# Outline of Fortran 90 Topics

- Overview of Computing
  - Computer Organization
  - Languages
  - Problem Solving
- Data, Part 1
  - Fortran 90 Character Set
  - Variables
  - Numeric Data Types
- Numeric Expressions
- Data, Part 2: Non-numeric Data Types
- Non-numeric Expressions
- Control Structures
  - Branching
  - Repetition
- Data, Part 3
  - 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)
  - Numeric Data Types
    (*Programming in Fortran 90/95*, Chapter 6)
- Later:
  - Non-numeric Data Types
    (*Programming in Fortran 90/95*, Chapter 6)
  - 1D Arrays
    (*Programming in Fortran 90/95*, Chapter 7)
  - Derived Data Types
    (*Programming in Fortran 90/95*, Chapter 8)

# Basic Data Types

- Numeric
  - Integer
  - Real
  - Complex
- Non-numeric
  - Character
  - Logical

```
PROGRAM bastypedec
  IMPLICIT NONE
  INTEGER :: count, number_of_silly_people
  REAL    :: standard_deviation, relative_humidity
  COMPLEX :: quadratic_root_1, quadratic_root_2
  LOGICAL :: count_is_less_than_5, I_am_Henry
  CHARACTER (LEN = 20) :: username, hometown
END PROGRAM bastypedec
```

# Reals

Mathematically, a *real number* is a number (positive, negative or zero) with any string of digits on either side of the decimal point:

$$-3984.8979729\ldots \qquad 3.1415926\ldots \qquad 0.1111111\ldots$$

In the computer, a real variable can only *approximate* this mathematical property, because it is stored in a finite number of bits.

Like integers, reals have particular ways of being stored in memory and of being operated on.

# Real Declaration

```
REAL ::  x
```

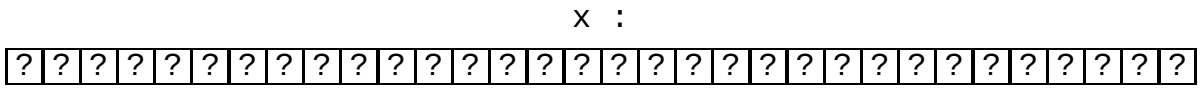
Here, the compiler grabs a group of bytes and calls them `x`.

## How many bytes?

That depends on the compiler. For example, on `ecnalpha`, the size of a real can be set with a compiler option:

```
f90 -real_size 32 -o bastypedec bastypedec.f90
```

The compiler option `-real_size 32` tells the compiler that reals will be 32 bits (4 bytes) long:




Likewise, `-real_size 64` and `-r8` both tell the compiler to make reals 64 bits (8 bytes), and `-real_size 128` and `-r16` both tell the compiler to make reals 128 bits (16 bytes).

Oddly enough, the DEC Alpha compiler doesn't have an `-r4` option.

On DEC Alphas like `ecnalpha`, and on many other platforms, the default for reals is 32 bits (4 bytes), so if you don't use one of the `-real_size` or `-r` options, the compiler will assume that you want `-real_size 32`. (And in fact, in our programming projects we will use the default real size.)

A notable exception is Cray's line of vector machines (e.g., the Cray J90), whose default real size is 64 bits (8 bytes).

# Scientific Notation

In technical classes, we often encounter *scientific notation*, which is a way of writing numbers that are either very very big or very very small:

$$\begin{aligned} 6,300,000,000,000,000 &= 6.3 \times 10^{18} \\ 0.0000000000271 &= 2.71 \times 10^{-11} \end{aligned}$$

In Fortran 90, we can express such numbers in a similar way:

$$\begin{aligned} 6,300,000,000,000,000 &= \texttt{6.3E+18} \\ 0.0000000000271 &= \texttt{2.71E-11} \end{aligned}$$

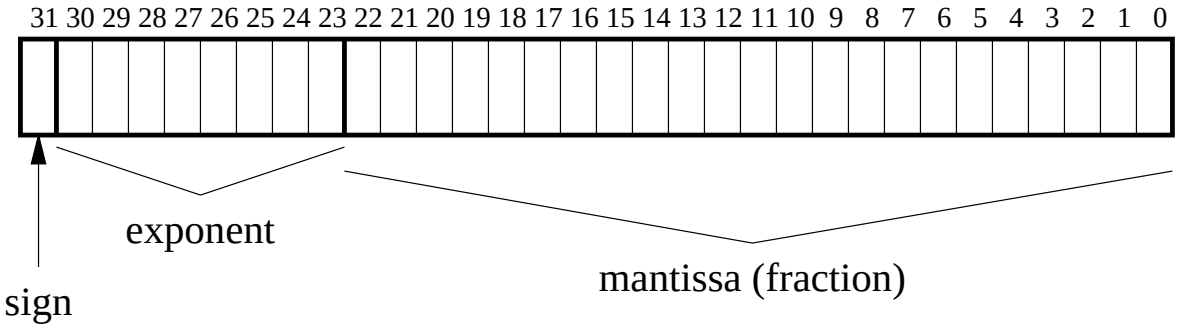
Because a real number can have its decimal point anywhere within its string of digits, we sometimes call real numbers *floating point* numbers.

Similarly, integers are sometimes called *fixed point* numbers, because they have an implicit decimal point that is always to the right of the "1's" digit (i.e., the rightmost digit), with implied zeros to the right of the implied decimal point:

$$6,300,000,000,000,000 = 6,300,000,000,000,000.0000\ldots$$

# How Are Reals Represented in Memory?

In computers, a real is represented in a manner analogous to scientific notation. There are endless ways to configure this. Here's one:




6,300,000,000: sign=0, exponent=19,mantissa=0.63

Note: for the exams, you need to know that a real is represented by a sign bit, a string of exponent bits, and a string of mantissa bits. You do not need to know the exact configuration, since this can vary from platform to platform.

# Why Can Fortran 90 Reals Only Approximate Mathematical Reals?

In Fortran 90 (and in most other computer languages), real numbers are represented by a finite number of bits.

For example, on Dec Alphas like `ecnalpha`, the default size of a real number is 32 bits (4 bytes).

We know that 32 bits can store $2^{32} \approx 4,000,000,000$ possible values. And that's a lot of possibilties.

But: there are infinitely many (mathematically) real numbers, and in fact infinitely many real numbers between any two real numbers. For example, between 1 and 10 we have:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2.9 | 3.8 | 4.7 | 5.6 | 6.5 | 7.4 | 8.3 | 9.2 |
| 2.09 | 3.08 | 4.07 | 5.06 | 6.05 | 7.04 | 8.03 | 9.02 |
| 2.009 | 3.008 | 4.007 | 5.006 | 6.005 | 7.004 | 8.003 | 9.002 |
| 2.0009 | 3.0008 | 4.0007 | 5.0006 | 6.0005 | 7.0004 | 8.0003 | 9.0002 |

...

So, no matter how many bits we use to represent a real number, we won't be able to exactly represent most real numbers, because we have an infinite set of real numbers to be represented in a finite number of bits.

For example:
if we can exactly represent 0.125 but not
0.12500000000000000000000000000001,
then we use 0.125 to approximate
0.12500000000000000000000000000001.

# Real Approximation Example

```
% cat realapprox.f90
  PROGRAM realapprox
    IMPLICIT NONE
    REAL :: input_value
    PRINT *, "What real value would you like stored?"
    READ *, input_value
    PRINT *, "That real value is stored as ", &
   &         input_value, "."
  END PROGRAM realapprox
% f90 -o realapprox realapprox.f90
% realapprox
 What real value would you like stored?
 0.125000000000000000000000000000001
 That real value is stored as   0.1250000     .
```

# Real Constants

A *real constant*, sometimes called a real *literal constant*, is an optional sign, a string of digits, a decimal point, an optional string of digits, and an optional exponent string, which consists of an E, an optional sign, and a string of digits.

```
0.      -345.3847     7.68E+05
  +12345.434E-13      125.E1
```

We can use real literal constants in declaring named constants:

```
REAL,PARAMETER ::  w = 0.0
```

in initializing declared variables:

```
REAL ::  x = -1E-05
```

in assignments:

```
y = +7.246901200
```

and in expressions:

```
z = y + 125E3
```

# Why Have Both Reals and Integers?

1. Precision: Integers are exact, reals are approximate.
2. Appropriateness: For some tasks, integers are better.
   For example:
   - counting the number of students in a class
   - array subscripting (which we'll see later on)
3. Readability: When we declare a variable to be an integer, we make obvious to anyone reading our program the fact that the variable will only have certain values.
4. Enforcement: When we declare a variable to be an integer, no one can put a non-integer into it.
5. History: For a long time, operations on integers were much quicker than operations on reals, so anything you could do with integers, you would.

   Nowadays, operations on reals can be as fast, or almost as fast, as operations on integers.

# Complex Numbers

Mathematically, a *complex number* is an ordered pair of real numbers; the first is called the *real part* and the second is called the *imaginary part*.

$$(24.0, 7.0) \quad (-3984.8979729\ldots, 3.1415926\ldots) \quad (123.456, 0.0)$$

A complex number whose imaginary part is zero is mathematically the same as the real number in its real part.

In the computer, a complex number is represented as an ordered pair of real numbers, and is stored as such – even if its imaginary part is zero.

Like integers and reals, complex numbers have particular ways of being stored in memory and of being operated on.

We will discuss complex numbers briefly, but for the most part we will not use them in this class, since they are only used in highly specialized applications.

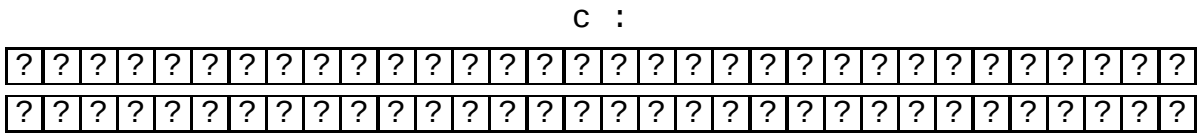
# Complex Declaration

```
COMPLEX ::  c
```

Here, the compiler grabs a group of bytes and calls them `c`.

## How many bytes?

Because a complex number is stored as a pair of real numbers, it takes up twice as much space as a real number.



# Complex Constants

A *complex constant*, sometimes called a complex *literal constant*, is an ordered pair of real literal constants, in parentheses and separated by a comma:

```
(5.7,-10.941)
```

We can use complex literal constants in declaring named constants:

```
COMPLEX,PARAMETER ::  w = (0.0,7.5)
```

in initializing declared variables:

```
COMPLEX ::  x = (9.2,234.01)
```

in assignments:

```
y = (+7.246901200,49479E12)
```

and in expressions:

```
z = y + (125E3,-0.00000001)
```