Introduction to Fortran 90

Alexander B. Pacheco

Research Computing
Lehigh University

Outline

Introduction

What is Fortran?

- Fortran is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing.
- Originally developed by IBM for scientific and engineering applications.
- The name Fortran is derived from The IBM Mathematical **For**mula **Tran**slating System.
- It was one of the first widely used "high-level" languages, as well as the first programming language to be standardized.
- It is still the premier language for scientific and engineering computing applications.

Many Flavors of Fortran

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

The correct spelling of Fortran for 1992 ANSI standard and later (sometimes called Modern Fortran) is "Fortran". Older standards are spelled as "FORTRAN".

Why Learn Fortran?

LEHIGH UNIVERSITY

- Fortran was designed by, and for, people who wanted raw number crunching speed.
- There's a great deal of legacy code and numerical libraries written in Fortran,
- attempts to rewrite that code in a more "stylish" language result in programs that just don't run as fast.
- Fortran is the primary language for some of the most intensive supercomputing tasks, such as
  - astronomy,
  - weather and climate modeling,
  - numerical linear algebra and libraries,
  - computational engineering (fluid dynamics),
  - computational science (chemistry, biology, physics),
  - computational economics, etc.
- How many of you are handed down Fortran code that you are expected to further develop?

LEHIGH UNIVERSITY

SAXPY Code

```
C234567890123456789012345678901234567890123456789012345678901234567890
      program test
      integer n
      parameter(n=100)
      real alpha, x(n), y(n)

      alpha = 2.0
      do 10 i = 1,n
         x(i) = 1.0
         y(i) = 2.0
 10   continue

      call saxpy(n,alpha,x,y)

      return
      end

      subroutine saxpy(n, alpha, x, y)
      integer n
      real alpha, x(*), y(*)
c
c Saxpy: Compute y := alpha*x + y,
c where x and y are vectors of length n (at least).
c
      do 20 i = 1, n
         y(i) = alpha*x(i) + y(i)
 20   continue

      return
      end
```

Why Learn Modern Fortran?

LEHIGH UNIVERSITY

- Free-format source code with a maximum of 132 characters per line,
- Variable names can consists of up to 31 alphanumeric characters (a-z,0-9) and underscores (_),
- Dynamic memory allocation and Ability to operate on arrays (or array sections) as a whole,
- generic names for procedures, optional arguments, calls with keywords, and many other procedure call options,
- Recursive procedures and Operator overloading,
- Structured data or derived types,
- Object Oriented Programming.
- See `http://en.wikipedia.org/wiki/Fortran#Obsolescence_and_deletions` for obsolete and deleted FORTRAN 77 features in newer standards.

LEHIGH UNIVERSITY

SAXPY Code

```
program test

  implicit none
  integer, parameter :: n = 100
  real :: alpha, x(n), y(n)

  alpha = 2.0
  x = 1.0
  y = 2.0

  call saxpy(n,alpha,x,y)

end program test

subroutine saxpy(n, alpha, x, y)
  implicit none
  integer ::  n
  real :: alpha, x(*), y(*)
!
! Saxpy: Compute y := alpha*x + y,
! where x and y are vectors of length n (at least).
!
  y(1:n) = alpha*x(1:n) + y(1:n)

end subroutine saxpy
```

Major Differences with C

- **No standard libraries:** No specific libraries have to be loaded explicitly for I/O and math.
- **Implicit type declaration:** In Fortran, variables of type real and integer may be declared implicitly, based on their first letter. *This behaviour is not recommended in Modern Fortran*.
- **Arrays vs Pointers:** Multi-dimension arrays are supported (arrays in C are one-dimensional) and therefore no vector or array of pointers to rows of a matrices have to be constructed.
- **Call by reference:** Parameters in function and subroutine calls are all passed by reference. When a variable from the parameter list is manipulated, the data stored at that address is changed, not the address itself. Therefore there is no reason for referencing and de-referencing of addresses (as commonly seen in C).

Basics

Fortran Source Code

- Fortran source code is in ASCII text and can be written in any plain-text editor such as vi, emacs, etc.
- For readability and visualization use a text editor capable of syntax highlighting and source code indentation.
- Fortran source code is case insensitive i.e. PROGRAM is the same as Program.
- Using mixed case for statements and variables is not considered a good programming practice. Be considerate to your collaborators who will be modifying the code.
- Some Programmers use uppercase letters for Fortran keywords with rest of the code in lowercase while others (like me) only use lower case letters.
- Use whatever convention you are comfortable with and be consistent throughout.

Fortran Source Code

- The general structure of a Fortran program is as follows

```
PROGRAM name
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END PROGRAM name
```

1. A Fortran program starts with the keyword `PROGRAM` followed by program name,
2. This is followed by the `IMPLICIT NONE` statement (avoid use of implicit type declaration in Fortran 90),
3. Followed by specification statements for various type declarations,
4. Followed by the actual execution statements for the program,
5. Any optional subprogram, and lastly
6. The `END PROGRAM` statement

Fortran Source Code

- A Fortran program consists of one or more program units.
  - `PROGRAM`
  - `SUBROUTINE`
  - `FUNCTION`
  - `MODULE`
- The unit containing the `PROGRAM` attribute is often called the *main program* or *main*.
- The main program should begin with the `PROGRAM` keyword. This is however not required, but it's use if highly recommended.
- A Fortran program should contain only one main program i.e. one `PROGRAM` keyword and can contain one or more subprogram units such as `SUBROUTINE`, `FUNCTION` and `MODULE`.
- Every program unit, must end with a `END` keyword.

Simple I/O

- In Fortran, the `print` command provides the most simple form of writing to standard output while,
- the `read` command provides the most simple form of reading input from standard input
- `print *, <var1> [, <var2> [, ... ]]`
- `read *, <var1> [, <var2> [, ... ]]`
- The $*$ indicates that the format of data read/written is unformatted.
- variables to be read or written should be separated by a comma (,).

Your first code in Fortran

- Open a text editor and create a file helloworld.f90 containing the following lines

```
program hello
  print *, 'Hello World!'
end program hello
```

- The standard extension for Fortran source files is .f90, i.e., the source files are named <name>.f90.
- The .f extension implies fixed format source or FORTRAN 77 code.

Compiling Fortran Code

- To execute a Fortran program, you need to compile it to obtain an executable.
- Almost all *NIX system come with GCC compiler installed. You might need to install the Fortran (gfortran) compiler if its not present.
- Command to compile a fortran program

```
<compiler> [flags] [-o executable] <source code>
```

- The [...] is optional. If you do not specify an executable, then the default executable is `a.out`

```
altair:Exercise apacheco$ gfortran helloworld.f90
altair:Exercise apacheco$ ./a.out
 Hello World!
```

- Other compilers available on our clusters are Intel (ifort) and NVIDIA HPC SDK (nvfortran) compilers.

```
ifort -o helloworld helloworld.f90; ./helloworld
```

Comments

- To improve readability of the code, comments should be used liberally.
- A comment is identified by an exclamation mark or bang (!), except in a character string.
- All characters after ! upto the end of line is a comment.
- Comments can be inline and should not have any Fortran statements following it

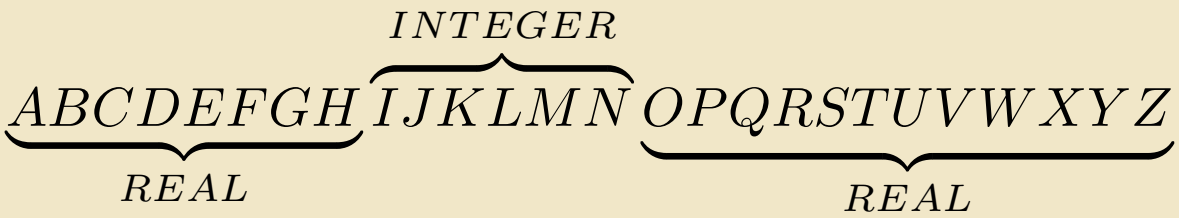
```
program hello
! A simple Hello World code
  print *, 'Hello World!' ! Print Hello World to screen

! This is an incorrect comment if you want Hello World to print to screen ! print *, 'Hello
      World!'
end program hello
```

Fortran Data Types

- Fortran provides five intrinsic data types
  - `INTEGER`: exact whole numbers
  - `REAL`: real, fractional numbers
  - `COMPLEX`: complex, fractional numbers
  - `LOGICAL`: boolean values
  - `CHARACTER`: strings
- and allows users to define additional types.
- The `REAL` type is a single-precision floating-point number.
- The `COMPLEX` type consists of two reals (most compilers also provide a `DOUBLE COMPLEX` type).

Explicit and Implicit Typing

- For historical reasons, Fortran is capable of implicit typing of variables.



- You might come across old FORTRAN program containing `IMPLICIT REAL*8(a-h,o-z)` or `IMPLICIT DOUBLE PRECISION (a-h,o-z)`.
- It is highly recommended to explicitly declare all variable and avoid implict typing using the statement `IMPLICIT NONE`.
- The `IMPLICIT` statement must precede all variable declarations.

Variables

- Variables are the fundamental building blocks of any program.
- A variable is nothing but a name given to a storage area that our programs can manipulate.
- Each variable should have a specific type,
  - which determines the size and layout of the variable's memory;
  - the range of values that can be stored within that memory; and
  - the set of operations that can be applied to the variable.
- A variable name may consist of up to 31 alphanumeric characters and underscores, of which the first character must be a letter.
- Names must begin with a letter and should not contain a space.
- Allowed names: a, compute_force, qed123
- Invalid names: 1a, a thing, $sign

Type	Description
Integer	It can hold only integer values.
Real	It stores the floating point numbers.
Complex	It is used for storing complex numbers.
Logical	It stores logical Boolean values.
Character	It stores characters or strings.

Variable Declarations

- Variables must be declared before they can be used.
- In Fortran, variable declarations must precede all executable statements.
- To declare a variable, preface its name by its type.

```
TYPE Variable
```

- A double colon may follow the type.

```
TYPE[, attributes] :: Variable
```

- This is the new form and is recommended for all declarations. If attributes need to be added to the type, the double colon format must be used.
- A variable can be assigned a value at its declaration.

Variable Declarations

- **Numeric Variables:**

```
INTEGER :: i, j = 2
REAL    :: a, b = 4.d0
COMPLEX :: x, y
```

- In the above examples, the value of j and b are set at compile time and can be changed later.
- If you want the assigned value to be constant that cannot change subsequently, add the attribute `PARAMETER`

```
INTEGER, PARAMETER :: j = 2
REAL, PARAMETER    :: pi = 3.14159265
COMPLEX, PARAMETER :: ci = (0.d0,1.d0)
```

- **Logical:** Logical variables are declared with the `LOGICAL` keyword

```
LOGICAL  :: l, flag=.true.
```

Variable Declarations

- **Character:** Character variables are declared with the `CHARACTER` type; the length is supplied via the keyword `LEN`.
- The length is the maximum number of characters (including space) that will be stored in the character variable.
- If the `LEN` keyword is not specified, then by default `LEN=1` and only the first character is saved in memory.

```
CHARACTER          :: ans = 'yes' ! stored as y not yes
CHARACTER(LEN=10) :: a
```

- FORTRAN programmers: avoid the use of `CHARACTER*10` notation.

KIND Parameter

- In FORTRAN, types could be specified with the number of bytes to be used for storing the value:
  - `real*4` - uses 4 bytes, roughly $\pm 10^{-38}$ to $\pm 10^{38}$.
  - `real*8` - uses 8 bytes, roughly $\pm 10^{-308}$ to $\pm 10^{308}$.
  - `complex*16` - uses 16 bytes, which is two `real*8` numbers.
- Fortran 90 introduced `kind` parameters to parameterize the selection of different possible machine representations for each intrinsic data types.
- The `kind` parameter is an integer which is processor dependent.
- There are only 2(3) kinds of reals: 4-byte, 8-byte (and 16-byte), respectively known as single, double (and quadruple) precision.
- The corresponding `kind` numbers are 4, 8 and 16 (most compilers)

KIND Parameter

KIND	Size (Bytes)	Data Type
1	1	integer, logical, character (default)
2	2	integer, logical
4[a]	4	integer, real, logical, complex
8	8	integer, real, logical, complex
16	16	real, complex

[a] default for all data types except character

- You might come across FORTRAN codes with variable declarations using `integer*4`, `real*8` and `complex*16` corresponding to `kind=4` (integer) and `kind=8` (real and complex).
- The value of the `kind` parameter is usually not the number of decimal digits of precision or range; on many systems, it is the number of bytes used to represent the value.
- The intrinsic functions `selected_int_kind` and `selected_real_kind` may be used to select an appropriate `kind` for a variable or named constant.

KIND Parameter

- `selected_int_kind(R)` returns the kind value of the smallest integer type that can represent all values ranging from -10^R (exclusive) to 10^R (exclusive)
- `selected_real_kind(P,R)` returns the kind value of a real data type with decimal precision of at least P digits, exponent range of at least R. At least one of P and R must be specified, default R is 308.

```
program kind_function

  implicit none
  integer,parameter :: dp = selected_real_kind(15)
  integer,parameter :: ip = selected_int_kind(15)
  integer(kind=4) :: i
  integer(kind=8) :: j
  integer(ip) :: k
  real(kind=4) :: a
  real(kind=8) :: b
  real(dp) :: c

  print '(a,i2,a,i4)', 'Kind of i = ',kind(i), '  with range =', range(i)
  print '(a,i2,a,i4)', 'Kind of j = ',kind(j), '  with range =', range(j)
  print '(a,i2,a,i4)', 'Kind of k = ',kind(k), '  with range =', range(k)
  print '(a,i2,a,i2,a,i4)', 'Kind of real a = ',kind(a),&
```

KIND Parameter

```
      '  with precision = ', precision(a),&
      '  and range =', range(a)
  print '(a,i2,a,i2,a,i4)', 'Kind of real b = ',kind(b),&
      '  with precision = ', precision(b),&
      '  and range =', range(b)
  print '(a,i2,a,i2,a,i4)', 'Kind of real c = ',kind(c),&
      '  with precision = ', precision(c),&
      '  and range =', range(c)
 print *, huge(i),kind(i)
 print *, huge(j),kind(j)
 print *, huge(k),kind(k)

end program kind_function
```

```
[apacheco@qb4 Exercise] ./kindfns
Kind of i =  4  with range =   9
Kind of j =  8  with range =  18
Kind of k =  8  with range =  18
Kind of real a =  4  with precision =  6  and range =  37
Kind of real b =  8  with precision = 15  and range = 307
Kind of real c =  8  with precision = 15  and range = 307
```

Operators

Fortran defines a number of operations on each data type.

Arithmetic Operators

+	: addition
-	: subtraction
*	: multiplication
/	: division
**	: exponentiation

Relational Operators (FORTRAN versions)

==	: equal to (.eq.)
/=	: not equal to (.ne.)
<	: less than (.lt.)
<=	: less than or equal to (.le.)
>	: greater than (.gt.)
>=	: greater than or equal to (.ge.)

Logical Expressions

.AND. intersection

.OR. union

.NOT. negation

.EQV. logical equivalence

.NEQV. exclusive or

Character Operators

// : concatenation

Operator Precedence

LEHIGH UNIVERSITY

Operator	Precedence	Example
expression in ()	Highest	(a+b)
user-defined monadic	-	.inverse.a
**	-	10**4
* or /	-	10*20
monadic + or -	-	-5
dyadic + or -	-	1+5
//	-	str1//str2
relational operators	-	a > b
.not.	-	.not.allocated(a)
.and.	-	a.and.b
.or.	-	a.or.b
.eqv. or .neqv.	-	a.eqv.b
user defined dyadic	Lowest	x.dot.y

Expressions

- An expression is a combination of one or more operands, zero or more operators, and zero or more pairs of parentheses.
- There are three kinds of expressions:
  - An arithmetic expression evaluates to a single arithmetic value.
  - A character expression evaluates to a single value of type character.
  - A logical or relational expression evaluates to a single logical value.
- Examples:

```
x + 1.0
97.4d0
sin(y)
x*aimag(cos(z+w))
a .and. b
'AB' // 'wxy'
```

Statements

- A statement is a complete instruction.
- Statements may be classified into two types: executable and non-executable.
- Non-executable statements are those that the compiler uses to determine various fixed parameters such as module use statements, variable declarations, function interfaces, and data loaded at compile time.
- Executable statements are those which are executed at runtime.
- A statements is normally terminated by the end-of-line marker.
- If a statement is too long, it may be continued by the ending the line with an ampersand (&).
- Max number of characters (including spaces) in a line is 132 though it's standard practice to have a line with up to 80 characters. This makes it easier for file editors to display code or print code on paper for reading.
- Multiple statements can be written on the same line provided the statements are separated by a semicolon.

Statements

- Examples:

```
force = 0d0 ; pener = 0d0
do k = 1, 3
   r(k) = coord(i,k) - coord(j,k)
```

- Assignment statements assign an expression to a quantity using the equals sign (=)
- The left hand side of the assignment statement must contain a single variable.
- $x + 1.0 = y$ is not a valid assignment statement.

Intrinsic Functions

- Fortran provide a large set of intrinsic functions to implement a wide range of mathematical operations.
- In FORTRAN code, you may come across intrinsic functions which are prefixed with `i` for integer variables, `d` for double precision, `c` for complex single precision and `cd` for complex double precision variables.
- In Modern Fortran, these functions are overloaded, i.e. they can carry out different operations depending on the data type.
- For example: the `abs` function equates to $\sqrt{a^2}$ for integer and real numbers and $\sqrt{\Re^2 + \Im^2}$ for complex numbers.

Arithmetic Functions

Function	Action	Example
INT	conversion to integer	J=INT(X)
REAL	conversion to real	X=REAL(J)
	return real part of complex number	X=REAL(Z)
DBLE[a]	convert to double precision	X=DBLE(J)
CMPLX	conversion to complex	A=CMPLX(X[,Y])
AIMAG	return imaginary part of complex number	Y=AIMAG(Z)
ABS	absolute value	Y=ABS(X)
MOD	remainder when I divided by J	K=MOD(I,J)
CEILING	smallest integer $\geq$ to argument	I=CEILING(a)
FLOOR	largest integer $\leq$ to argument	I=FLOOR(a)
MAX	maximum of list of arguments	A=MAX(C,D)
MIN	minimum of list of arguments	A=MIN(C,D)
SQRT	square root	Y=SQRT(X)
EXP	exponentiation	Y=EXP(X)
LOG	natural logarithm	Y=LOG(X)
LOG10	logarithm to base 10	Y=LOG10(X)

[a] use real(x,kind=8) instead

Trignometric Functions

Function	Action	Example
SIN	sine	X=SIN(Y)
COS	cosine	X=COS(Y)
TAN	tangent	X=TAN(Y)
ASIN	arcsine	X=ASIN(Y)
ACOS	arccosine	X=ACOS(Y)
ATAN	arctangent	X=ATAN(Y)
ATAN2	arctangent(a/b)	X=ATAN2(A,B)
SINH	hyperbolic sine	X=SINH(Y)
COSH	hyperbolic cosine	X=COSH(Y)
TANH	hyperbolic tangent	X=TANH(Y)

hyperbolic functions are not defined for complex argument

Character Functions

Function	Description
len(c)	length
len_trim(c)	length of c if it were trimmed
lge(s1,s2)	returns .true. if s1 follows or is equal to s2 in lexical order
lgt(s1,s2)	returns .true. if s1 follows s1 in lexical order
lle(s1,s2)	returns .true. if s2 follows or is equal to s1 in lexical order
llt(s1,s2)	returns .true. if s2 follows s1 in lexical order
adjustl(s)	returns string with leading blanks removed and same number of trailing blanks added
adjustr(s)	returns string with trailing blanks removed and same number of leading blanks added
repeat(s,n)	concatenates string s to itself n times
scan(s,c)	returns the integer starting position of string c within string s
trim(c)	trim trailing blanks from c

Simple Temperature Conversion Problem

- A simple program that
  1. Converts temperature from celsius to fahrenheit
  2. Converts temperature from fahrenheit to celsius

```
program temp

  implicit none
  real :: tempC, tempF

  ! Convert 10C to fahrenheit

  tempF = 9 / 5 * 10 + 32

  ! Convert 40F to celsius

  tempC = 5 / 9 * (40 - 32 )

  print *, '10C = ', tempF, 'F'
  print *, '40F = ', tempC, 'C'

end program temp
```

```
altair:Exercise apacheco$ gfortran simple.f90
altair:Exercise apacheco$ ./a.out
 10C =    42.0000000     F
 40F =    0.00000000     C
```

- So what went wrong? $10C = 50F$ and $40F = 4.4C$

Type Conversion

- In computer programming, operations on variables and constants return a result of the same type.
- In the temperature code, $9/5 = 1$ and $5/9 = 0$. Division between integers is an integer with the fractional part truncated.
- In the case of operations between mixed variable types, the variable with lower rank is promoted to the highest rank type.

Variable 1	Variable 2	Result
Integer	Real	Real
Integer	Complex	Complex
Real	Double Precision	Double Precision
Real	Complex	Complex

Type Conversion

- As a programmer, you need to make sure that the expressions take type conversion into account

```
program temp

  implicit none
  real :: tempC, tempF

  ! Convert 10C to fahrenhiet

  tempF = 9. / 5. * 10 + 32

  ! Convert 40F to celsius

  tempC = 5. / 9. * (40 - 32 )

  print *, '10C = ', tempF, 'F'
  print *, '40F = ', tempC, 'C'

end program temp
```

```
altair:Exercise apacheco$ gfortran temp.f90
altair:Exercise apacheco$ ./a.out
 10C =    50.0000000     F
 40F =    4.44444466     C
```

- The above example is not a good programming practice.
- 10, 40 and 32 should be written as real numbers (10., 40. and 32.) to stay consistent.

Control Constructs

Control Constructs

- A Fortran program is executed sequentially

```
program somename
  variable declarations
  statement 1
  statement 2
  ...
end program somename
```

- Control Constructs change the sequential execution order of the program
  1. Conditionals: `IF`
  2. Loops: `DO`
  3. Switches: `SELECT/CASE`
  4. Branches: `GOTO` (obsolete in Fortran 95/2003, use CASE instead)

If Statement

- The general form of the `if` statement

```
if ( expression )statement
```

- When the `if` statement is executed, the logical expression is evaluated.
- If the result is true, the statement following the logical expression is executed; otherwise, it is not executed.
- The statement following the logical expression **cannot** be another `if` statement. Use the `if-then-else` construct instead.

```
if (value < 0)value = 0
```

If-then-else Construct

- The `if-then-else` construct permits the selection of one of a number of blocks during execution of a program
- The `if-then` statement is executed by evaluating the logical expression.
- If it is true, the block of statements following it are executed. Execution of this block completes the execution of the entire `if` construct.
- If the logical expression is false, the next matching `else if`, `else` or `end if` statement following the block is executed.

```
if ( expression 1) then
  executable statements
else if ( expression 2 ) then
  executable statements
else if ···

  .
  .
  .
else
  executable statements
end if
```

If-then-else Construct

- Examples:

```
if ( x < 50 ) then
  GRADE = 'F'
else if ( x >= 50 .and. x < 60 ) then
  GRADE = 'D'
else if ( x >= 60 .and. x < 70 ) then
  GRADE = 'C'
else if ( x >= 70 .and. x < 80 ) then
  GRADE = 'B'
else
  GRADE = 'A'
end if
```

If-then-else Construct

- The `else if` and `else` statements and blocks may be omitted.
- If `else` is missing and none of the logical expressions are true, the `if-then-else` construct has no effect.
- The `end if` statement must not be omitted.
- The `if-then-else` construct can be nested and named.

no else if

```
[outer_name:] if ( expression ) then
  executable statements
else
  executable statements
  [inner_name:] if ( expression ) then
    executable statements
  end if [inner_name]
end if [outer_name]
```

no else

```
if ( expression ) then
  executable statements
else if ( expression ) then
  executable statements
else if ( expression ) then
  executable statements
end if
```

Case Construct

- The `case` construct permits selection of one of a number of different block of instructions.
- The value of the expression in the `select case` should be an integer or a character string.

```
[case_name:] select case ( expression )
  case ( selector )
    executable statement
  case ( selector )
    executable statement
  case default
    executable statement
end select [case_name]
```

- The `selector` in each `case` statement is a list of items, where each item is either a single constant or a range of the same type as the expression in the `select case` statement.
- A range is two constants separated by a colon and stands for all the values between and including the two values.
- The `case default` statement and its block are optional.

Case Construct

- The `select case` statement is executed as follows:
  1. Compare the value of expression with the case selector in each case. If a match is found, execute the following block of statements.
  2. If no match is found and a `case default` exists, then execute those block of statements.

Notes

- The values in selector must be unique.
- Use `case default` when possible, since it ensures that there is something to do in case of error or if no match is found.
- `case default` can be anywhere in the `select case` construct. The preferred location is the last location in the `case` list.

Case Construct

- Example for character case selector

```
select case ( traffic_light )
  case ( "red" )
    print *, "Stop"
  case ( "yellow" )
    print *, "Caution"
  case ( "green" )
    print *, "Go"
  case default
    print *, "Illegal value: ", traffic_light
end select
```

- Example for integer case selector

```
select case ( score )
  case ( 50 : 59 )
    GRADE = "D"
  case ( 60 : 69 )
    GRADE = "C"
  case ( 70 : 79 )
    GRADE = "B"
  case ( 80 : )
    GRADE = "A"
  case default
    GRADE = "F"
end select
```

Do Construct

- The looping construct in fortran is the `do` construct.
- The block of statements called the loop body or `do` construct body is executed repeatedly as indicated by loop control.
- A `do` construct may have a construct name on its first statement

```
[do_name:] do loop_control
  execution statements
end do [do_name]
```

- There are two types of loop control:
  1. Counting: a variable takes on a progression of integer values until some limit is reached.
     - *variable = start, end[, stride]*
     - *stride* may be positive or negative integer, default is 1 which can be omitted.
  2. General: a loop control is missing
- Before a `do` loop starts, the expression *start, end* and *stride* are evaluated. These values are not re-evaluated during the execution of the `do` loop.

Do Construct

- *stride* cannot be zero.
- If *stride* is positive, this `do` counts up.
  1. The *variable* is set to *start*
  2. If *variable* is less than or equal to *end*, the block of statements is executed.
  3. Then, *stride* is added to *variable* and the new *variable* is compared to *end*
  4. If the value of *variable* is greater than *end*, the `do` loop completes, else repeat steps 2 and 3
- If *stride* is negative, this `do` counts down.
  1. The *variable* is set to *start*
  2. If *variable* is greater than or equal to *end*, the block of statements is executed.
  3. Then, *stride* is added to *variable* and the new *variable* is compared to *end*
  4. If the value of *variable* is less than *end*, the `do` loop completes, else repeat steps 2 and 3

Do Construct: Nested

- The `exit` statement causes termination of execution of a loop.
- If the keyword `exit` is followed by the name of a do construct, that named loop (and all active loops nested within it) is exited and statements following the named loop is executed.
- The `cycle` statement causes termination of the execution of *one iteration* of a loop.

  The `do` body is terminated, the `do` variable (if present) is updated, and control is transferred back to the beginning of the block of statements that comprise the `do` body.
- If the keyword `cycle` is followed by the name of a construct, all active loops nested within that named loop are exited and control is transferred back to the beginning of the block of statements that comprise the named `do` construct.

Do Construct: Nested

```
program nested_doloop

  implicit none
  integer,parameter :: dp = selected_real_kind(15)
  integer :: i,j
  real(dp) :: x,y,z,pi

  pi = 4d0*atan(1.d0)

  outer: do i = 0,180,45
     inner: do j = 0,180,45
        x = real(i)*pi/180d0
        y = real(j)*pi/180d0
        if ( j == 90 ) cycle inner
        z = sin(x) / cos(y)
        print '(2i6,3f12.6)', i,j,x,y,z
     end do inner
  end do outer
end program nested_doloop
```

```
[apacheco@qb4 Exercise] ./nested
     0     0    0.000000    0.000000    0.000000
     0    45    0.000000    0.785398    0.000000
     0   135    0.000000    2.356194   -0.000000
     0   180    0.000000    3.141593   -0.000000
    45     0    0.785398    0.000000    0.707107
    45    45    0.785398    0.785398    1.000000
    45   135    0.785398    2.356194   -1.000000
    45   180    0.785398    3.141593   -0.707107
    90     0    1.570796    0.000000    1.000000
    90    45    1.570796    0.785398    1.414214
    90   135    1.570796    2.356194   -1.414214
    90   180    1.570796    3.141593   -1.000000
   135     0    2.356194    0.000000    0.707107
   135    45    2.356194    0.785398    1.000000
   135   135    2.356194    2.356194   -1.000000
   135   180    2.356194    3.141593   -0.707107
   180     0    3.141593    0.000000    0.000000
   180    45    3.141593    0.785398    0.000000
   180   135    3.141593    2.356194   -0.000000
   180   180    3.141593    3.141593   -0.000000
```

Do Construct: General

- The General form of a `do` construct is

```
[do_name:] do
  executable statements
end do [do_name]
```

- The `executable statements` will be executed indefinitly.
- To exit the `do` loop, use the `exit` or `cycle` statement.
- The `exit` statement causes termination of execution of a loop.
- The `cycle` statement causes termination of the execution of *one iteration* of a loop.

```
finite: do
  i = i + 1
  inner: if ( i < 10 ) then
    print *, i
    cycle finite
  end if inner
  if ( i > 100 ) exit finite
end do finite
```

Do While Construct

- If a condition is to be tested at the top of a loop, a `do ... while` loop can be used

```
[do_name:] do while ( expression )
  executable statements
end do [do_name]
```

- The loop only executes if the logical expression evaluates to `.true.`

```
finite: do while ( i <= 100 )
  i = i + 1
  inner: if ( i < 10 ) then
    print *, i
  end if inner
end do finite
```

```
finite: do
  i = i + 1
  inner: if ( i < 10 ) then
    print *, i
    cycle finite
  end if inner
  if ( i > 100 ) exit finite
end do finite
```

Input and Output

Input and Output Descriptors

- Input and output are accomplished by operations on files.
- Files are identified by some form of file handle, in Fortran called the **unit number**.
- We have already encountered read and write command such as `print *`, and `read *`,
- Alternative commands for read and write are

```
read(unit,*)
```

```
write(unit,*)
```

- There is no comma after the ')'. FORTRAN allowed statements of the form `write(unit,*)`, which is not supported on some compilers such as IBM XLF. Please avoid this notation in FORTRAN programs.
- The default unit number 5 is associated with the standard input, and
- unit number 6 is assigned to standard output.

Input and Output Descriptors

- You can replace `unit` with $\star$ in which case standard input (5) and output (6) file descriptors are used.
- The second $\star$ in `read/write` or the one in the `print */read *` corresponds to unformatted input/output.
- If I/O is formatted, then $\star$ is replaced with

  `fmt=<format specifier>`

File Operations

- A file may be opened with the statement

  ```
  OPEN([UNIT=]un, FILE=fname [, options])
  ```

- Commonly used options for the open statement are:

  `IOSTAT=ios`: This option returns an integer ios; its value is zero if the statement executed without error, and nonzero if an error occured.

  `ERR=label`: label is the label of a statement in the same program unit. In the event of an error, execution is transferred to this labelled statement.

  `STATUS=istat`: This option indicates the type of file to be opened. Possible values are:

  old : the file specified by the file parameter must exist.
  new : the file will be created and must not exist.
  replace : the file will be created if it does not exist or if it exists, the file will be deleted and created i.e. contents overwritten.
  unknown : the file will be created if it doesn't exist or opened if it exists without further processing.

scratch : file will exist until the termination of the executing program or until a `close` is executed on that unit.

`position=todo`: This options specifies the position where the read/write marker should be placed when opened. Possible values are:

rewind : positions the file at its initial point. Convenient for rereading data from file such as input parameters.

append : positions the file just before the endfile record. Convenient while writing to a file that already exists. If the file is new, then the position is at its initial point.

File Operations

- A file may be closed with the statement

  ```
  CLOSE([UNIT=]un [, options])
  ```

- Commonly used options for the close statement are:

  `IOSTAT=ios`: Same use as `open` statement.

  `ERR=label`: Same use as `open` statement.

  `STATUS=todo`: What actions needs to be performed on the file while closing it. Possible values are

  keep : file will continue to exist after the close statement, default option except for scratch files.

  delete : file will cease to exist after the close statement, default option for scratch files.

- The `WRITE` statement is used to write to a file.
- Syntax for writing a list of variable, `varlist`, to a file associated with unit number `un`

  ```
  WRITE(un, options)varlist
  ```

- The most common options for `WRITE` are:

  `FMT=label` A format statement label specifier.

  You can also specify the exact format to write the data to be discussed in a few slides.

  `IOSTAT=ios` Returns an integer indicating success or failure; zero if statement executed with no erros and nonzero if an error occured.

  `ERR=label` The label is a statement label to which the program should jump if an error occurs.

Reading and Writing Data

- The `READ` statement is used to read from a file.
- Syntax for reading a list of variable, `varlist`, to a file associated with unit number `un`

  ```
  READ(un, options)varlist
  ```

- Options to the `READ` statement are the same as that of the `WRITE` statement with one additional option,

  `END=label` The label is a statement label to which the program should jump if the end of file is detected.

List-Directed I/O

- The simplest method of getting data into and out of a program is list-directed I/O.
- The data is read or written as a stream into or from specified variables either from standard input or output or from a file.
- The unit number associate with standard input is 5 while standard output is 6.
- If data is read/written from/to standard input/output, then
  - the unit number, `un` can also be replaced with $*$,
  - use alternate form for reading and writing i.e. the `read *,` and `print *,` covered in an earlier slide.
  - If data is unformatted i.e. plain ASCII characters, the option to `write` and `read` command is $*$

List-Directed I/O

- Example of list-directed output to standard output or to a file associated with unit number 8

```
print *, a, b, c, arr
write(*,*) a, b, arr
write(6,*) a, b, c, arr
write(8,*) a, b, c, &
  arr
```

- Unlike C/C++, Fortran always writes an end-of-line marker at the end of the list of item for any `print` or `write` statements.
- Printing a long line with many variables may thus require continuations.
- Example of list-directed input from standard output or to a file associated with unit number 8

```
read *, a, b, c, arr
read(*,*) a, b, c, arr
read(5,*) a, b, c, arr
read(8,*) a, b, c, arr
```

List-Directed I/O

- When reading from standard input, the program will wait for a response from the console.
- Unless explicitly told to do so, no prompts to enter data will be printed. Very often programmers use a print statement to let you know that a response is expected.

```
print *, 'Please enter a value for the variable inp'
read *, inp
```

Formatted Input/Output

- List-directed I/O does not always print the results in a particularly readable form.
- For example, a long list of variable printed to a file or console may be broken up into multiple lines.
- In such cases it is desirable to have more control over the format of the data to be read or written.
- Formatted I/O requires that the programmer control the layout of the data.
- The type of data and the number of characters that each element may occupy must be specified.

- A formatted data description must adhere to the generic form,

  `nCw.d`

  where

  - `n` is an integer constant that specifies the number of repititions (default 1 can be omitted),
  - `C` is a letter indicating the type of the data variable to be written or read,
  - `w` is the total number of spaces allocated to this variable, and,
  - `d` is the number of spaces allocated to the fractional part of the variable. Integers are padded with zeros for a total width of `w` provided $d \leq w$.
  - The decimal (.) and `d` designator are not used for integers, characters or logical data types. Note that `d` designator has a different meaning for integers and is usually referred to as `m` to avoid confusion.
- Collectively, these designators are called **edit descriptors**.
- The space occupied by an item of data or variable is called *field*.

Formatted Input/Output

Data Type	Edit Descriptor	Examples	Result
Integer	nIw[.m]	I5.5	`00010`
Real[a] (floating point)	nFw.d	F12.6	`10.123456`
Real (exponential)	Ew.d[en][b]	E15.8	`0.12345678E1`
Real (engineering)	ESw.d[c]	ES12.3	`50.123E-3`
Character	nAw	A12	`Fortran`

[a] For complex variables, use two appropriate real edit descriptors

[b] en is used when you need more than 2 digits in the exponent as in 100. E15.7e4 to represent 2.3×10^{1021}

[c] data is printed in multiples of 1000

- **Control descriptors** alter the input or output by addings blanks, new lines and tabs.

Space	nX	add n spaces
Tabs	tn	tab to position n
	tln	tab left n positions
	trn	tab right n positions
New Line	/	Create a new line record

Format Statements

- Edit descriptors must be used in conjunction with a `PRINT`, `WRITE` or `READ` statement.
- In the simplest form, the format is enclosed in single quotes and parentheses as as argument to the keyword.

```
print '(I5,5F12.6)', i, a, b, c, z ! complex z
write(6,'(2E15.8)') arr1, arr2
read(5,'(2a)') firstname, lastname
```

- If the same format is to be used repeatedly or it is complicated, the `FORMAT` statement can be used.
- The `FORMAT` statement must be labeled and the label is used in the input/output statement to reference it

```
label FORMAT(formlist)
PRINT label, varlist
WRITE(un, label) varlist
READ(un, label) varlist
```

Format Statements

- The FORMAT statements can occur anywhere in the same program unit. Most programmers list all FORMAT statements immediately after the type declarations before any executable statements.

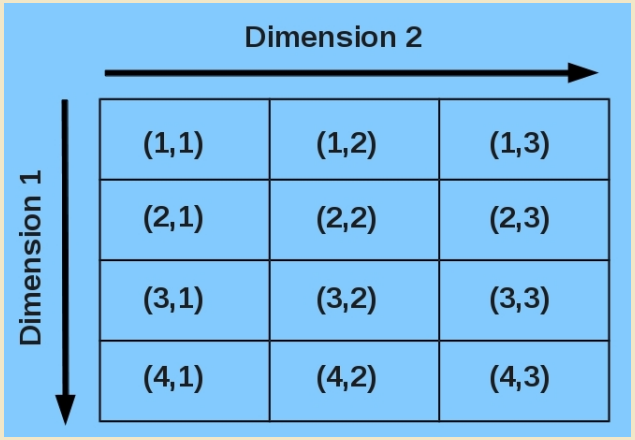
```
10 FORMAT(I5,5F12.6)
20 FORMAT(2E15.8)
100 FORMAT(2a)

print 10, i, a, b, c, z ! complex z
write(6,20) arr1, arr2
read(5,100) firstname, lastname
```

Arrays

- Arrays (or matrices) hold a collection of different values at the same time.
- Individual elements are accessed by subscripting the array.
- A 10 element array is visualized as

1	2	3	· · ·	8	9	10

  while a 4x3 array as



- Each array has a type and each element of the array holds a value of that type.

Array Declarations

- The `dimension` attribute declares arrays.
- Usage: `dimension(lower_bound:upper_bound)`

  Lower bounds of one `(1:)` can be omitted
- Examples:

```
integer, dimension(1:106) :: atomic_number
real, dimension(3,0:5,-10:10) :: values
character(len=3),dimension(12) :: months
```

- Alternative form for array declaration

```
integer :: days_per_week(7), months_per_year(12)
real :: grid(0:100,-100:0,-50:50)
complex :: psi(100,100)
```

- Another alternative form which can be very confusing for readers

```
integer, dimension(7) :: days_per_week, months_per_year(12)
```

Array Terminology

```
real :: a(0:20), b(3,0:5,-10:10)
```

Rank: Number of dimensions.

`a` has rank 1 and `b` has rank 3

Bounds: upper and lower limits of each dimension of the array.

`a` has bounds 0:20 and `b` has bounds 1:3, 0:5 and -10:10

Extent: Number of element in each dimension

`a` has extent 21 and `b` has extents 3,6 and 21

Size: Total number of elements.

`a` has size 21 and `b` has 30

Shape: The shape of an array is its rank and extent

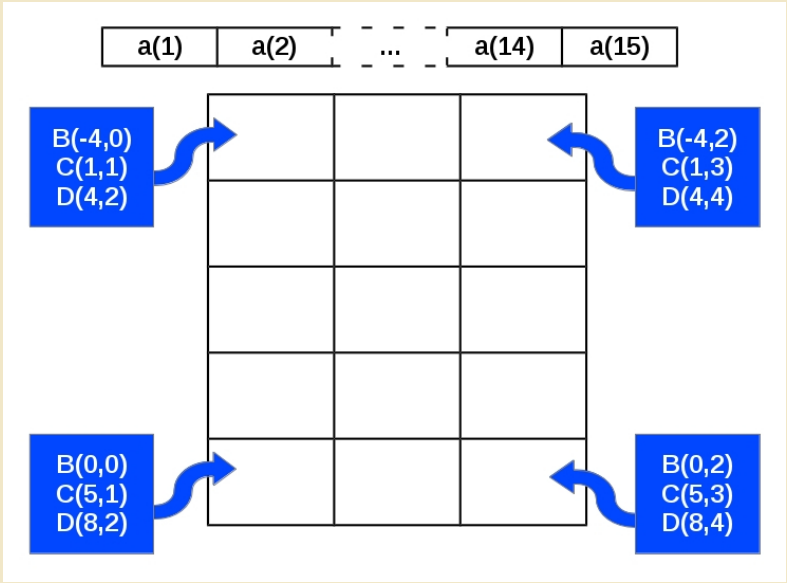
`a` has shape 21 and `b` has shape (3,6,21)

- Arrays are conformable if they share a shape.
- The bounds do not have to be the same

```
c(4:6)= d(1:3)
```

- Define arrays `a`,`b`,`c` and `d` as follows

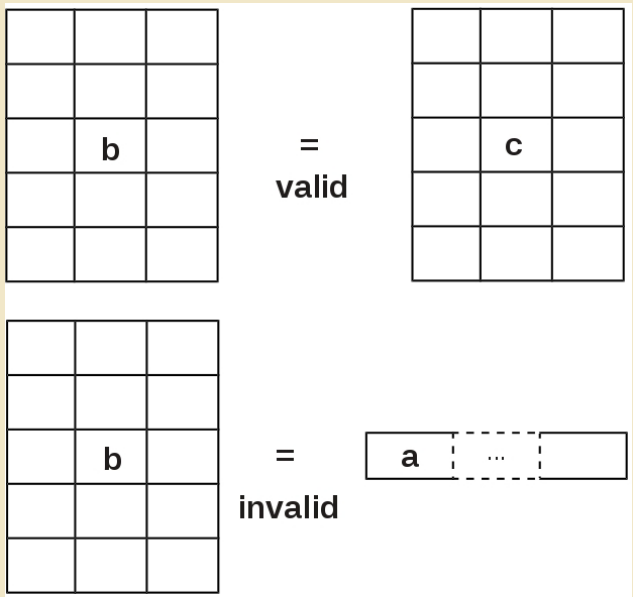
```
real,dimension(15) :: a
real,dimension(-4:0,0:2) :: b
real,dimension(5,3) :: c
real,dimension(4:8,2:4) :: d
```



Array Conformance

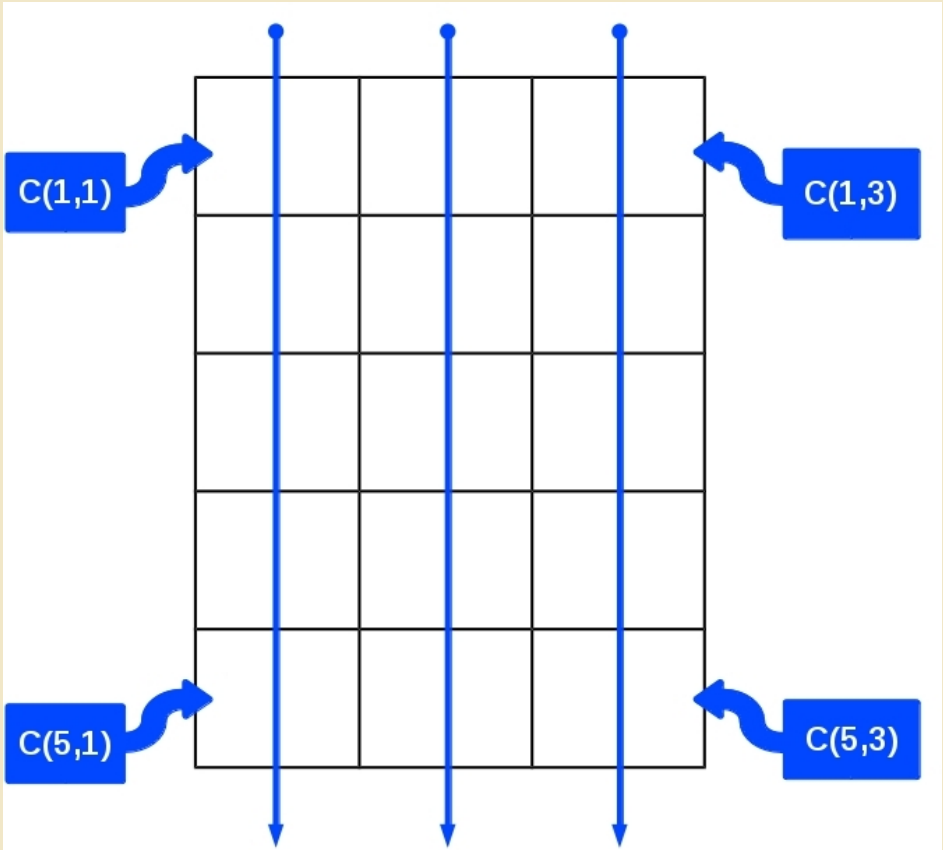
- Array or sub-arrays must conform with all other objects in an expression
  1. a scalar conforms to an array of any shape with the same value for every element

     `c = 1.0` is the same as `c(:,:)= 1.0`
  2. two array references must conform in their shape.



Array Element Ordering

- Fortran is a column major form i.e. elements are added to the columns seqeuntially. This ordering can be changed using the `reshape` intrinsic.



Array Constructors

- Used to give arrays or sections of arrays specific values

```
implicit none
integer :: i
integer, dimension(10) :: ints
character(len=5),dimension(3) :: colors
real, dimension(4) :: height
height = (/5.10, 5.4, 6.3, 4.5 /)
colors = (/'red  ', 'green', 'blue ' /)
ints = (/ 30, (i = 1, 8), 40 /)
```

- constructors and array sections must conform.

  `ints = (/ 30, (i = 1, 10), 40/)` is invalid
- strings should be padded so that character variables have correct length.
- use `reshape` intrinsic for arrays for higher ranks
- `(i = 1, 8)` is an implied `do`.
- You can also specify a stride in the implied `do`.

  `ints = (/ 30, (i = 1, 16, 2), 40/)`
- There should be no space between / and (or)

Array Constructors

- `reshape(source, shape, pad, order)` constructs an array with a specified shape `shape` starting from the elements in a given array `source`.
- If `pad` is not included then the size of `source` has to be at least `product (shape)`.
- If `pad` is included it has to have the same type as `source`.
- If `order` is included, it has to be an `integer` array with the same shape as `shape` and the values must be a permutation of (1,2,3,...,N), where N (max value is 7) is the number of elements in `shape`.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & a & a \\ a & 0 & a \\ a & a & 0 \end{pmatrix}$$

```
rcell = reshape( (/ &
     0.d0, 0.d0, a,    a,   &
     0.d0, a,    0.d0, a,   &
     0.d0, a,    a,    0.d0 &
     /),(/4,3/) )
```

```
rcell = reshape( (/ &
     0.d0, 0.d0, 0.d0 &
     0.d0, a   , a    &
     a,    0.d0, a    &
     a,    a,    0.d0 &
     /),(/4,3/),order=(/2,1/))
```

Array Constructors

- In Fortran, for a multidimensional array, the first dimension has the fastest index while the last dimension has the slowest index i.e. memory locations are continuous for the last dimension.
- The `order` statement allows the programmer to change this order. The last example above sets the memory location order which is consistent to that in C/C++.
- Arrays can be initialized as follows during variable declaration

```
integer, dimension(4) :: imatrix = (/ 2, 4, 6, 8/)
character(len=*),dimension(3) :: colors = (/'red  ', 'green', 'blue '/)}
! All strings must be the same length}
real, dimension(4) :: height = (/5.10, 5.4, 6.3, 4.5/)
integer, dimension(10) :: ints = (/ 30, (i = 1, 8), 40/)
real, dimension(4,3), parameter :: rcell = reshape( (/0.d0, 0.d0, 0.d0, 0.d0,\&
  a, a, a,0.d0, a, a, a, 0.d0 /),(/4,3/),order=(/2,1/))
```

- Arrays can be treated as a single variable when performing operations
  1. set whole array to a constant: `a = 0.0`
  2. can use intrinsic operators between conformable arrays (or sections)

     `b = c * d + b**2`

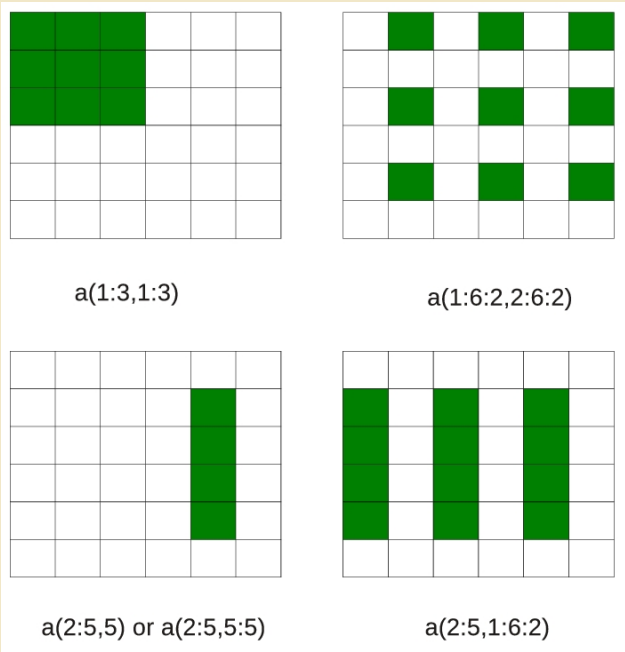
     this is equivalent to

```
b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
b(-3,0) = c(2,1) * d(5,2) + b(-3,0)**2
...
b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
b(-4,1) = c(1,2) * d(4,3) + b(-4,1)**2
...
b(-3,2) = c(4,3) * d(7,4) + b(-3,2)**2
b(-4,2) = c(5,3) * d(8,4) + b(-4,2)**2
```

  3. elemental intrinsic functions can be used: `b = sin(c)+ cos(d)`
  4. All operations/functions are applied element by element

```
real, dimension(6:6):: a
```

- `a(1:3,1:3)= a(1:6:2,2:6:2)` and `a(1:3,1:3)= 1.0` are valid
- `a(2:5,5)= a(2:5,1:6:2)` and `a(2:5,1:6:2)= a(1:6:2,2:6:2)` are not
- `a(2:5,5)` is a 1D section while `a(2:5,1:6:2)` is a 2D section



- The general form for specifying sub-arrays or sections is *[<bound1>]:[<bound2>][:<stride>]*
- The section starts at *<bound1>* and ends at or before *<bound2>*.
- *<stride>* is the increment by which the locations are selected, by default *stride=1*
- *<bound1>*, *<bound2>*, *<stride>* must all be scalar integer expressions.

```
real, dimension(1:20) :: a
integer :: m,n,k
```

a(:)	the whole array
a(3:9)	elements 3 to 9 in increments of 1
a(3:9:1)	as above
a(m:n)	elements m through n
a(m:n:k)	elements m through n in increments of k
a(15:3:-2)	elements 15 through 3 in increments of -2
a(15:3)	zero size array
a(m:)	elements m through 20, default upper bound
a(:n)	elements 1, default lower bound through n
a(::2)	all elements from lower to upper bound in increments of 2
a(m:m)	1 element section
a(m)	array element not a section

are valid sections.

```
real,dimension(4,4):: a
```

- Arrays are printed in the order that they appear in memory

  ```
  print *, a
  ```

  would produce on output

  ```
  a(1,1),a(2,1),a(3,1),a(4,1),a(1,2),a(2,2),···,a(3,4),a(4,4)
  ```

  ```
  read *, a
  ```

  would read from input and assign array elements in the same order as above
- The order of array I/O can be changed using intrinsic functions such as `reshape`, `transpose` or `cshift`.

- Example: consider a 3x3 matrix

1	4	7
2	5	8
3	6	9

- The following print statements

```
print *, 'array element   = ',a(3,3)
print *, 'array section   = ',a(:,2)
print *, 'sub-array       = ',a(:3,:2)
print *, 'whole array     = ',a
print *, 'array transpose = ',transpose(a)
```

- would produce the following output

```
array element   = 9
array section   = 4 5 6
sub-array       = 1 2 3 4 5 6
whole array     = 1 2 3 4 5 6 7 8 9
array transpose = 1 4 7 2 5 8 3 6 9
```

Array Intrinsic Functions

size(x[,n])	The size of x (along the n^{th} dimension, optional)
shape(x)	The shape of x
lbound(x[,n])	The lower bound of x
ubound(x[,n])	The upper bound of x
minval(x)	The minimum of all values of x
maxval(x)	The maximum of all values of x
minloc(x)	The indices of the minimum value of x
maxloc(x)	The indices of the maximum value of x
sum(x[,n])	The sum of all elements of x (along the n^{th} dimension, optional)

$sum(x) = \sum_{i,j,k,\ldots} x_{i,j,k,\ldots}$

Array Intrinsic Functions

LEHIGH UNIVERSITY

product(x[,n]) The product of all elements of x (along the n^{th} dimension, optional)

$prod(x) = \prod_{i,j,k,\ldots} x_{i,j,k,\ldots}$

transpose(x) Transpose of array x: $x_{i,j} \Rightarrow x_{j,i}$

dot_product(x,y) Dot Product of arrays x and y: $\sum_i x_i * y_i$

matmul(x,y) Matrix Multiplication of arrays x and y which can be 1 or 2 dimensional arrays: $z_{i,j} = \sum_k x_{i,k} * y_{k,j}$

conjg(x) Returns the conjugate of x: $a + \imath b \Rightarrow a - \imath b$

cshift(ARRAY, SHIFT, dim) perform a circular shift by SHIFT positions to the left on array ARRAY along the $\mathrm{dim}^{\mathrm{th}}$ dimension

Allocatable Arrays

LEHIGH UNIVERSITY

Why?

- At compile time we may not know the size an array needs to be
- We may want to change the problem size without recompiling

- Allocatable arrays allow us to set the size at run time.

```
real, allocatable :: force(:,:)

real, dimension(:), allocatable :: vel
```

- We set the size of the array using the allocate statement.

```
allocate(force(natoms,3))
```

- We may want to change the lower bound for an array

```
allocate(grid(-100,100))
```

Allocatable Arrays

- We may want to use an array once somewhere in the program, say during initialization. Using allocatable arrays also us to dynamically create the array when needed and when not in use, free up memory using the `deallocate` statement

```
deallocate(force,grid)
```

- Sometimes, we want to check whether an array is allocated or not at a particular part of the code
- Fortran provides an intrinsic function, `allocated` which returns a scalar logical value reporting the status of an array

```
if ( allocated(grid))deallocate(grid)
```

```
if ( .not. allocated(force))allocate(force(natoms,3))
```

Procedures

Program Units

- Most programs are hundreds or more lines of code.
- Use similar code in several places.
- A single large program is extremely difficult to debug and maintain.
- Solution is to break up code blocks into procedures

Subroutines:	Some out-of-line code that is called exactly where it is coded
Functions:	Purpose is to return a result and is called only when the result is needed
Modules:	A module is a program unit that is not executed directly, but contains data specifications and procedures that may be utilized by other program units via the use statement.

Subroutines

- Call Statement:
  - The `call` statement evaluates its arguments and transfers control to the subroutine
  - Upon return, the next statement is executed.
- SUBROUTINE Statement:
  - The `subroutine` statement declares the procedure and its arguments.
  - These are also known as dummy arguments.
- The subroutine's interface is defined by
  - The `subroutine` statement itself
  - The declarations of its dummy arguments
  - Anything else that the subroutine uses

Functions

- `function`s operate on the same principle as `subroutine`s
- The only difference is that `function` returns a value and does not involve the `call` statement

Sum of Two numbers

```
subroutine calc(a,b,c)

  implicit none
  real :: a, b, c

  c = a + b
  return

end subroutine calc
```

```
function calc(a,b)

  implicit none
  real :: a, b, calc

  calc = a + b

end function calc
```

Subroutine Call

```
call calc(x,y,z)
```

Function Call

```
z = calc(x,y)
```

Modules

- Modules were introduced in Fortran 90 and have a wide range of applications.
- Modules allow the user to write object based code.
- A `module` is a program unit whose functionality can be exploited by other programs which attaches to it via the `use` statement.
- `module`s can be compiled separately. They should be compiled before the program unit that uses them.
- The `use` statement names a module whole public definitions are to be made accessible.
- It's good programming practice to use only those variables from modules that are neccessary to avoid name conflicts and overwrite variables.
- For this, use the `use <modulename>, only` statement

Modules

```
module precision
  implicit none
  save
  integer, parameter :: ip = selected_int_kind(15)
  integer, parameter :: dp = selected_real_kind(15)
end module precision

module param
  use precision
  implicit none
  integer(ip) :: npartdim, natom, nstep, istep
  real(dp) :: tempK, dt, boxl(3), alat, mass
  real(dp) :: avtemp, ke, kb, epsilon, sigma, scale
  real(dp),dimension(3,4) :: rcell = reshape( (/ &
       0.0D+00, 0.0D+00, 0.0D+00, &
       0.5D+00, 0.5D+00, 0.0D+00, &
       0.0D+00, 0.5D+00, 0.5D+00, &
       0.5D+00, 0.0D+00, 0.5D+00 /), (/ 3, 4 /) )
  character(len=2) :: pot
end module param


subroutine verlet(coord,force,pener)
  use param,only : dp,npart,boxl,tstep
  ...
end subroutine verlet
```

- Fortran 95/2003 Explained, Michael Metcalf
- Modern Fortran Explaned, Michael Metcalf
- Guide to Fortran 2003 Programming, Walter S. Brainerd
- Introduction to Programming with Fortran: with coverage of Fortran 90, 95, 2003 and 77, I. D. Chivers
- Fortran 90 course at University of Liverpool, `http://www.liv.ac.uk/HPC/F90page.html`
- Introduction to Modern Fortran, University of Cambridge, `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/Fortran`
- Scientific Programming in Fortran 2003: A tutorial Including Object-Oriented Programming, Katherine Holcomb, University of Virginia.
- Fortran Wiki `http://fortranwiki.org/fortran/show/HomePage`

Exercise

Calculate pi by Numerical Integration

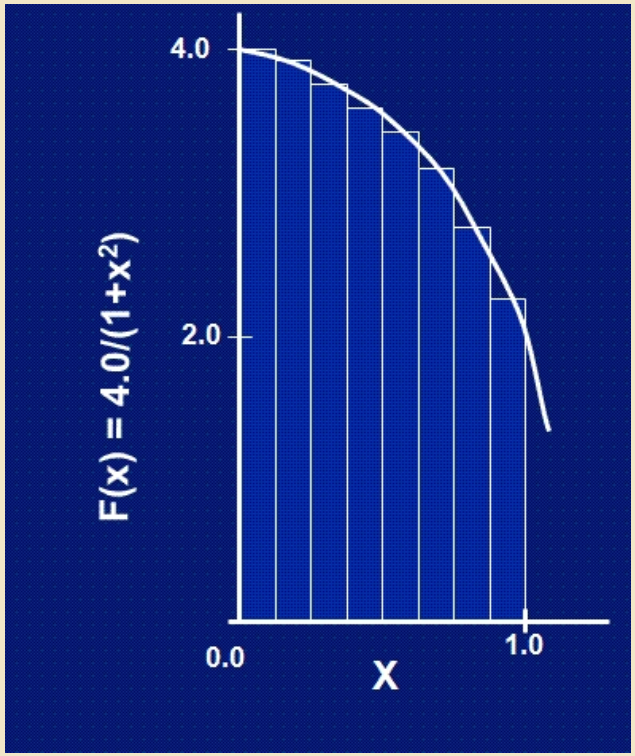
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A "hands-on" introduction to OpenMP, SC09



Calculate pi by Numerical Integration

Algorithm 1 Pseudo Code for Calculating Pi

program CALCULATE_PI
  $step \leftarrow 1/n$
  $sum \leftarrow 0$
  do $i \leftarrow 0 \cdots n$
    $x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$
  end do
  $pi \leftarrow sum * step$
end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$

- Write a SAXPY code to multiply a vector with a scalar.

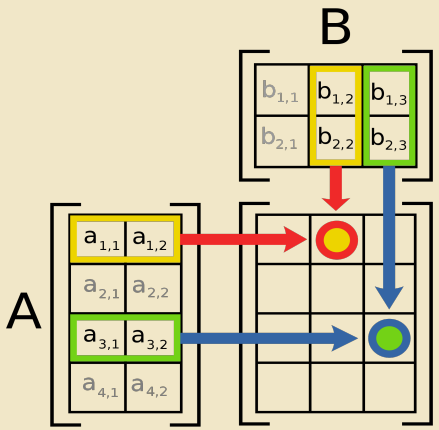
Algorithm 2 Pseudo Code for SAXPY

```
program SAXPY
    n ← some large number
    x(1 : n) ← some number say, 1
    y(1 : n) ← some other number say, 2
    a ← some other number ,say, 3
    do i ← 1 · · · n
        y_i ← y_i + a ∗ x_i
    end do
end program SAXPY
```

Matrix Multiplication

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL