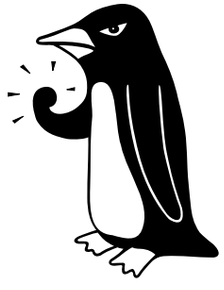
Last bits of Fortran



Warwick RSE

There are only two kinds of languages: the ones people complain about and the ones nobody uses.

- Bjarne Stroustrup - The inventor of C++

Intent Again

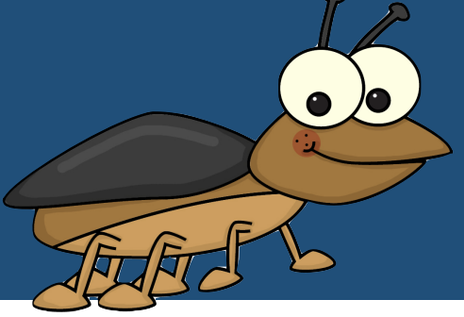
Remember Intent?

- The **INTENT** statement tells the compiler what you are intending to do with a dummy variable

  - INTENT(IN) - I want to use the value that this variable has when my subroutine is called inside my subroutine. This is the **only** intent that allows passing a literal value or the result of another function

  - INTENT(OUT) - I want to set the value of this variable. I neither want nor have access to it's value at the calling point of my subroutine

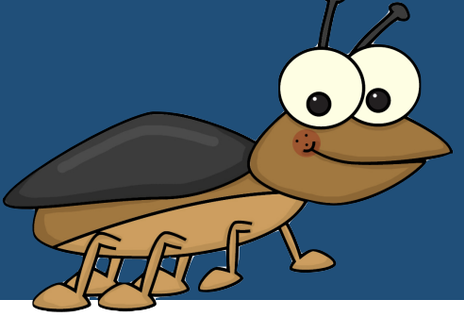
  - INTENT(INOUT) - I want to both know the value of this variable when the subroutine is called and change the value. Almost but not quite the same as not specifying intent



Bug!

```
MODULE mymodule

 CONTAINS

  SUBROUTINE assignfn(arg1, arg2)
    INTEGER :: arg1, arg2
    arg2 = arg1
  END SUBROUTINE assignfn

END MODULE mymodule

PROGRAM driver
  USE mymodule
  IMPLICIT NONE

  CALL assignfn(1234,5678)
END PROGRAM driver
```

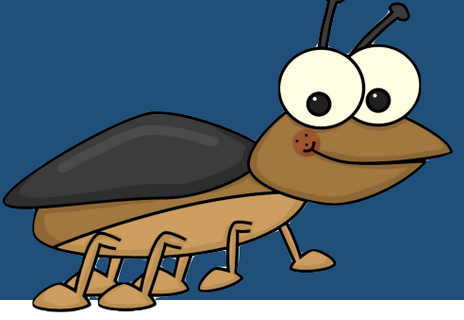
- This code will crash!
- CALL line is equivalent to **5678=1234**
  - **NONSENSE!**
- Because I am calling the function with two **literal values** setting one of them equal to the other is nonsense



Bug!

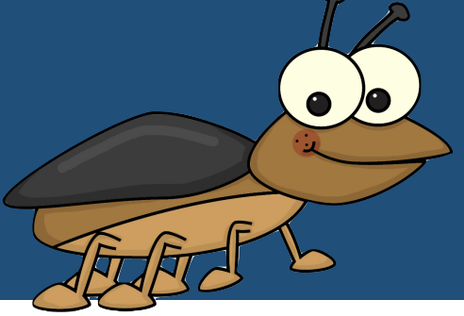
```
MODULE mymodule

 CONTAINS

  SUBROUTINE assignfn(param1, param2)
    INTEGER :: param1, param2
    param2 = param1
  END SUBROUTINE assignfn

END MODULE mymodule

PROGRAM driver
  USE mymodule
  IMPLICIT NONE
  INTEGER :: a, b
  a = 1234

  CALL assignfn(a, b)
END PROGRAM driver
```

- Note that the bug is in the CALL line, NOT the subroutine
- If you have variables instead of literals in the CALL assign line it'll work fine
- Is there a way of saying what you are going to do to a variable to stop this problem?
  - Yes, INTENT statements



Bug!

```
MODULE mymodule

  CONTAINS

    SUBROUTINE assignfn(arg1, arg2)
      INTEGER, INTENT(IN)  :: arg1
      INTEGER, INTENT(OUT) :: arg2
      arg2 = arg1
    END SUBROUTINE assignfn

END MODULE mymodule

PROGRAM driver
  USE mymodule
  IMPLICIT NONE

  CALL assignfn(1234,5678)
END PROGRAM driver
```



Bug!

```
broken.f90:17:21:

   17 |   CALL assignfn(1234,5678)
      |                     1
Error: Non-variable expression in variable definition context (actual argument to INTENT = OUT/INOUT) at (1)
```

- Effect of INTENT statement is to let the compiler know that I want to write data to arg2

- The compiler knows that this is invalid so it fails during compilation with a fairly helpful error message

- Much better than a crash at runtime

Intent

- **ONLY DUMMY ARGUMENTS CAN HAVE INTENT**

- If you apply an intent attribute to any other variable your code will fail to compile

- Intent helps you to avoid lots and lots of problems so you should ***always give every dummy variable an intent***

- There are a few more exotic things where you can't specify an intent but we only cover these in extension material

- Many codes in the wild don't bother but intent can save a lot of heartache

SAVE attribute

SAVE and Implicit SAVE

- Fortran variables can have quite a lot of attributes attached to them.
  - You've already seen intent for dummy variables and PARAMETER
  - More will come in the next section about arrays
- There's one useful but rather slippery one, the SAVE attribute
  - SAVE means that a variables should keep it's state between calls to a function or subroutine
  - Same idea as SAVE in a module but applied to a single variable

Save attribute

```
MODULE mymodule

 CONTAINS

  FUNCTION running_max(arg)
    INTEGER, INTENT(IN) :: arg
    INTEGER :: running_max
    INTEGER, SAVE :: current_max = 0

    current_max = MAX(current_max, arg)
    running_max = current_max
  END FUNCTION running_max

END MODULE mymodule
```

```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE

  PRINT *, running_max(1)
  PRINT *, running_max(10)
  PRINT *, running_max(2)
  PRINT *, running_max(20)

END PROGRAM driver
```

```
 1
10
10
20
```

Save attribute

```
MODULE mymodule

 CONTAINS

  FUNCTION running_max(arg)
    INTEGER, INTENT(IN) :: arg
    INTEGER :: running_max
    INTEGER :: current_max = 0

    current_max = MAX(current_max, arg)
    running_max = current_max
  END FUNCTION running_max

END MODULE mymodule
```

```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE

  PRINT *, running_max(1)
  PRINT *, running_max(10)
  PRINT *, running_max(2)
  PRINT *, running_max(20)

END PROGRAM driver
```

```
 1
10
10
20
```

SAVE and Implicit SAVE

- You get the same result!!!
- There's a rather annoying feature of Fortran called "implicit save"
- If you assign a variable a value on the same line where you define it **it automatically becomes a SAVE variable whether you specify SAVE or not!**
- So by assigning a value of 0 to **`current_max`** I make it a save variable anyway
  - This is required by standard but I would suggest always putting a SAVE attribute on a variable that you want to have SAVEd just to make it clear

SAVE and Implicit SAVE

- If you want to avoid implicit save behaviour you have to put the definition of the initial state of a variable on a separate line

```
SUBROUTINE demo()
    !Implicit save
    INTEGER:: var = 0
END SUBROUTINE demo
```

```
SUBROUTINE demo()
    !Not implicit save
    INTEGER:: var
    var = 0
END SUBROUTINE demo
```

Variable Kinds

Number Storage Sizes

- If we have 8 bits of space (1 byte), we have 2^8 = 256 distinct values
  - Either 0 to 255 or -127 to 128 as integers
    - Could in theory have *any* set of 256 distinct values but these are the two that languages support natively
- Fortran doesn't (yet) have unsigned variables so and only has the second variant

Number Storage Sizes

- Using more bits to store the number lets you store larger numbers but
  - Uses more memory
    - Takes more time to do computations with (with caveats)
- Want to select variables that are large enough for your purpose but no larger

Number Storage Sizes

- Because Fortran comes from an era where computers were rapidly changing, the standard mandates as little as possible
    - As programmer you know how much space you ***need***
        - Generally, more isn't a problem
    - Computer is faster using types it natively understands
        - Compiler writers for given architecture knew what this was
- You specify how much is enough and get at least that

Fortran Variable KINDs

- Rather than different names for different lengths of variable, Fortran uses a KIND parameter to INTEGER and REAL types to specify how long they should be

- **`SELECTED_INT_KIND(R)`** - Give me an integer kind that can hold numbers of at least 10^R

- **`SELECTED_REAL_KIND(P,R)`** - Give me a real or complex kind that has at least P digits of precision and can represent numbers of at least 10^R

- Effect of using a REAL kind on an INTEGER or vice versa is not defined

- Usually you create these kinds once and store them in an **`INTEGER, PARAMETER`**

Fortran variables

```
PROGRAM variables
  IMPLICIT NONE
  INTEGER, PARAMETER :: ik = &
    SELECTED_INT_KIND(5)
  INTEGER(ik) :: myint

  myint = 10_ik
END PROGRAM variables
```

- Define parameter “ik”, and re-use for every integer that you want to specify the length of
- You should also specify any literals in your code with “kind warts” to make sure that the compiler understands your literal

Fortran 2003

```
INTEGER, PARAMETER :: INT8 = SELECTED_INT_KIND(2)
INTEGER, PARAMETER :: INT16 = SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: INT32 = SELECTED_INT_KIND(9)
INTEGER, PARAMETER :: INT64 = SELECTED_INT_KIND(15)
INTEGER, PARAMETER :: REAL32 = SELECTED_REAL_KIND(6, 37)
INTEGER, PARAMETER :: REAL64 = SELECTED_REAL_KIND(15, 307)
INTEGER, PARAMETER :: REAL128 = SELECTED_REAL_KIND(33, 4931)
```

- Fortran 2003 SELECTED_REAL_KIND and SELECTED_INT_KINDS for

- 8,16,32,64 bit integers (char, short, long, long long in C)

- 32, 64, 128(?) bit reals (float, double, ? in C)

Fortran 2008

- With modern computers the idea of requesting “at least with this capability” feels a bit old fashioned

- Fortran 2008 introduced a module called ISO_FORTRAN_ENV which defines actual fixed length types for integers and reals

- INT8, INT16, INT32, INT64

- REAL32, REAL64, REAL128

- Be careful with REAL128 doesn’t mean what the name suggests

  - Requires more than 64 bits but otherwise unspecified

Derived Types

Derived types

- It’s quite common to have situations where you effectively have a “thing” that has several different properties that describe it

- It’s useful to be able to bundle all of these properties together into a single representation

- In Fortran these representations are called Derived Types

Derived types

- There are three elements to creating a derived type variable

  - First you have create the definition of the derived type

  - Then you have to create **instances** of the type that are actual variables

  - Then you have to be able to access the data in the derived type instance

Derived types

```
PROGRAM typetest

  IMPLICIT NONE
  !This defines a type
  TYPE :: mytype
    INTEGER :: myint
    REAL :: myreal
  END TYPE

  !This creates an instance of my type
  TYPE(mytype) :: t

  t%myint = 10
  t%myreal = 1.5

  PRINT *, t%myint
  PRINT *, t%myreal

END PROGRAM typetest
```

Derived types

```
PROGRAM typetest

  IMPLICIT NONE
  !This defines a type
  TYPE :: mytype
    INTEGER :: myint
    REAL :: myreal
  END TYPE

  !This creates an instance of my type
  TYPE(mytype) :: t

  t%myint = 10
  t%myreal = 1.5

  PRINT *, t%myint
  PRINT *, t%myreal

END PROGRAM typetest
```

Derived types

```
PROGRAM typetest

  IMPLICIT NONE
  !This defines a type
  TYPE :: mytype
    INTEGER :: myint
    REAL :: myreal
  END TYPE
```

```
  !This creates an instance of my type
  TYPE(mytype) :: t

  t%myint = 10
  t%myreal = 1.5

  PRINT *, t%myint
  PRINT *, t%myreal

END PROGRAM typetest
```

Derived types

```
PROGRAM typetest

  IMPLICIT NONE
  !This defines a type
  TYPE :: mytype
    INTEGER :: myint
    REAL :: myreal
  END TYPE
```

```
  !This creates an instance of my type
  TYPE(mytype) :: t

  t%myint = 10
  t%myreal = 1.5

  PRINT *, t%myint
  PRINT *, t%myreal

END PROGRAM typetest
```

Derived types

- You define a type inside a **TYPE{name} / END TYPE{name}** block

- You get an instance of the type using **TYPE(name) :: {instance_name}**

    - This is just like a normal variable definition and you can put attributes like **DIMENSION** on it

- You access elements of the type using "{instance_name}%{element_name}"

    - Yes "." is more common but Fortran uses "%"

Derived types

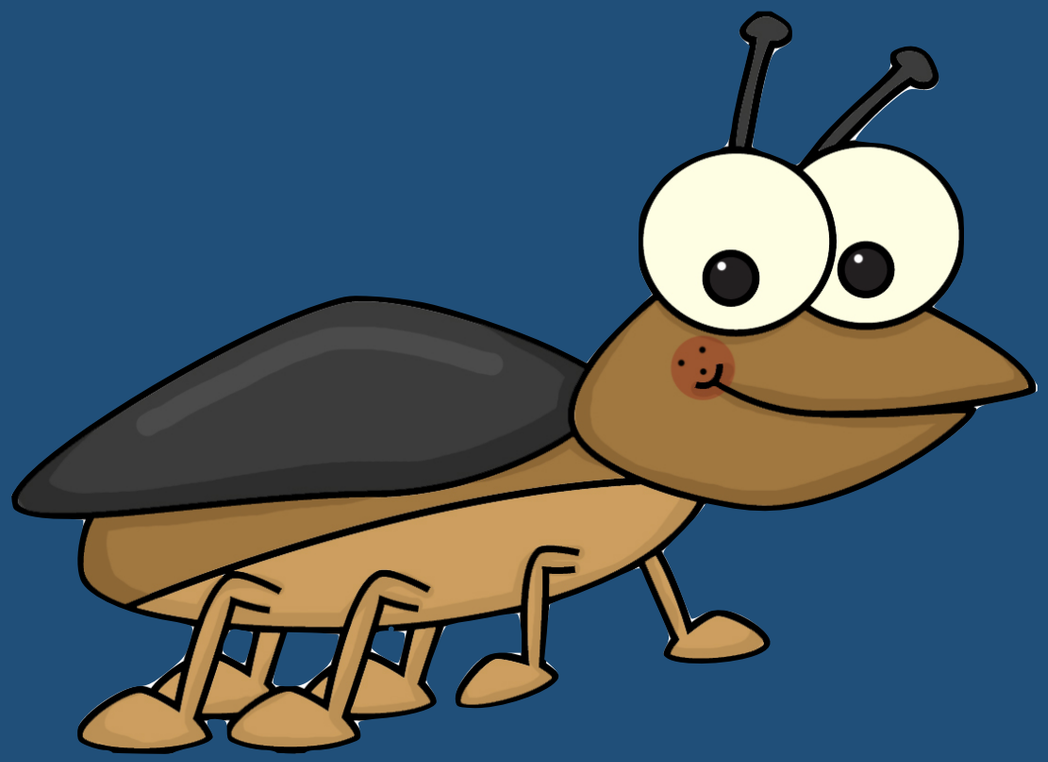
- Almost any variable type with almost any attributes that you like can be inside a derived type

- You can have derived types inside derived types

  - But you can only have a type inside itself if you give it the **POINTER** attribute (until F2008 when it can be **ALLOCATABLE**)

- You can have arrays, including allocatable arrays in derived types

- You can’t give elements inside a type the **TARGET** attribute (see later)



The End