can be constructed and used to perform bulk operations on arrays.

```
real, dimension(100):: x
```

- assign x somehow

```
if ( all(x>=0.0) ) then
```

```
  y=sqrt(x)
```

```
endif
```

ADVANCED ARRAY INDEXING

- Arrays can be addressed with arrays of integers (but not logicals).

```
integer, dimension(1) :: maxtemp
```

```
real, dimension(365) :: temps
```

```
character(len=5), dimension(365) :: dates
```

```
maxtemp=maxloc(temps)
```

```
print *, "maximum temp was at ", dates(maxtemp)
```

CONDITIONALS WITH ARRAYS

- Logical arrays can be assigned with conditionals derived from other arrays to construct masks.

```
logical, dimension(365) :: is_max
integer                :: day
is_max=temps==maxval(temps)
do day=1,size(is_max)
    if (is_max(day)) then
        print *, dates(day)
    endif
enddo
```

WHERE

- Where functions like a “vectorized” loop+conditional.
- The clauses must be array assignments.

```
where ( A >= 0.0 )
```

```
    B = sqrt(A)
```

```
elsewhere
```

```
    B = 0.
```

```
end where
```

EXERCISE

- In arrays.f90
 - Add a WHERE to set elements to W equal to the corresponding elements of B if the element of B is less than 7.

FORTRAN INPUT/OUTPUT

LIST-DIRECTED IO

- List-directed IO allows the compiler to format the data.
- Input
 - Fortran read from standard input. Separates values on comma or whitespace.

```
read(*,*) var1, var2, var3
```
 - Fortran write to standard output

```
print *, var1, var2, var3
```

```
write(*,*) var1, var2, var3
```
- In Fortran the `print` statement always writes an EOL after all variables have been output. The `write` statement does as well unless told otherwise (this is the opposite of `write` in most other languages).

READING FROM THE COMMAND LINE

- We can read strings only. You must convert if necessary to a numerical type using internal read/write.

```
nargs=command_argument_count()
if ( nargs .ne. 1 ) then
    stop "No input specified"
else
    call get_command_argument(1,nval)
    read(nval, '(i4)') n
endif
```

FORMATTED IO

- In Fortran it is best to avoid formatted *input* as much as possible, as it can lead to errors.
- Formatted output, on the other hand, is frequently required for legibility. Compilers tend to let list-directed output sprawl.
- Formatted output is similar to other languages (in fact, they all got it from Fortran, the oldest higher-level programming language).

EDIT DESCRIPTORS

- The edit descriptor modifies how to output the variables. They are combined into forms like

$RaF.w$

where R is a repeat count, a is the descriptor, F is the total field width *including* space for $+-$, and if requested $+-e$ and exponent, and w is the number of digits to the right of the decimal point. $Ra.w$ alone works and $Ra.0$ will print the integer part.

- Strings take only F (RaF) and do not usually require the F since the length will be known to the compiler.
- Integers can be written as iF and any of the F spaces not needed will be blank filled, with the digits right justified. When written as $iF.m$ they will be printed with at least m digits and the rest of the field zero-filled on the left if all of F is not needed.

COMMON EDIT DESCRIPTORS

- As usual, they are not case sensitive.
 - I integer
 - F real (decimal output)
 - E real (exponential output)
 - G general
 - D double precision (prints D rather than E for exponent)
 - A character (does not require a field width in most cases)
 - X space
- The real descriptors F, E, G, and D all work for both single and double precision. G allows the compiler to choose whether to use decimal or exponential format.

MODIFIERS

- Some modifiers can change the appearance of the output.
- p multiply by 10. k_p multiply by 10^k . Applies till the next scale factor is encountered.
- ES use scientific notation. The default exponential format writes in machine normalization, with the leading digit between 0 and 1. ES causes it to write with the leading digit between 1 and 9, which is what most humans can read most easily. Ignores p on output.
- / write an EOL and go to the next line (record) within the format

FORMAT STRINGS

- The format string is constructed as a list of how to output the variables. Unlike some other languages, literal strings are never included, but must have their own edit descriptors.
- The format string can be placed directly into the `write` statement or it can be in a separate `format` statement. In the `write` it is enclosed in parentheses and quotes.
- For most purposes it is best to put the format string into the `write` statement. The format statement is older and will be in old code, but it is usually harder to see what is happening. It is useful for particularly long strings, however.

EXAMPLES

```
write(*, '(i5,2x,i6)') i1,i2
write(*, '(i5,a,i6)')) i1," ",i2
write(*, '(a,i4,es15.7)') "row",n,var
write(*, '(3(i2,3x,f8.3)') (r(j),var(j),j=1,3)
write(*, '(2f8.2)') z !complex
write(*, '(2L)') is_zero,is_finite
write(*, '(2p,f8.2,0p,f8.2)') var1, var2
write(*, '(a,f8.2,/ ,a,i6)') mess1,x,mess2,i
```

FORMAT STATEMENTS

- Format statements are abundant in older code, before the strings could be inserted into writes.
- `FORMAT` is non-executable but can appear anywhere in the source. It is the only non-executable statement that can do so.
- It can still be useful for a particularly complex format (to keep the write statement short and readable) or for formats that are repeated in many write statements.
- The second parameter to the write is then an integer statement label. The label marks the format statement.

FORMAT EXAMPLE

```
      write(*,100)  x,y,z  
100  format(3e15.8)
```

- Traditionally the format is placed immediately below the line which refers to it, or else all format statements are grouped together just before the end statement of their program unit.

FORTRAN NON-ADVANCING IO

- If we'd like to write to and read from standard input on the same line we can use non-advancing IO:

```
write(*, '(a)', advance='no') "Enter input value:"  
read(*,*) value
```

- *Must* be formatted
 - 'yes' for advance is valid also but is the default.
 - Argument to `advance` can be a character variable so you can decide based on conditionals to advance or not.

EXERCISE

- Write a program that computes pi using a trig identity such as $\pi = 4 * \text{atan}(1)$
- Use kind to switch between real and double precision
 - integer, parameter :: rk=kind(1.0) or (1.0d0)
- Using single precision, print pi in
 - E format
 - Scientific notation
 - Scientific notation with 8 decimal places
- Repeat for double precision

EXERCISE

- In an “infinite” while loop:
- Request an integer from the user with non-advancing input/output, e.g.
- “Please enter an integer:” <then read integer>
- If the integer is 1, print “zebra”. If it is 2, print “kangaroo”. If it is anything else except for zero, print “not found”. If it is 0, exit the loop.

FORTRAN FILE IO

OPEN

- Files are identified with integers called *unit numbers*.

```
open(unit=iunit, file=fname, end=10)
```

There are many other options. Only `iunit` and `fname` are required. If the unit argument is first it does not need the `"unit="` keyword.

On Unix file names will be *case sensitive*.

- The unit number is assigned by the programmer.
- In Unix unit 5 is conventionally standard input and unit 6 is standard output. Standard error is not as uniform but it is usually unit 2.
- Programmers can reassign units 2, 5, and 6, but it is strongly advised that you not do so.

POPULAR OPTIONS TO OPEN

`iostat=ios`

- Returns status into the integer variable `ios`. If zero the statement succeeded, otherwise it failed. Specific nonzero values are system-dependent.

`err=label`

- Jumps to statement labeled `label` if an error occurs.

`end=label`

- Jumps to statement labeled `label` on end of file

`status=stat`

- `stat` can be `'old'`, `'new'`, `'replace'`, `'scratch'`, or `'unknown'`. The default is `'unknown'` (read/write). If `'old'` it must exist, and if `'new'` it must not exist. A `'scratch'` file is automatically deleted after being closed.

`form=fmt`

- `fmt` is `'formatted'` (default, text) or `'unformatted'` (a system-dependent binary format).

`access=acc`

- `acc` can be `'sequential'` (default), `'direct'`, or `'stream'`. Direct files must be unformatted.
- Unless access is `stream`, an unformatted file will have a header and footer that is specific to a compiler and platform and may not be portable.

`position=pos`

- `pos` is `'asis'` (default), `'rewind'`, or `'append'`. Rewind returns the file pointer to the top. Append leaves it at the end of the file.

INQUIRE

- The inquire statement tests the status of a file. Most usually we wish to check whether the file exists, or is already open, before we attempt to open it.

```
inquire (unit=iunit, options)
```

- or

```
inquire (file=fname, optionslist)
```

- So we can inquire by unit or name but not both.

COMMON OPTIONS TO INQUIRE

`iostat=ios`

- Like `open`

`err=label`

- Like `open`

`exist=exists`

- Returns `.true.` or `.false.` into logical `exists`

`opened=is_open`

- Returns `.true.` or `.false.` into logical `is_open`

CLOSE

- Much of the time, it is not necessary to close a file explicitly. Files are automatically closed when execution terminates.
- If many files are opened, it is good practice to close them before the end of the run.

```
close (unit=iunit, iostat=ios, err=ier, &  
                                             status=st)
```

- Status can be 'keep' (default) or 'delete'

```
close (iunit)
```

REWIND

- An open unit can be rewound. This places the *file pointer* back to the beginning of the file.
- The default is to rewind a file automatically when it is closed.
- If you want to rewind the file to reread it, use
`rewind(iunit)`
- Rewind is convenient if the program must handle files whose lengths may vary. Read through the file without storing any variables, count the number of lines, rewind, then allocate any arrays needed.
- If your input files will always be of known length this isn't necessary (or efficient), but often file length could vary with different data.

EXAMPLE

```
nlines=0
do
  read(iunit,*,end=1)
  nlines=nlines+1
end do
1 continue
rewind(iunit)
allocate(obs(nlines))
do n=1,nlines
  read(iunit,*) obs(n)
enddo
```

- QUIZ
- Why do I increment `nlines` *after* the read?
- What would I do if I had one or more header lines?

READING FROM A FILE

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

- For the most part I do not recommend formatted input, but if it is required it is

```
READ(iunit, '(fmtstr)')
```

- or

```
READ(iunit, label)
```

```
label FORMAT(fmtstr)
```

- Each READ statement reads one line (unless you provide a format string and insert a forward slash).

WRITING TO A FILE

```
WRITE (iunit, *)
```

- List IO

```
WRITE (iunit, ' (fmtstr) ')
```

or

```
WRITE (iunit, label)
```

```
label FORMAT (fmtstr)
```

label must be an integer

- Fortran always writes an EOL for each `WRITE` statement; you may add more with the `/` character but there is seldom a need for this.
- If you do *not* want to advance, use `advance='no'`

FORTRAN NAMELIST

- One of the most convenient I/O statements in Fortran is `NAMelist`. With this statement, parameters in an input file can be specified by `name=value` and in any order.
- The namelist must be declared. This is a non-executable statement. Syntax:

```
NAMelist /name/ var1, var2, var3
```

The name is chosen by the programmer.

- The namelist is read with a special form of the `READ` statement

```
read(iunit, name)
```

NAMELIST INPUT

- The input file containing the namelist must follow a specific format. Namelist was not part of the Fortran 77 standard (it was standardized in Fortran 90) so there is some variation. However, the namelist always starts with

`&name`

The variable list follows, with each variable on a separate line and consisting of the `varname=value` pair.

In older code, the namelist frequently ends with another ampersand (&), or `&end`. Also, in Fortran 77 there may be rules about in which column the & can occur.

In Fortran 90, the namelist is terminated with a forward slash /

NAMelist EXAMPLE

- In the program

```
NAMelist /params/ rho, eps, x0
```

```
OPEN(10,file='paramlist.txt
```

```
' )
```

```
READ(10,params)
```

- The input file (Fortran 90 format)

```
&params
```

```
rho=1.3
```

```
eps=1.e-7
```

```
x0=0.0
```

```
/
```

EXAMPLE

```
program third
character(len=80)                                ::infile,outfile
character(len=40)
::common_name,common_name_adj
character(len=20)                                ::species_id
logical                                           ::file_exists
real,      dimension(:), allocatable            ::num_obs,years
integer                                           ::ios
  nargs=command_argument_count()
  if (nargs .ne. 1) then
    stop "No data file specified"
  else
    call get_command_argument(1,infile)
  endif

  inquire(file=infile,exist=file_exists)
  if (.not. file_exists) then
    stop "Data File not found"
  endif
  iunit=11
  open(iunit,file=infile,iostat=ios)
  if (ios .ne. 0) then
    stop "Cannot open file"
  endif
```

EXAMPLE (CONTINUED)

```
nlines=0
do
  read(iunit,*,end=10)
  nlines=nlines+1
enddo
10 continue
rewind(iunit)
! Subtract header line
nyears=nlines-1
allocate(years(nyears),num_obs(nyears))

!Read header
read(iunit,*)

!Read data (only the values of interest)
!Species info is the same on all lines so we dont make an array
do n=1,nyears
  read(iunit,*,iostat=ios) species_id,common_name_adj,
common name, years(n), num_obs(n)
  if (ios .ne. 0) stop "Error reading file"
enddo

!Adjust years to absolute number (known in advance)
years=years+1900
print *, "Read observation ",num_obs(85)," for year ",years(85)
end program
```

EXERCISE

- Write a program that creates a file `mydata.txt` containing four rows consisting of
 - 1, 2, 3
 - 4, 5, 6
 - 7, 8, 9
 - 10, 11, 12
- Rewind the file and read the data back. Write a loop to add 1 to each value and print each row to the console.

SUBPROGRAMS

WHAT IS A SUBPROGRAM

- A subprogram is a self-contained (but not standalone) program unit. It performs a specific task, usually by accepting *parameters* and returning a result to the unit that invokes (calls) it.
- Subprograms are essential to good code practice. Among other benefits, they are
 - Reusable. They can be called anywhere the task is to be performed.
 - Easier to test and debug than a large, catch-all unit.
 - Effective at reducing errors such as cut and paste mistakes.

FUNCTIONS AND SUBROUTINES

- Unlike most languages, Fortran makes a distinction between **functions** and **subroutines**.
- Functions take any number (up to compiler limits) of arguments and return one item. This item can be a compound type.
- Functions must be declared to a type like variables. (Better: use an interface.)
- Subroutines take any number of arguments (up to the compiler limit) and return any number of arguments. All communication is through the argument list.
- If they are in the same file as the calling unit, subprograms *follow* the caller.

INTENT

- In order to manage side effects, Fortran 90 introduced `INTENT` for declaring subprogram parameters.

```
<type> INTENT(<state>) :: param
```

where *state* can be `in`, `out`, or `inout`.

```
<type> INTENT(in) :: param
```

- It is illegal to change a parameter declared `INTENT(in)`

```
<type> INTENT(out) :: param
```

- The compiler will warn if a parameter declared `INTENT(out)` is never changed.

```
<type> INTENT(inout) :: param
```

- `INTENT(inout)` means that the programmer intends to overwrite the parameter. Thus the programmer makes clear that the side effect is desired or necessary. Parameters passed down to other subprograms must be `inout`.

- Always use `INTENT` in your programs.

FUNCTIONS

The return value is indicated by assigning to the name of the function.

```
FUNCTION myfunc(param1,param2,param3,param4)
    <type>                                :: myfunc
    <type>, INTENT(in) :: param1, param2
    <type>, INTENT(in) :: param3, param4
    statements
    myfunc=whatever
    return                               !Optional unless premature
END FUNCTION myfunc
```

ALTERNATIVE DECLARATION

```
<type> FUNCTION      & !continuation due to PPT
  myfunc(param1,param2,param3,param4)
  <type>, INTENT(in)  :: param1, param2
  <type>, INTENT(in)  :: param3, param4
  statements
  myfunc=whatever
return                !Optional unless premature
END FUNCTION myfunc
```

RENAMING THE RESULT

- Normally the function value is returned by assigning a value to the name of the function.
- We can return it in a different variable with the `RESULT` clause.

```
function summit(x,y) result(s)
```

- Especially used for recursive functions (it is required in this case until F2008).
- When using `RESULT` we declare the type of the name of the `RESULT` rather than the name of the function.

EXAMPLE

```
FUNCTION myfunc(param1,param2) RESULT value
    <type>                :: value
    <type>, INTENT(in)    :: param1, param2
    <type>, INTENT(in)    :: param3, param4
    statements
    value=whatever
return                    !Optional unless premature
END FUNCTION myfunc
```

SUBROUTINES

```
SUBROUTINE mysub(param1,param2,param3)
    <type> INTENT(in)      :: param1
    <type> INTENT(out)     :: param2
    <type> INTENT(inout)   :: param3
    statements
    return                ! Optional unless
premature
END SUBROUTINE mysub
```

INVOKING FUNCTIONS AND SUBROUTINES

- Function

- Invoke by its name

`x=myfunc (z,w)`

`y=c*afunc (z,w)`

A function is just like a variable except it cannot be an *lvalue* (appear on the left-hand side of =)

- Subroutine

- Use the `call` keyword

`CALL mysub (x,y,z)`

PASSING BY REFERENCE

- Fortran passes all parameters by *reference*, meaning that the memory location holding the variable (or its starting position) is what is actually passed.
- This means that any argument can be changed by the subprogram and it will be changed in the caller as well. This is a **side effect**.
- Subroutines operate *entirely* by side effects.
 - Sometimes this is not called a “side effect” when it is intentional, only when it is unintentional.

EXERCISE

1. Write a function that computes Euclidean distance between points x_1, y_1 and x_2, y_2 . Fortran has a built-in `sqrt` intrinsic that you should use.

Write the main program to call this function for

```
x1=-1, y1=2, x2=3, y2=5
```

```
x1=11, y1=4, x2=7, y2=9
```

2. Given two points x_1, y_1 and x_2, y_2 , write a subroutine to determine which is closer to a third point x_3, y_3 . It should pass back a message. You can pass in the points and call the Euclidean distance function from the subroutine, or you can pass in the two distances. (The former would be better programming but if you feel uncertain please go ahead and compute distances separately for now.) Test with

```
x3=10, y3=5
```

You will need to declare the function name in the calling unit as if it were a variable, e.g.

```
real eu_dist
```

PASSING ARRAYS TO SUBPROGRAMS

- Arrays may be passed in one of three ways.
- Static
 - Dimensions are declared as fixed numbers in both calling unit and callee.
- Automatic
 - Dimensions may be passed in the argument list
- Assumed-Shape
 - Only the rank is given, with an appropriate number of colons.

EXAMPLES

```
real, dimension(100) :: A
call sub(A)
subroutine sub(A)
real, dimension(100) :: A ! in sub
```

```
real, dimension(n) :: A
call sub(A,n)
subroutine sub(A,n)
real, dimension(n) :: A ! in sub
integer                :: n
```

```
real, dimension(n) :: A
call sub(A)
subroutine sub(A)
real, dimension(:) :: A ! in sub
```

LOCAL ARRAYS

- Arrays that are local to a subprogram may be sized using an integer passed to the subprogram

```
double precision function myfunc(A,n)
integer,                                intent(in)  :: n
double precision, dimension(n), intent(in) :: A
double precision, dimension(n)           :: B
```

INTERFACES

- If your subprogram is not in a module (to be covered later) you should provide an `INTERFACE`.
- The `INTERFACE` is equivalent to the *prototype* of some other languages.
- Interfaces enable the compiler to check that the *number* and *type* of the argument list in invocations agrees with the declared parameter list.
- Interfaces are nonexecutable and should be placed with (or after) variable declarations.

SYNTAX

INTERFACE

```
function myfunc(x, y, z)
```

```
  implicit none
```

```
  real :: myfunc
```

```
  real :: x, y
```

```
  complex :: z
```

```
end function myfunc
```

END INTERFACE

MORE INTERFACES

```
INTERFACE
```

```
    SUBROUTINE mysub (x, y, z)
```

```
        use mymod
```

```
        implicit none
```

```
        <type> :: x
```

```
        <type> :: y, z
```

```
    END SUBROUTINE mysub
```

```
END INTERFACE
```


INTERFACE BLOCKS

- Only one interface block is required per program unit.

```
INTERFACE
```

```
    function mysub  
        declarations  
    end function mysub  
    subroutine mysub1  
        declarations  
    end subroutine mysub1  
    subroutine mysub2  
        declarations  
    end subroutine mysub2
```

```
END INTERFACE
```

EXERCISE

- Correct your previous exercise with Euclidean distance function and subroutine to use an interface. Each calling unit must have an interface for every subprogram it calls.
- The interface *replaces* declarations of the type of the function name.

SAVING AND DEALLOCATING

- According to the standard, local variables in a procedure are deallocated upon exit from the procedure.
- Allocatable local arrays are automatically deallocated (a form of “garbage collection”)
- If you need some local variables to retain their value from one call to another, use the SAVE keyword

```
SAVE var1, var2, var3
```

```
SAVE
```

- With no variable list it saves all variables
- Allocatable local arrays cannot be saved.

PASSING A SUBPROGRAM NAME

- The name of a subprogram can be passed to another subprogram.
- Example: a numerical-integration subroutine needs the function to be integrated.

```
subroutine trap(f, a, b)
```

where f is a function.

- The unit in which the subprogram receiving the name is called must have an interface for the subprogram to be passed.

OPTIONAL AND KEYWORD ARGUMENTS

OPTIONAL ARGUMENTS

- Subroutines and functions may take optional arguments. Such arguments need not be passed. If they are passed, they take on the passed value. They are declared with the `OPTIONAL` attribute.

```
subroutine mysub(x, y, z, w)
  implicit none
  real, intent(in)           :: x, y
  real, intent(in), optional :: z, w
```

USING OPTIONAL ARGUMENTS

- The call to the previously-defined subroutine could be
`call mysub(a,b)`

in which case c and d would have no values and the subroutine would need to handle that situation appropriately. The call could also be

`call mysub(a,b,c)`

or

`call mysub(a,b,c,d)`

depending on how many of the optional arguments needed to be passed.

KEYWORD ARGUMENTS

- Suppose it were desired to pass `d` but not `c` in the preceding subroutine. The `c` parameter can be skipped by using a *keyword* argument; the optional argument is called as

`dummy=actual`

where `dummy` is its name in the program unit where it is defined, and the `actual` argument is its name in the calling program unit.

Example:

```
call mysub (aa, bb, w=d)
```

- Positional (non-optional) arguments must appear before any optional or keyword arguments.

THE PRESENT INTRINSIC

- The PRESENT () intrinsic function tests whether a particular optional argument is present in the argument list of the caller. If it is not present, defaults can be set or other action taken.

Example

```
IF (PRESENT (w) ) then
```

```
    dd=w
```

```
ELSE
```

```
    dd=3.14
```

```
ENDIF
```

PURE AND ELEMENTAL PROCEDURES

PURE FUNCTIONS

- Side effects should be avoided in functions. Fortran offers subroutines to handle situations where changes to the parameters make sense.
- The programmer can declare a function `PURE` to tell the compiler it is free of side effects.

```
pure function myfunc(x)
```

```
integer :: myfunc
```

```
real, intent(in) :: x
```

- Pure functions must declare all parameters `intent(in)`.

PURE PROCEDURES

- Subroutines may also be `PURE`. They may change their parameters but should declare those as `intent(out)`.
- `PURE` procedures must have an interface.
- Any additional procedures they call must be `PURE`
- Neither pure functions nor pure subroutines are permitted to
 - Alter any accessible global variables (e.g. from `contains`)
 - Perform any IO
 - `SAVE` any variables
 - Contain any `STOP` statement

ELEMENTAL PROCEDURES

- PURE procedures were intended for automatic parallelization. However, a particularly useful derivative is the ELEMENTAL procedure.
- ELEMENTAL procedures operate elementwise on arrays.
- All ELEMENTAL functions must obey the rules for PURE functions. Arguments must be scalars.

```
elemental function f2c(tempF)
    real                :: f2c
    real, intent(in)    :: tempF
    f2c=(tempF-32.)/1.8
end function f2c
```

USING ELEMENTAL PROCEDURES

```
real                :: tempF, tempC
real, dimension(100) :: tempFs, tempCs
real, dimension(10,10) :: dataF, dataC
tempC = f2c(tempF)
tempCs= f2c(tempFs)
dataC = f2c(tempCs)
```

- The elemental function can be called for any arrays as long as they conform. Each element is modified by the function appropriately. It can also be called as a normal scalar function.

EXERCISE

- Write an elemental function that accepts a single parameter as an angle in degrees and converts it to radians.
- Write a program that computes the cosine of 0 to 90 degrees in increments of 5 degrees. Print the output. Do it with a loop, and then by creating a one-dimensional array of angles and passing it to the function.

SCOPE

VARIABLE SCOPE

- In Fortran, scope is defined by the program unit.
- A calling unit may have a variable named `x`, and a function may also have a variable named `x`, and if `x` is not an argument to the function then it will be distinct from the `x` in the calling unit.

```
x=20.
```

```
call sub(x)
```

```
etc.
```

```
subroutine sub(y)
```

```
  real, intent(inout) :: y
```

```
  real                :: x
```

```
  x=10.
```

```
  y=30.
```

```
end subroutine sub
```

CONTAINS AND NESTED PROCEDURES

- The `contains` keyword extends the scope into the contained program unit.
- The end of the “container” must *follow* the end of the “containeer”
- A contained subprogram can access *all* the variables in the container except those that are explicitly passed.
- The interface is implicit and should not be made explicit.
- Contained procedures are often said to be *nested*.
- Only one level of nesting is permitted.

EXAMPLE

```
program myprog
  implicit none
  real    :: x,y,z
  x=5.; y=10.
  call mysub(z)
contains
  subroutine mysub(w)
    real, intent(inout) :: w
    w=x+y
  end subroutine mysub
end program myprog
```

COMMON

- COMMON is a deprecated feature that is frequently seen in older code. It is a means of providing global variables. Syntax:

```
common /comname/ var1, var2, var3
```

The variables in the common list will be available to *any* program unit that includes the above line. Variables in common between two program units should not be passed as subroutine parameters.

- Newer code should use modules rather than common.

PITFALLS WITH COMMON

- The `common` statement *must* be in every program unit that will share the variables, and it *must* be identical in each one. It is highly recommended that any code using common put each one into a separate file and use the include statement to merge the files. `INCLUDE` is a standard Fortran statement, not a preprocessor statement; its syntax is

```
include 'file.h'
```

where `file.h` can be any name (Fortran does not have a rule about file extensions for included files).

- `COMMON` is a frequent source of memory errors.
- `COMMON` makes interface control difficult to impossible.

COMPILERS, LINKERS, AND MAKE

BUILDING AN EXECUTABLE

- The compiler first produces an *object file* for each *source file*. In Unix these end in `.o`
- Object files are binary (machine language) but cannot be executed. They must be linked into an executable.
- If not told otherwise a compiler will attempt to compile and link the source file(s) it is instructed to compile.
- For Unix compilers the `-c` option suppresses linking. The compiler must then be run again to build the executable from the object files.
- The option `-o` is used to name the binary something other than `a.out`

LINKERS AND LIBRARIES

- When the executable is created any external libraries must also be linked.
- The compiler will search a standard path for libraries. On Unix this is typically `/usr/lib`, `/usr/lib64`, `/usr/local/lib`, `/lib`
- If you have others you must give the compiler the path. `-L` followed by a path works, then the libraries must be named `libfoo.a` or `libfoo.so` and it is referenced `-lfoo`
- Example:

```
gfortran -o mycode -L/usr/lib64/foolib mymain.o mysub.o -lfoo
```


MAKE

- `make` is a tool to manage builds, especially with multiple files.
- It has a rigid and peculiar syntax.
- It will look for a `makefile` first, followed by `Makefile` (on case-sensitive systems).
- The `makefile` defines one or more *targets*. The target is the product of one or more *rules*.
- The target is defined with a colon following its name. If there are *dependencies* those follow the colon.
- Dependencies are other files that are required to create the current target.

TARGETS AND RULES

- Example:

```
myexec: main.o module.o
```

```
<tab>gfortran -o myexec main.o module.o
```

- The tab is *required* in the rule. Don't ask why.
- Macros (automatic targets) for rules:
- `$@` the file name of the current target
- `$<` the name of the first prerequisite

VARIABLES AND COMMENTS

- We can define variables in makefiles
- `F90=gfortran`
- `CXX=g++`
- We then refer to them as `$(F90)`, `$(CXX)`, etc.
- Common variables: `F90`, `CC`, `CXX`, `FFLAGS`, `F90FLAGS`, `CFLAGS`, `CXXFLAGS`, `CPPFLAGS` (for the preprocessor), `LDFLAGS`

SUFFIX RULES

- If all .f90 (or .cc or whatever) files are to be compiled the same way, we can write a *suffix rule* to handle them.
- It uses a *phony target* called .SUFFIXES.

```
.SUFFIXES: .f90 .o
```

```
$(F90) -c $(F90FLAGS) -c $<
```

PATTERN RULES

- An extension by Gnu make (`gmake`), but nearly every `make` is `gmake` now.
- Similar to suffix rules.
- Useful for Fortran 90+:

```
% .mod: % .o
```

- Pattern for creating the `.o`:

```
% .o: % .f90
```

```
$ (F90) $ (F90FLAGS) -c $<
```

EXAMPLE

```
PROG = bmidata
SRCS = bmi_calculator.f90 bmi_data.f90 csv_file.f90 prec.f90 stats.f90
OBJS = bmi_calculator.o bmi_data.o csv_file.o prec.o stats.o
LIBS =

F90 = gfortran
#F90FLAGS=-O
F90FLAGS = -g -C
LDFLAGS =
all: $(PROG)
$(PROG): $(OBJS)
    $(F90) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

.PHONY: clean
clean:
    rm -f $(PROG) $(OBJS) *.mod

.SUFFIXES: $(SUFFIXES) .f90 .F90 .f95
.f90.o .f95.o .F90.o:
    $(F90) $(F90FLAGS) -c $<

stats.o:      prec.o
bmi_calculator.o: csv_file.o prec.o
bmi_data.o:    bmi_calculator.o csv_file.o prec.o stats.o
```

FORTRAN MODULES

WHY USE MODULES

- Modules allow you to organize your code into logically-connected units. It is a form of *object oriented programming*.
- Modules should contain coherent *data+procedures*.
- Modules permit *data hiding*. Variables and subprograms may be kept private from other program units. This prevents another source of error, by reducing the number of variables an outside program can affect or procedures it can call.

FORTRAN MODULES

- Each module has a name that must be unique. A module begins with
`module modname`
- and ends with
`end module`
- Modules are typically placed into separate files. The file name does not need to be the same as the module name, but the module will be referenced by its name and not by the file name.

USING MODULES

- Modules are brought in via the `use` statement

```
use mymodule
```

- All `use` statements must be the first nonexecutable statements after the declaration of the program unit (program, function, subroutine), before `implicit none` and the variable declarations.
- There is no distinct namespace for a Fortran module.

VARIATIONS OF USE

- Only specific routines can be brought in:

```
USE mymod, ONLY : f1, f2, s4
```

- Routines can be renamed:

```
USE mymod, name-here => name-in-  
module
```

```
USE stats_lib, sprod=>prod
```

MODULE VARIABLES.

- `IMPLICIT NONE` at the top applies throughout the module.
- All variables declared or types defined before a `contains` statement are global throughout the module.
- Module symbols (variables and names of routines) can be **private**. You may also explicitly declare them **public** but that is the default.
- The `private` and `public` attributes may be added to the declaration, or they may be specified separately with a list following.
- Private variables are not accessible by program units that use the module. Only units in the same module can access them.
- Example:

```
real, private    :: x, y, z
private          :: r_fun, d_fun
```

SUBPROGRAMS IN MODULES

- Subprograms defined in a module must be within a `CONTAINS` clause.
- Variables declared above the `CONTAINS` are global to all the subprograms.
- The `function` or `subroutine` keywords after `end` are *not* optional, e.g. `end subroutine` is required. The name of the procedure is still optional and some authors do not use it with `end`, in case it is changed later.
- All subprograms in a module have an implicit interface. You should not write an explicit interface for them (and in fact it's illegal to do so).

EXAMPLE

```
module mymod
implicit none
integer      ::  Nmax=100000
contains
    subroutine mysub(a,x)
    real, dimension(:), intent(in)  ::  a
    real                                intent(out) ::  x
    real, dimension(Nmax)              ::  b
        do stuff
    end subroutine mysub
end module mymod
```

EXERCISE

- Type in the module mymod into a file mymod.f90
 - Fortran allows the module and the file to have either the same or a different name, but the name of the module is the name that must appear in the use statement.
- Fill out the subroutine mysub to set b to 11., then set x to the sum of corresponding elements of a and b. Hint: you can use $x=a(:)+b(:\text{size}(a))$ to avoid a loop.
- Write a main program main.f90 that uses mymod, initializes A allocatable, allocates it to 1000, sets its values to $i+3$, then passes it to mysub. Print the value of x that is returned.
- Copy the example Makefile. Make the appropriate changes to the program name, the names of the source files, and the names of the object files. Make the dependency line at the end

```
main.o:main.o mymod.o
```

- Run a `make` project in Geany.

FORTRAN ABSTRACT TYPES

DERIVED TYPES

- In Fortran abstract types are called *derived types*.
- The syntax is extremely simple (ptype stands for a primitive type)

```
type mytype
  <ptype> var1
  <ptype> var2
  <ptype>, dimension(:), allocatable :: var3
  type(anothertype) :: var4
end type mytype
```

FORTRAN DERIVED TYPES

- We nearly always put derived types into modules; the module will define functions that operate on the type.
- The module must not have the same name as the derived type (this is somewhat inconvenient).
- If you need to allocate memory, say for an allocatable array, to create a variable of a given type this *will not* happen automatically. You must write a **constructor** to allocate the memory.
- If you want to delete the variable's memory, you must write a **destructor** to free any allocated memory. This also does not happen automatically.

FORTRAN: DECLARING TYPES AND ACCESSING FIELDS

```
type(mytype) :: thevar  
type(mytype), dimension(100) :: var22  
type(mytype), dimension(:), allocatable :: var11
```

To access the fields of the type use the name of the type, the percent sign as a separator, and the name of the field.

```
thevar%var2  
var11(12)%var1  
var22(1)%var4%varx
```

EXAMPLE

- This type a set of observations for birds denoted by their common name.

```
type bird_data
  character(len=50)      :: species
  integer, dimension(:), allocatable ::
obs
end type bird_data
```

PROCEDURE OVERLOADING

PURPOSE

- Overloaded procedures (functions or subroutines) can accept different types and/or number of arguments and return different results, but use the same name in a calling unit.
- Fortran does not (yet) provide anything like generic typing (templates) so if we want a procedure to work e.g. for both single and double precisions we will write two procedures and overload a generic name for them.

OVERLOADING

- Overloaded procedures must be defined in a module using the `MODULE PROCEDURE` syntax.
- The parameter list must differ in number and/or type for the different procedures to be overloaded.
- `MODULE PROCEDURE` is used within an explicit interface. This is the only time a procedure in a module needs or should have an explicit interface.

EXAMPLE

```
MODULE matrix
  IMPLICIT NONE
  PUBLIC  diagonal
  PRIVATE rtrace, dtrace
  INTERFACE trace
    MODULE PROCEDURE rtrace, dtrace
  END INTERFACE trace
  CONTAINS
    REAL FUNCTION rtrace (x)

      REAL, DIMENSION(:, :) :: x

    END FUNCTION rtrace

    DOUBLE PRECISION FUNCTION dtrace(x)

      DOUBLE PRECISION, DIMENSION(:, :) :: x

    END FUNCTION dtrace

  END MODULE matrix
```


EXERCISE

- Fill in the code for the `rtrace` and `dtrace` functions. It should take a 2-dimensional array argument.
 - The trace of a square matrix is the sum of all the elements on the diagonal. Write a main program that uses the module to compute the trace for a real and a double precision array.
- Add another function to compute the Frobenius norm, which is the square root of the sum of the squares of the absolute values of all the elements in the array.

EXAMPLE – FRACTIONS

- If we wanted to do more careful fraction arithmetic we need a type (later probably a class but type is good enough for now). Note: F2008 constructs for 64-bit integer. Also we are *overloading* two operators.

```
module Rationals
  use iso_fortran_env
  type Fraction
    integer(int64)  :: num, denom
  end type
  module procedure
    interface operator(+)
      function add_frac(a,b)
    end function add_frac
    interface assignment(=)
      ..... Etc
    end interface
  contains
    --what would you want to do here?
  end module Rationals
```

OVERLOADING OPERATORS

- interface operator (+) can be + - * /
 - Must be function with two arguments of the type/class
 - Arguments must be declared `intent(in)`
 - Function must return a new instance of the type
- interface assignment(=)
 - Must be a subroutine with two arguments of the type
 - The first argument must be declared `intent(out)`
 - The second argument must be declared `intent(in)`
 - The realization in the user is then
`firstarg=secondarg`

FRACTIONS

use Rationals

- We can now define

```
type (Fraction) :: A, B, C
```

...

```
A%num=11
```

```
A%denom=12
```

```
B%num=90
```

```
B%denom=20
```

```
C=A+B
```

EXERCISE


- Type in the Rationals module and implement the addition and assignment operators.
- Write a main program that declares three variables of type Fraction, initializes two of them, and sets the third equal to the sum of those two. Print the fraction as `f.num//"/"/f.denom`.
- Extra: implement -, *, and /. Extra extra: look up the Euclid algorithm for greatest common divisor and implement it so that fractions can be reduced.

FORTRAN CLASSES

OOP TERMINOLOGY

- An *instance* of a type or class is a variable of that type/class.

`type(mytype) :: A, B`



A and B are instances of mytype

- A variable that is a member of the type/class is often called an *attribute*.
- A *method* is a subprogram that is a member of a class. Loosely refers to subprograms associated with a non-class type in a module.

TYPES WITH PROCEDURES

- Types containing *type-bound procedures* were introduced in Fortran 2003. They are nearly synonymous with *methods* in other languages.
- Type-bound procedures can be renamed to another name to be used with an instance.
- If a type-bound procedure is public it can be called in the conventional way from a unit that creates a variable of the type.
- If the method is private then it can be accessed only via an instance of the type.

INSTANCE PARAMETERS

- In Fortran the instance variable must be passed explicitly as the first parameter to the method.
- The instance variable must be declared as `class` rather than `type`.
- When we invoke the method we do *not* explicitly pass the instance argument.
- If it does not need to be passed at all (for the equivalent of a *class method* in other languages), the `nopass` attribute can be added.

EXAMPLE

```
module mytype_class
implicit none
type MyType
    integer    :: i,j
    real       :: x,y
contains
    procedure :: init=>init_class
    procedure :: write=>write_class
end type MyType
private init_class

contains

subroutine init_class(self,stuff1,stuff2)
    class(MyType), intent(inout):: self
    real, intent(in)              :: stuff1, stuff2
    self%i=0; self%j=0
    self%x=stuff1; self%y=stuff2
end subroutine init_class
```

EXAMPLE (P. 2)

```
subroutine write_class(self,iunit)
  class(MyType), intent(in) :: self
  integer,          intent(in) :: iunit
      write(*,*) "Integers ",self%i, self%j
      write(*,*) "Reals ", self%x, self%y
end subroutine write_class
end module mytype_class
```

...in caller: `write_class` is not private so the second two calls are equivalent.

```
call myvar%init(x,y)
call write_class(myvar,11)
call myvar%write(12)
```

DATA HIDING

- One of the main purposes of OOP is to prevent outside units from doing anything without “sending a message” to an appropriate instance.
- The previous example violates this principle. We can make everything private, which means that only members of the module can access the symbols. We must then go through an instance of the type/class to invoke the methods.
- Making a type public “exposes” the type name and its type-bound procedures, but not its attributes.
- We will modify the example to accomplish this.

MODIFIED EXAMPLE

```
module mytype_class
  implicit none
  private  !Everything contained is now private
  public :: MyType  !so need to make the type public
  type MyType
    private  !Methods must be declared private
    integer  :: i,j
    real     :: x,y
    contains
      procedure :: init=>init_class
      procedure :: write=>write_class
  end type MyType
  contains
    subroutine init_class(self,stuff1,stuff2)
      class(MyType), intent(inout) :: self
      real,          intent(in)     :: stuff1, stuff2
      self%i=0; self%j=0
      self%x=stuff1; self%y=stuff2
    end subroutine init_class
```

MODIFIED EXAMPLE, P. 2

```
subroutine write_class(self,iunit)
  class(MyType), intent(in) :: self
  integer, intent(in)       :: iunit
  write(*,*) "Integers ",self%i, self%j
  write(*,*) "Reals ", self%x, self%y
end subroutine write_class
end module mytype_class
```

...in caller:

```
call myvar%init(x,y)
call write_class(myvar,11) ! illegal, link error
call myvar%write(12)      ! OK
```

CONSTRUCTORS AND DESTRUCTORS

- A constructor is a subprogram that handles the bookkeeping to initialize an instance of a type. This may entail:
 - Assigning values to attributes
 - Allocating memory for allocatable arrays
 - This *never* happens automatically. If an allocatable is a member of a type, a constructor must be written.
- A destructor is a subprogram that releases memory for a type. This may be required if you allocate in a constructor.

CONSTRUCTORS

- Fortran has no special syntax for a constructor or destructor. Programmers can define an `init` function or equivalent, then declare it `private` to be sure it can be accessed only through a type instance. Destructors can be similarly written to deallocate arrays.

EXERCISE

- Write a class Atom that contains the following attributes:
 - Element symbol
 - Element name
 - Atomic mass
 - Atomic number
- The method should be
 - Compute and return the number of neutrons from the mass and number ($n = \text{mass} - \text{number}$)
- Also write a routine to initialize an instance of the class (a constructor).

INHERITANCE

- Inheritance is not restricted to classes in Fortran.

```
type Parenttype
    integer :: my_id
    real     :: my_value
end type Parenttype
type Childtype extends (Parenttype)
    integer :: my_int
end type Childtype
```

ATTRIBUTE INHERITANCE

- The child type inherits all the attributes of its parent.

```
type (ChildType) :: billy
```

`billy%my_id` is valid, and is equivalent to

```
billy%ParentType%my_id
```

- But `billy%my_int` does not refer back to the parent, since that variable occurs only in the extension.

RECOMMENDATIONS FOR OLDER CODE

UPDATING OLDER CODE

- If your dissertation depends on it (and you are allowed to do it) it's worth the time unless the code is more than 50,000 to 100,000 lines or so.
- Step 1: Replace all COMMON blocks with modules. Initially these modules only need to declare the variables.
- Step 2: Reorganize subprograms into modules. This gives you a free interface, and checks agreement of type and number of arguments.

UPDATING (CONTINUED)

- Step 3: Change variable names to something more meaningful as you can.
- Step 4: Globals are poison, and a major source of bugs (I found a bug in a model that was due to something being global—it had a "memory" when it should not have had one, but since they'd never run the model for different conditions in the same run, they had never noticed it.) Move variables out of the "common" modules into parameter lists.
- Step 5: Introduce types/classes as appropriate.

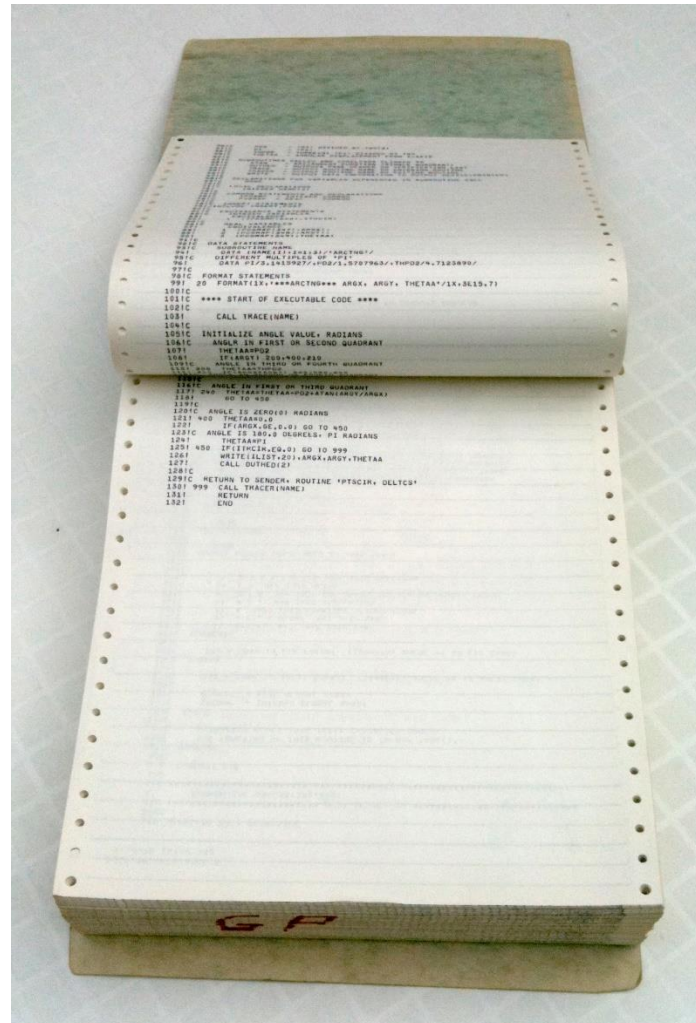
TESTING

- You will definitely need a set of regression tests for this job.
- You may find the original code was not as well tested as you expected (or hoped).

PROGRAMMING STYLE

- Style makes a big difference, especially in compiled languages that tend to be more verbose than interpreted languages.
- Indentation is very important
- Variable names are important – choose variable names that communicate some meaning to the reader. Variable names are no longer limited to 8 characters
- Insert blank lines between logical chunks of code
 - We do not use green-and-white striped fanfold paper anymore to separate lines visually. You have to do it with blank lines and indentation

THE OLD WAY (1978)



(Credit: A Reinhold, Wikipedia)

BEFORE

```
subroutine biogeoclimate
  include "common.txt"
  real tx(365),tn(365),ta(365),p(365)
  real t1(12),t2(12),p1(12)
  real littc1,littc2,littn1,littn2
  real prcp_n
  data prcp_n/0.00002/
c   prcp_n: Nitrogen content for unit rianfall (tN/ha)
c
  rain_1=0.0
  rain_n=0.0
  do i=1,12
    tmp1=ynormal(0.0,1.0)
    tmp1=max(-1.0,min(tmp1,1.0))
    t1(i)=tmin(i)+tmp1*tminv(i)
    t2(i)=tmax(i)+tmp1*tmaxv(i)
    tmp2=ynormal(0.0,1.0)
    tmp2=max(-0.5,min(tmp2,0.5))
c   forest cover can increase rainfall by maximum 15%
    p1(i)=max(prec(i)+tmp2*precv(i),0.0)
c   *(1.0+lai*0.02)
    rain_1=rain_1+p1(i)
c   write(*,*) tmp1, tmp2, tmin(i),tmax(i),prec(i)
  rain_n=rain_n+p1(i)*prcp_n
  end do
  call cov365(t1,tn)
  call cov365(t2,tx)
  call cov365a(p1,p)
  do i=1,365
    ta(i)=0.5*(tn(i)+tx(i))
  end do
c
c   Daily cycles of C, N, H2O
c
  rrr=0.0
  ypet=0.0
  yaet=0.0
  avail_n=0.0
  degd=0.0
  growdays=0.0
  drydays=0.0
  drydays1=0.0
  flooddays=0.0
c
  aoc0=aoc0+go_ao_c
  aon0=aon0+go_ao_n
end
```

AFTER

```

real, parameter      :: growth_thresh=0.05
real                 :: growth_min=0.01

contains
subroutine BioGeoClimate(site,year)
integer, intent(in)  :: year
type(SiteData), intent(inout) :: site

integer              :: gcm_year
integer              :: num_species

real, dimension(NTEMPS) :: tmin, tmax, prcp
real, dimension(NTEMPS) :: tmptmin, tmptmax, tmpprec
real, dimension(days_per_year) :: daytemp, daytemp_min, daytemp_max
real, dimension(days_per_year) :: daynums, dayprecip

real :: litter_c_lev1, litter_c_lev2
real :: litter_n_lev1, litter_n_lev2
real :: rain, rain_n, freeze
real :: temp_f, prcp_f
real :: total_rsp, avail_n, n_avail, C_resp, pet, aet
real :: growdays, drydays_upper, drydays_base
real :: floddays, degday
real :: outwater
real :: exrad, daylength, exradmx
real :: pot_ev_day
real :: act_ev_day
real :: laiw0_ScaledByMax, laiw0_ScaledByMin
real :: aow0_ScaledByMax, aow0_ScaledByMin
real :: sbw0_ScaledByMax, sbw0_ScaledByMin
real :: saw0_ScaledByFC, saw0_ScaledByWP
real :: yxd3 !used but never set
! used to temporarily hold accumulated climate variables
real :: tmpstep1, tmpstep2
real :: tmp
integer :: i, j, k, m

real, parameter :: min_grow_temp =5.0
real, parameter :: max_dry_parm  =1.0001
real, parameter :: min_flood_parm=0.9999

save
num_species=size(site%species)

rain  =0.0
rain_n=0.0

! The user is expected to input decr_by values as positive.
if ( linear_cc ) then
  if ( year_ge. begin_change_year .AND. year .le. &
    (begin_change_year + duration_of_change)) then
    accumulated_tmin = accumulated_tmin + tmin_change
    accumulated_tmax = accumulated_tmax + tmax_change
    do m=1,12
      tmpstep1 = site%precip(m) + accumulated_precip(m)
      tmpstep2 = tmpstep1 * precip_change
      accumulated_precip(m) = accumulated_precip(m) + tmpstep2
    end do
  endif
else if ( use_gcm ) then
  gcm_year=start_gcm+year-begin_change_year
  if ( gcm_year_ge. start_gcm .and. gcm_year .le. end_gcm ) then
    call read_gcm_climate(site%site_id,gcm_year,start_gcm,tmin,tmax,prcp)
    site%tmin=tmin
    site%tmax=tmax
    site%precip=prcp*mm_to_cm
  endif
endif
end subroutine BioGeoClimate
<module continues>

```

CHANGES

- Converted to modules
- Eliminated COMMON, passing all variables (this not only controls the interface, but eventually eliminated a bug)
- Renamed most variables to something more descriptive
- Added lots of whitespace for visual separation
- Aligned typographically (better in source than will convert to PowerPoint)