# Outline of Fortran 90 Topics

- Overview of Computing
  - Computer Organization
  - Languages
  - Problem Solving
- Data
  - Fortran 90 Character Set
  - Variables
  - Basic Data Types
- Expressions
  - Numeric
  - Non-numeric
- Control Structures
  - Branching
  - Repetition
- More Data
  - 1D Arrays
  - Derived Data Types
- Procedures
  - Subroutines
  - Functions
- Modularity
  - Modules
  - Interface Blocks
- Input/Output
  - Formatting
  - Files

Note: this is **roughly** how this part of the semester will go.

# Data

- Now:
  - Fortran 90 Character Set
    (*Programming in Fortran 90/95*, Chapter 5)
  - Variables
    (*Programming in Fortran 90/95*, Chapters 5-6)
  - Basic Data Types
    (*Programming in Fortran 90/95*, Chapter 6)
- Later:
  - 1D Arrays
    (*Programming in Fortran 90/95*, Chapter 7)
  - Derived Data Types
    (*Programming in Fortran 90/95*, Chapter 8)

# Fortran 90 Character Set

These are the characters that Fortran 90 recognizes.

- Letters

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

- Digits

```
0 1 2 3 4 5 6 7 8 9
```

- Special Characters

  space

```
' " ( ) * + - / :  = _
!  & $ ; < > % ?  , .
```

Fortran 90 is **case insensitive**: it doesn't distinguish between upper case and lower case letters, except those inside **character strings**.

```
PROGRAM helloworld
  IMPLICIT NONE
  PRINT *, 'Hello, world!'
END PROGRAM helloworld
```

is treated the same as

```
program helloWORLD
  iMpLiCiT NoNE
  prinT *, 'Hello, world!'
end proGRAM HELLOworld
```

# Variables

A **variable** is an association (sometimes called a **binding**) between a name and a location in memory.

## Fortran 90 Variable Declaration

```
INTEGER ::  x
```

This tells the compiler to choose a location in memory, call it `x`, and think of it as an integer.

The compiler might decide that `x` will live at, say, address 3989 or address 98234098 or address 56435. We neither know nor care what address `x` lives at, because the compiler will take care of that for us.

`x :` `????????` (Address `56435`)

When `x` is first declared, we don't know what its value is, because we haven't put anything into its memory location yet.

**Note**: some compilers for some languages automatically initialize newly declared variables to default values (e.g., all integers get initialized to zero), but not every compiler does. You should **never** assume that the compiler will initialize your variables for you.

# Variable Assignment

```
x = 5
```

This tells the compiler to put the integer value 5 into the memory location it calls x, like so:

```
x :  [      5 ]  (Address 56435)
```

So, for example, we might have:

```
INTEGER :: x
      ⇓
x :  [???????? ]  (Address 56435)
```

```
x = 5
  ⇓
x :  [      5 ]  (Address 56435)
```

```
x = x + 7
    ⇓
x :  [     12 ]  (Address 56435)
```

# Variable Assignment Example

```
1% cat xvardec.f90
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!! Program: xvardec                    !!!
  !!! Module:  main                       !!!
  !!! Author:  Henry Neeman 123-45-6789 !!!
  !!! Class:   CS 1313 010 Spring 2000  !!!
  !!! Lab:     Sec 013 Fridays 2:30pm   !!!
  !!! Description: Declares, assigns    !!!
  !!!   and prints a variable.          !!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  PROGRAM xvardec
    !
    ! Declarations
    !
    ! No implicitly typed variables.
    IMPLICIT NONE
    !
    INTEGER :: x
    !
    ! Execution
    !
    ! Assign x the integer value 5
    x = 5
    ! Print x to stdout
    PRINT *, 'x = ', x
  END PROGRAM xvardec
2% f90 -o xvardec xvardec.f90
3% xvardec
 x =           5
```

# Variable Initialization

We can **initialize** a variable's value in the variable declaration:

```
INTEGER ::  x = 5
```

```
1% cat xvardecinit.f90
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!! Program: xvardecinit                !!!
  !!! Module:  main                       !!!
  !!! Author:  Henry Neeman 123-45-6789 !!!
  !!! Class:   CS 1313 010 Spring 2000  !!!
  !!! Lab:     Sec 013 Fridays 2:30pm   !!!
  !!! Description: Declares with        !!!
  !!!   initialization, then prints a   !!!
  !!!   variable                        !!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  PROGRAM xvardecinit
    !
    ! Declarations
    !
    ! No implicitly typed variables.
    IMPLICIT NONE
    !
    INTEGER :: x = 5
    !
    ! Execution
    !
    ! Print x to stdout
    PRINT *, 'x = ', x
  END PROGRAM xvardecinit
2% f90 -o xvardecinit xvardecinit.f90
3% xvardecinit
 x =           5
```

# Variable Names

Fortran 90 **symbolic names** (including variable names) are constructed of the following characters:

- Letters (case insensitive)
  `a b c d e f g h i j k l m`
  `n o p q r s t u v w x y z`
- Digits
  `0 1 2 3 4 5 6 7 8 9`
- Underscore
  _

These are their properties:

- length $\leq 31$:
  `name56789012345678901234567890`1
  but not
  `name567890123456789012345678901`2
- first character is a letter:
  `a123_456` but not `1a23_456` nor `_a123_456`

So, since there are 26 letters, 10 digits and 1 underscore, the number of possible variable names is:

$$\sum_{i=0}^{30} 26 \cdot 37^i \;=\; 2,084,172,676$$

Note: symbolic names are also called **identifiers.**

# Implicitly Typed Variables Are BAD BAD BAD

Most versions of Fortran, including Fortran 90, support **implicitly typed variables**: if a variable is not explicitly declared

```
INTEGER ::  x
```

then it will be implicitly declared the first time it appears, and its type will depend on the first letter of its variable name:

- If the first letter of the variable name is one of
  `i j k l m n`
  then the variable is implicitly declared as an integer.
- Otherwise, the variable is implicitly declared as a real.

This leads to all kinds of problems:

```
% cat ruin.f90
      PROGRAM ruin
        INTEGER :: bankrupt = 1
        bankrpt = bankrupt - 1
        IF (bankrupt > 0) THEN
          PRINT *, 'Bankrupt!'
        ELSE
          PRINT *, 'Whew!'
        END IF
      END PROGRAM ruin
% f90 -o ruin ruin.f90
% ruin
 Bankrupt!


% cat dontruin.f90
      PROGRAM dontruin
        IMPLICIT NONE
        INTEGER :: bankrupt = 1
        bankrpt = bankrupt - 1
        IF (bankrupt > 0) THEN
          PRINT *, 'Bankrupt!'
        ELSE
          PRINT *, 'Whew!'
        END IF
      END PROGRAM dontruin
% f90 -o dontruin dontruin.f90
f90: Error: dontruin.f90, line 4:
  This name does not have a type, and must have an explicit type.
    [BANKRPT]
        bankrpt = bankrupt - 1
--------^
```

So we **ALWAYS ALWAYS ALWAYS** use `IMPLICIT NONE`, which turns off implicit data declaration.

# Declaring Constants

If a symbolic name represents a value that will never change, then we call it a **named constant**. In its declaration, we indicate that it's a constant by using the PARAMETER specifier:

```
REAL,PARAMETER ::  PI = 3.1415926
```

```
% cat circlecalc.f90
    PROGRAM circlecalc
      IMPLICIT NONE
      REAL,PARAMETER :: pi = 3.1415926
      REAL,PARAMETER :: diameter_factor = 2.0
      REAL,PARAMETER :: area_power = 2.0
      REAL :: radius, perimeter, area
      PRINT *, "I'm going to calculate the ", &
     &         "perimeter and area"
      PRINT *, "  of a circle."
      PRINT *, "What's the radius of the circle?"
      READ *, radius
      perimeter = pi * radius * diameter_factor
      area = pi * radius ** area_power
      PRINT *, "The perimeter is ", perimeter
      PRINT *, "  and the area is ", area, "."
    END PROGRAM circlecalc
% f90 -o circlecalc circlecalc.f90
% circlecalc
 I'm going to calculate the perimeter and area
   of a circle.
 What's the radius of the circle?
 5
 The perimeter is    31.41592
   and the area is    78.53981    .
```

What if we try to assign a value to a named constant?

```
% cat paramassign.f90
      PROGRAM paramassign
        IMPLICIT NONE
        REAL,PARAMETER :: pi = 3.1415926
        pi = pi * 2.0
      END PROGRAM paramassign
% f90 -o paramassign paramassign.f90
f90: Error: paramassign.f90, line 4:
This PARAMETER constant name is invalid in this context.   [PI]
      pi = pi * 2.0
------^
```

# Assigning Values to Variables via List-Directed Input

Another way to get a value into a variable is by reading it from the keyboard:

```
% cat xvardecread.f90
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!! Program: xvardecread                !!!
  !!! Module:  main                       !!!
  !!! Author:  Henry Neeman 123-45-6789 !!!
  !!! Class:   CS 1313 010 Spring 2000  !!!
  !!! Lab:     Sec 013 Fridays 2:30pm   !!!
  !!! Description: Declares, reads       !!!
  !!!   and prints a variable.           !!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  PROGRAM xvardecread
    !
    ! Declarations
    !
    ! No implicitly typed variables.
    IMPLICIT NONE
    !
    INTEGER :: x
    !
    ! Execution
    !
    ! Ask the user to enter the value
    !  of x.
    PRINT *, "What's the value of x?"
    ! Input x
    READ *, x
    ! Print x to stdout
    PRINT *, 'x = ', x
  END PROGRAM xvardecread
% f90 -o xvardecread xvardecread.f90
% xvardecread
 What's the value of x?
 5
 x =           5
```

# Multiple Variables Per Input

```
% cat xvarsread.f90
    PROGRAM xvarsread
      IMPLICIT NONE
      INTEGER :: number_of_silly_people, number_of_toys
      REAL    :: silliness_standard_deviation
      LOGICAL :: toys_put_away
      CHARACTER (LEN = 20) :: username, hometown
      PRINT *, "How many silly people are there,"
      PRINT *, "  and what's the standard deviation"
      PRINT *, "  of their silliness index?"
      READ *, number_of_silly_people, &
   &          silliness_standard_deviation
      PRINT *, "There are ", number_of_silly_people, &
   &           " silly people"
      PRINT *, " with a silliness standard deviation of ", &
   &           silliness_standard_deviation, "."
      PRINT *, "How many toys do I have, and"
      PRINT *, "  is it true that I put then away?"
      READ *, number_of_toys, toys_put_away
      PRINT *, 'I have ', number_of_toys, ' toys.'
      PRINT *, "Did I put my toys away? : ", toys_put_away
      IF (toys_put_away) THEN
        PRINT *, "  Yes I did."
      ELSE
        PRINT *, "  No I didn't."
      END IF
      PRINT *, "What's my username and hometown?"
      READ *, username, hometown
      PRINT *, 'My username is ', username
      PRINT *, 'and my hometown is ', hometown, '.'
    END PROGRAM xvarsread
% f90 -o xvarsread xvarsread.f90
% xvarsread
 How many silly people are there,
   and what's the standard deviation
   of their silliness index?
 7,0.74
 There are            7 silly people
  with a silliness standard deviation of   0.7400000    .
 How many toys do I have, and
   is it true that I put then away?
 43 T
 I have           43 toys.
 Did I put my toys away? :  T
   Yes I did.
 What's my username and hometown?
 neeman
    Buffalo
 My username is neeman
 and my hometown is Buffalo             .
```

# Program Variables vs. Algebra Variables

Variables in Fortran 90 (and many other programming languages) look and feel very similar to variables that you deal with in your math classes, from high school algebra on up.

This is on purpose.

The main difference between an algebra variable and a program variable is that a program variable can change its value during a run:

| **Algebra** | **Fortran 90** | **Output** |
|---|---|---|
| Let $x$ be 5. | `x = 5` | |
| | `PRINT *, x` | `5` |
| Let $y$ be 7. | `y = 7` | |
| | `PRINT *, y` | `6` |
| $z = x + y$ | `z = x + y` | |
| $\therefore z = 12$ | `PRINT *, z` | `12` |
| | `z = z *2` | |
| | `PRINT *, z` | `24` |

# How Your Programs Should Look

```
PROGRAM progform
IMPLICIT NONE
!
! Constants
!
REAL,PARAMETER    :: pi = 3.1415926
INTEGER,PARAMETER :: string_length = 20
 ...
!
! Variables
!
CHARACTER (LEN=string_length) :: mystring
COMPLEX :: mycomplex1,mycomplex2
REAL    :: myreal1, myreal2
INTEGER :: myinteger1, myinteger2
LOGICAL :: myflag1 = .FALSE., myflag2 = .TRUE.
 ...
!
! Execution
!
PRINT *, "Start Program progform"
PRINT *, "  Shows the proper format for a program."
 ...
PRINT *, "End Program progform"
END PROGRAM progform
```