Today:
- Interpreted vs. compiled languages
- Fortran

Wednesday:
- More Fortran
- Computer storage of numbers

Read: Class notes and references.
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.

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.)
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.
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.
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**.)
Simple Fortran program

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

  x = 3.d0
  y = 1.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

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

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

More notes:

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

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

  x = 3.d0
  y = 1.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.
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 = 3.200000000000000
```

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.
Introduce an error in the code: \((zz \text{ instead of } z)\)

```
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: \( \text{zz} \) instead of \( \text{z} \)

```fortran
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 = -3.626667641771191E-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

! $CLASSHG/codes/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

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 4, April 4, 2011
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

```fortran
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...

```fortran
   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

! $CLASSHG/codes/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
Booleans: .true. .false.

Comparisons:

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

Examples:

(i >= 5) .and. (i < 12)
((i .lt. 5) .or. (i .ge. 12)) .and. &
(i .ne. 20)

Note: & is the Fortran continuation character.
Statement continues on next line.
Fortran if-then-else

! $CLASSHG/codes/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))
endo

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

FORTRAN = FORmula TRANslator
Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator


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

Major changes again from Fortran 77 to Fortran 90.
Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator


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

Major changes again from Fortran 77 to Fortran 90.

Fortran 95: minor changes.

Fortran 2003: not fully implemented by most compilers.

We will use Fortran 90/95.
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.
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!
Fortran 90: free format.
Indentation is optional (but highly recommended).
gfortran will compile Fortran 77 or 90/95.
Use file extension .f for Fortran 77.