Compiled Languages (Fortran)

RJ LeVeque, UWHPSC

Compiled vs. interpreted language

Not so much a feature of language syntax as of how language is converted into machine instructions.

Many languages use elements of both.

Interpreter:

- Takes commands one at a time, converts into machine code, and executes.
- Allows interactive programming at a shell prompt, as in Python or Matlab.
- Can't take advantage of optimizing over a entire program — does not know what instructions are coming next.
- Must translate each command while running the code, possibly many times over in a loop.

Compiled language

The program must be written in 1 or more files (source code).

These files are input data for the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Why two steps?

Object code contains symbols such as variables that may be defined in other objects. Linker resolves the symbols and converts them into addresses in memory.

Often large programs consist of many separate files and/or library routines — don't want to re-compile them all when only one is changed. (Later we'll use Makefiles.)

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator

Fortran I: 1954–57, followed by Fortran II, III, IV, Fortran 66.

Major changes in Fortran 77, which is still widely used.

> *"I don't know what the language of the year 2000 will look like, but I know it will be called Fortran."*
>
> *– Tony Hoare, 1982*

Fortran history

Major changes again from Fortran 77 to Fortran 90.

Fortran 95: minor changes.

Fortran 2003, 2008: not fully implemented by most compilers.

We will use Fortran 90/95.

gfortran — GNU open source compiler

Several commercial compilers also available.

Fortran syntax

Big differences between Fortran 77 and Fortran 90/95.

Fortran 77 still widely used:

- Legacy codes (written long ago, millions of lines...)
- Faster for some things.

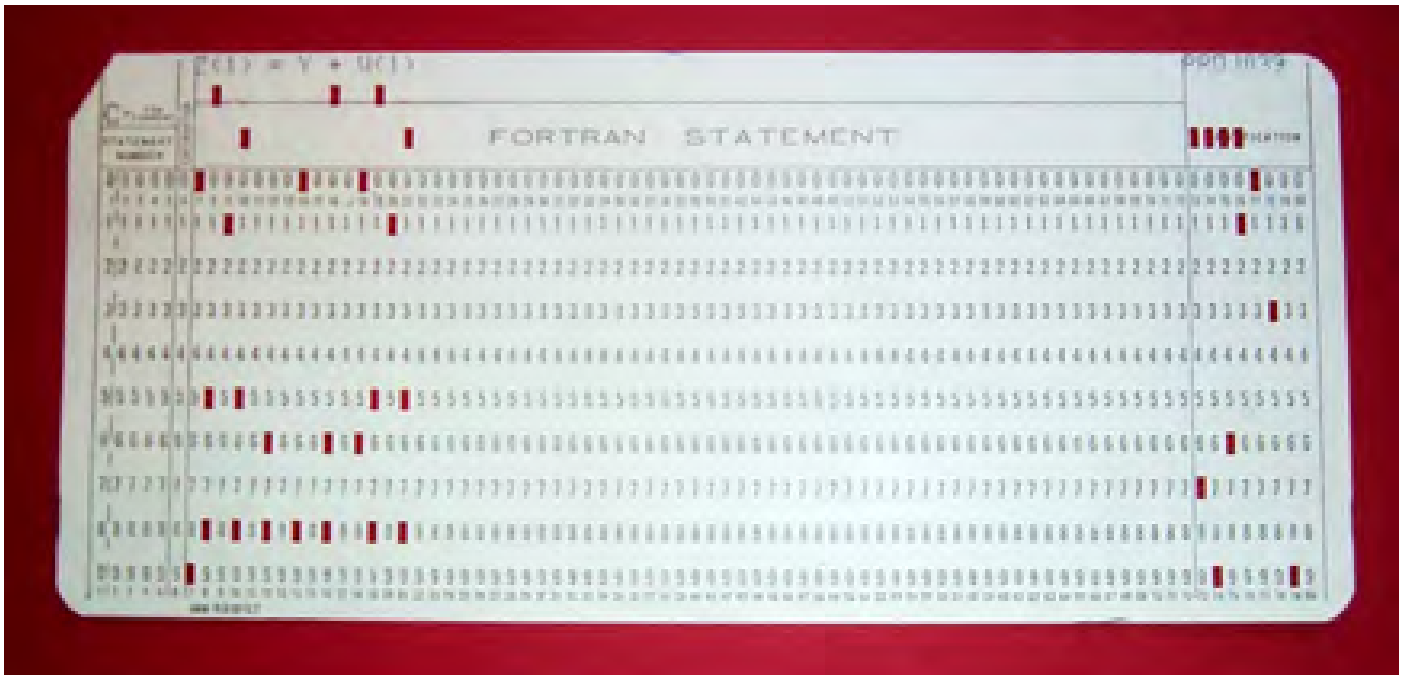
Note: In general adding more high-level programming features to a language makes it harder for compiler to optimize into fast-running code.

Fortran syntax

One big difference: Fortran 77 (and prior versions) required fixed format of lines:

Executable statements must start in column 7 or greater,

Only the first 72 columns are used, the rest ignored!



http://en.wikipedia.org/wiki/File:FortranCardPROJ039.agr.jpg

Punch cards and decks

http://en.wikipedia.org/wiki/File:PunchCardDecks.agr.jpg

Paper tape

http://en.wikipedia.org/wiki/Punched_tape

Fortran syntax

Fortran 90: free format.

Indentation is optional (but highly recommended).

gfortran will compile Fortran 77 or 90/95.

Use file extension `.f` for fixed format (column 7 ...)

Use file extension `.f90` for free format.

Simple Fortran program

```
! /fortran/example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 2.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

Notes:

- Indentation optional (but make it readable!)
- First declaration of variables then executable statements
- implicit none means all variables must be declared

Simple Fortran program

```
! /fortran/example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 2.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- `(kind = 8)` means 8-bytes used for storage,
- `3.d0` means 3×10^0 in double precision (8 bytes)
- `2.d-1` means $2 \times 10^{-1} = 0.2$

Simple Fortran program

```
! /fortran/example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 2.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- `print *, ...`: The `*` means no special format specified
  As a result all available digits of `z` will be printed.
- Later will see how to specify print format.

Compiling and running Fortran

Suppose example1.f90 contains this program.

Then:

```
$ gfortran example1.f90
```

compiles and links and creates an executable named `a.out`

To run the code after compiling it:

```
$ ./a.out
 z =     3.20000000000000
```

The command `./a.out` executes this file (in the current directory).

Compiling and running Fortran

Can give executable a different name with -o flag:

```
$ gfortran example1.f90 -o example1.exe
$ ./example1.exe
  z =    3.20000000000000
```

Can separate compile and link steps:

```
$ gfortran -c example1.f90 # creates example1.o

$ gfortran example1.o -o example1.exe
$ ./example1.exe
  z =    3.20000000000000
```

This creates and then uses the object code example1.o.

Compile-time errors

Introduce an error in the code: (`zz` instead of `z`)

```
program example1
    implicit none
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This gives an error when compiling:

```
 $ gfortran example1.f90
   example1.f90:11.6:
    zz = x + y
     1
Error: Symbol 'zz' at (1) has no IMPLICIT type
```

Without the “implicit none”

Introduce an error in the code: (`zz` instead of `z`)

```
program example1
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This compiles fine and gives the result:

```
$ gfortran example1.f90
$ ./a.out
    z =  -3.62666764177191E-038
```

Or some other random nonsense since `z` was never set.

Fortran types

Variables refer to particular storage location(s), must declare variable to be of a particular type and this won't change.

The statement

```
implicit none
```

means all variables must be explicitly declared.

Otherwise you can use a variable without prior declaration and the type will depend on what letter the name starts with. Default:

- integer if starts with i, j, k, l, m, n
- real (kind=4) otherwise (single precision)

Many older Fortran codes use this convention!

Much safer to use `implicit none` for clarity, and to help avoid typos.

Fortran arrays and loops

```
! /fortran/loop1.f90
program loop1
   implicit none
   integer, parameter :: n = 10000
   real (kind=8), dimension(n) :: x, y
   integer :: i

   do i=1,n
      x(i) = 3.d0 * i
      enddo

   do i=1,n
      y(i) = 2.d0 * x(i)
      enddo

   print *, "Last y computed: ", y(n)
end program loop1
```

Fortran arrays and loops

```
program loop1
    implicit none
    integer, parameter :: n = 10000
    real (kind=8), dimension(n) :: x, y
    integer :: i
```

Comments:

- `integer, parameter` means this value will not be changed.
- `dimension(n) :: x, y` means these are arrays of length `n`.

Fortran arrays and loops

```
do i=1,n
   x(i) = 3.d0 * i
   enddo
```

Comments:

- `x(i)` means i'th element of array.
- Instead of `enddo`, can also use labels...

```
      do 100 i=1,n
         x(i) = 3.d0 * i
100      continue
```

The number 100 is arbitrary. Useful for long loops.
Often seen in older codes.

Fortran if-then-else

```
! /fortran/ifelse1.f90

program ifelse1
    implicit none
    real(kind=8) :: x
    integer :: i

    i = 3

    if (i<=2) then
        print *, "i is less or equal to 2"
    else if (i/=5) then
        print *, "i is greater than 2, not equal to 5"
    else
        print *, "i is equal to 5"
    endif
end program ifelse1
```

Fortran if-then-else

Booleans: `.true.` `.false.`

Comparisons:

`<`	or `.lt.`	`<=`	or `.le.`
`>`	or `.gt.`	`>=`	or `.ge.`
`==`	or `.eq.`	`/=`	or `.ne.`

Examples:

```
if ((i >= 5) .and. (i < 12)) then

if (((i .lt. 5) .or. (i .ge. 12)) .and. &
 (i .ne. 20)) then
```

Note: `&` is the Fortran continuation character.
Statement continues on next line.

Fortran if-then-else

```
! /fortran/boolean1.f90
program boolean1
    implicit none
    integer :: i,k
    logical :: ever_zero

    ever_zero = .false.
    do i=1,10
        k = 3*i - 1
        ever_zero = (ever_zero .or. (k == 0))
        enddo

    if (ever_zero) then
        print *, "3*i - 1 takes the value 0 for some i"
    else
        print *, "3*i - 1 is never 0 for i tested"
    endif
end program boolean1
```

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of modules.

Functions take some input arguments and return a single value.

Usage: `y = f(x)` or `z = g(x,y)`

Should be declared as external with the type of value returned:

```
real(kind=8), external ::  f
```

Fortran subroutines

Subroutines have arguments, each of which might be for input or output or both.

Usage: `call sub1(x,y,z,a,b)`

Can specify the intent of each argument, e.g.

```
real(kind=8), intent(in) :: x,y
real(kind=8), intent(out) :: z
real(kind=8), intent(inout) :: a,b
```

specifies that `x, y` are passed in and not modified,
`z` may not have a value coming in but will be set by `sub1`,
`a, b` are passed in and may be modified.

After this call, `z, a, b` may all have changed.

Array operations in Fortran

Fortran 90 supports some operations on arrays...

```
! $UWHPSC/codes/fortran/vectorops.f90
program vectorops
    implicit none
    real(kind=8), dimension(3) :: x, y

    x = (/10.,20.,30./)          ! initialize
    y = (/100.,400.,900./)

    print *, "x = "
    print *, x

    print *, "x**2 + y = "
    print *, x**2 + y            ! componentwise
```

Array operations in Fortran

```
! $UWHPSC/codes/fortran/vectorops.f90
! continued...

    print *, "x*y = "
    print *, x*y          ! = (x(1)y(1), x(2)y(2), ...)

    print *, "sqrt(y) = "
    print *, sqrt(y)                  ! componentwise


    print *, "dot_product(x,y) = "
    print *, dot_product(x,y)         ! scalar product

end program vectorops
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2,3) :: b
    real(kind=8), dimension(3,3) :: c
    integer :: i

    print *, "a = "
    do i=1,3
        print *, a(i,:)     ! i'th row
        enddo

    b = transpose(a)        ! 2x3 array

    c = matmul(a,b)         ! 3x3 matrix product
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2) :: x
    real(kind=8), dimension(3) :: y

    x = (/5,6/)
    y = matmul(a,x)     ! matrix-vector product
    print *, "x = ",x
    print *, "y = ",y
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[10,20,30], [40,50,60]]
```

Or, could be stored by columns, as done in Fortran!

Multi-dimensional array storage

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (3,2))
Afort = reshape((/10,20,30,40,50,60/), (/3,2/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

loc 3401	Apy[0,0] = 10	Afort(1,1) = 10
loc 3402	Apy[0,1] = 20	Afort(2,1) = 40
loc 3403	Apy[0,2] = 30	Afort(1,2) = 20
loc 3404	Apy[1,0] = 40	Afort(2,2) = 50
loc 3405	Apy[1,1] = 50	Afort(1,3) = 30
loc 3406	Apy[1,2] = 60	Afort(2,3) = 60

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

Memory allocation

```
real(kind=8) dimension(:), allocatable :: x
real(kind=8) dimension(:,:), allocatable :: a

allocate(x(10))
allocate(a(30,10))

! use arrays...

! then clean up:
deallocate(x)
deallocate(a)
```

Passing arrays to subroutines — code with bug

```
1  ! $CLASSHG/codes/fortran/arraypassing1.f90
2
3  program arraypassing1
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing1
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 3.
23     implicit none
24     real(kind=8), intent(inout) :: a(3)
25     integer i
26     do i = 1,3
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: `x` is a scalar, setvals dummy argument `a` is an array.

Passing arrays to subroutines

The `call setvals(x)` statement passes the address where `x` is stored.

In the subroutine, the array `a` of length 3 is assumed to start at this address. So next $3 * 8 = 24$ bytes are assumed to be elements of `a(1:3)`.

In fact these 24 bytes are occupied by

- `x` (8 bytes),
- `y` (8 bytes),
- `i` (4 bytes),
- `j` (4 bytes).

So setting `a(1:3)` changes all these variables!

Passing arrays to subroutines

This produces:

```
x =     5.00000000000000
y =     5.00000000000000
i =    1075052544
j =             0
```

Nasty!!

- The storage location of `x` and the next 2 storage locations were all set to the floating point value 5.0e0
- This messed up the values originally stored in `y, i, j`.
- Integers are stored differently than floats. Two integers take up 8 bytes, the same as one float, so the assignment `a(3) = 5.` overwrites both `i` and `j`.
- The first half of the float 5., when interpreted as an integer, is huge.

Passing arrays to subroutines — another bug

```
1  ! $CLASSHG/codes/fortran/arraypassing2.f90
2
3  program arraypassing2
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing2
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 1000.
23     implicit none
24     real(kind=8), intent(inout) :: a(1000)
25     integer i
26     do i = 1,1000
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: We now try to set 1000 elements in memory!

Passing arrays to subroutines

This compiles fine, but running it gives:

```
Segmentation fault
```

This means that the program tried to change a value of memory it was not allowed to.

Only a small amount of memory is devoted to the variables declared.

The memory we tried to access might be where the program itself is stored, or something related to another program that's running.

Segmentation faults

Debugging segmentation faults can be difficult.

Tip: Compile using `-fbounds-check` option of gfortran.

This catches some cases when you try to access an array out of bounds.

But not the case just shown! The variable was passed to a subroutine that doesn't know how long the array should be.

For a case where this helps, see
`$UWHPSC/codes/fortran/segfault1.f90`