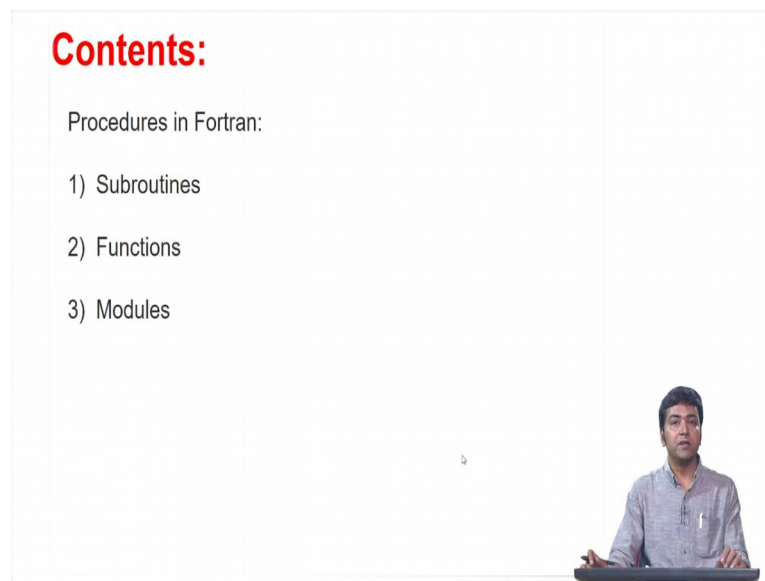


Computational Physics
Dr. Apratim Chatterji
Dr. Prasenjit Ghosh
Department of Physics
Indian Institute of Science Education and Research, Pune

Lecture - 05
Introduction To Fortran Part-3

So, welcome back to the third part of this module. So, in the previous 2 parts we saw how to write a Fortran program and what are the basic attributes of Fortran program. For example, we saw declaration of how variables are declared, how one does mathematical operations, some intrinsic functions in the Fortran which is inbuilt in the Fortran program itself, we also saw the use of do loops and if loops. So, in this part we will talk about 3 more things. So, these are primarily called procedures in Fortran.

(Refer Slide Time: 00:51)



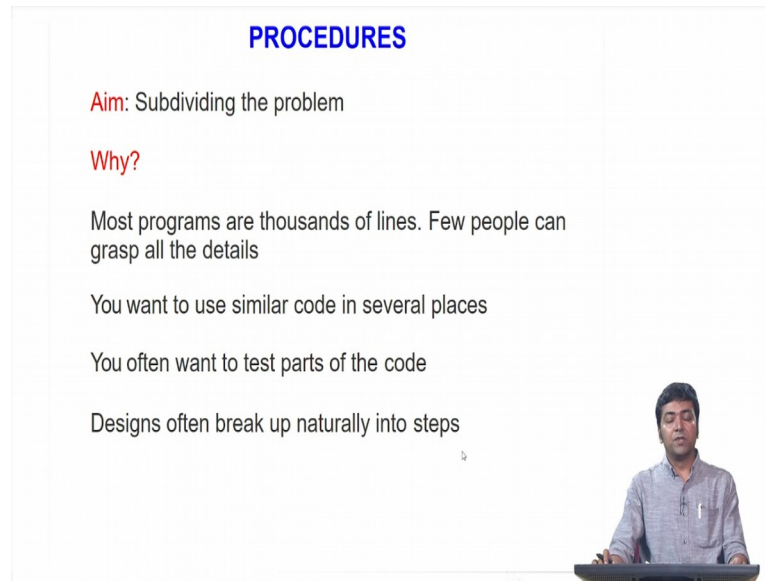
Contents:

Procedures in Fortran:

- 1) Subroutines
- 2) Functions
- 3) Modules

So, we will talk about subroutines functions and then modules.

(Refer Slide Time: 01:01)



PROCEDURES

Aim: Subdividing the problem

Why?

- Most programs are thousands of lines. Few people can grasp all the details
- You want to use similar code in several places
- You often want to test parts of the code
- Designs often break up naturally into steps

A person is visible in the bottom right corner of the slide, sitting at a desk with a keyboard.

So, what are procedures? So, the aim of the procedure is to subdivide a given problem. Now the question is why you need to subdivide a problem. So, typically when you write software or something using a Fortran code, so, you have the programs consists of several thousands of lines and often it happens that there are certain functions which the programs need to do repeatedly. So, instead of writing a few hundreds of lines for of code for those functions the idea here is that you write those lines of code once and you keep them in a separate place and then you call them as and when necessary in your program.

So, this helps us in the following ways. So, you can use the similar code in several places, you often want to test parts of the code, it will help you in that and the designs often break up naturally into steps.

(Refer Slide Time: 01:59)

SORTING REVISITED

```
PROGRAM sorting
INTEGER, DIMENSION (:), ALLOCATABLE::nos
INTEGER:: temp, i,j,n

! Reading in the size of the array nos
PRINT*, "Enter the size of the array"
READ*, n ! store it in variable n

! Allocate nos
ALLOCATE(nos(n))

! Reading in the set of n nos
PRINT*, "Type n nos, each no. on a line"
DO i=1,n
  READ*, nos(i)
END DO

! Sort the nos into ascending order of magnitudes
CALL sort(nos,n)

! Print the ordered nos
DO i=1,n
  PRINT*, "index", i, "value", nos(i)
END DO

deallocate(nos)
END PROGRAM sorting
```



So, let us look at this following this code which I left with you in the last lecture. So, it is the same sorting program to sort numbers in either in ascending or in descending order. So, we will what we will do now is we will see how we can convert this code into 2 parts; a main code and then what we called a subroutine which is a type of procedure in Fortran.

So, the important job which this code does is this part of the program where basically it is sorting the numbers into ascending orders of magnitude as per the example. So, what we will do is we will remove this part of the code from the main program because that is something which we want say for example, we want to do it repeatedly and put it in a subroutine. Now the way how to do that is the following.

(Refer Slide Time: 02:51)

```
SORTING REVISITED

PROGRAM sorting
INTEGER, DIMENSION (:), ALLOCATABLE::nos
INTEGER:: temp, i,j,n

! Reading in the size of the array nos
PRINT*, "Enter the size of the array"
READ*, n ! store it in variable n

! Allocate nos
ALLOCATE(nos(n))

! Reading in the set of n nos
PRINT*, "Type n nos, each no. on a line"
DO i=1,n
  READ*, nos(i)
END DO

! Sort the nos into ascending order of magnitudes
CALL sort(nos,n)

! Print the ordered nos

DO i=1,n
  PRINT*, "index", i, "value", nos(i)
END DO

deallocate(nos)
END PROGRAM sorting
```

So, this is where I have the program written again, the sorting program. So, as you can see that the first part, the array declarations they are almost similar compared to the previous program which we had here. Then we like the previous program here we are asking the user to provide the number of numbers which we want to give in the input, then we allocate that number to set the dimension of the array where we stored the input and then we read the numbers from the user, the user needs to type one number at a time. So, up till this point it is exactly same as what we had in the previous code that we this part.


So, now the change in this new code is that instead of having this part of the code this part. So, in the new code what we have done is we have removed that part of the code and we have introduced this new line. So, what it says is call space sort within bracket numbers comma n. So, what this line is doing is. So, what we have done is this part of the code we have put it in a separate sub code sort of thing which we are calling as a subroutine and the name of that subroutine is sort.

So, what we are doing here in this line the moment the controller reaches this particular line the program what the program does is it calls this subroutine using the call statement. So, once the program the subroutine is called the controller goes through this particular section of my program which contains the subroutine.

(Refer Slide Time: 04:35)

SORTING REVISITED

```
SUBROUTINE sort(array, length)
  IMPLICIT NONE
  INTEGER:: length,array(length), i,j, temp
  DO i=1,length-1
    DO j=i+1,length
      IF (array(i) > array(j)) THEN
        temp=array(j)
        array(j)=array(i)
        array(i)=temp
      END IF
    END DO
  END DO
END SUBROUTINE sort
```



So, the subroutine and the content of the subroutine this is exactly same as what we have been doing in the previous lecture in the same code.

And at the end of the day what the subroutine will do is it will return as me an array of numbers which has been arranged according to the ascending order. So, and then the rest again rest of this code is again same as the older version of the code which we saw in this previous slide here.

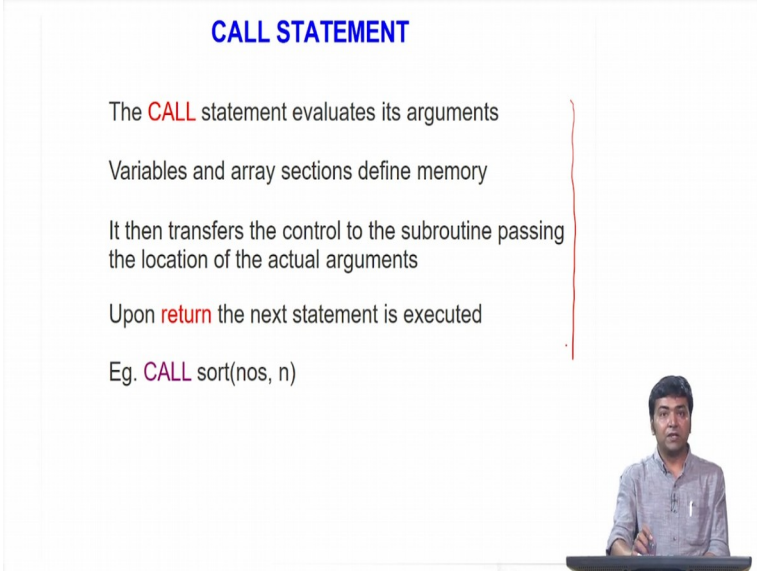
So, now what one should note here is that. So, these 2 variables here the numbers array and the n which tells us the size of the dimension of this array these variables are declared in the same as integers in the main program one is an array integer and the other is just a scalar number. So, when we go to the subroutine. So, in subroutine also has 2 variables here in the argument of the subroutine.

So, the point which I am trying to make here is that the argument in the subroutine statement in your subroutine should be same as the arguments in the main code. Not only that the type of the arguments, so that is their number the data type that is whether it is an integer or it is a real whether it is an array or it is a scalar quantity all those should match.

So, once the controller of the program reads reaches this part of the program and it executes this set of statements what it will do is it will in the same array variable it will return the control to the main program and the set of numbers which are now arranged

according to the ascending order will be stored in this numbers array and then we will print it out. So, this is an example of how we can split a program into a main program and a procedure called subroutine.

(Refer Slide Time: 06:47)



CALL STATEMENT

- The **CALL** statement evaluates its arguments
- Variables and array sections define memory
- It then transfers the control to the subroutine passing the location of the actual arguments
- Upon **return** the next statement is executed

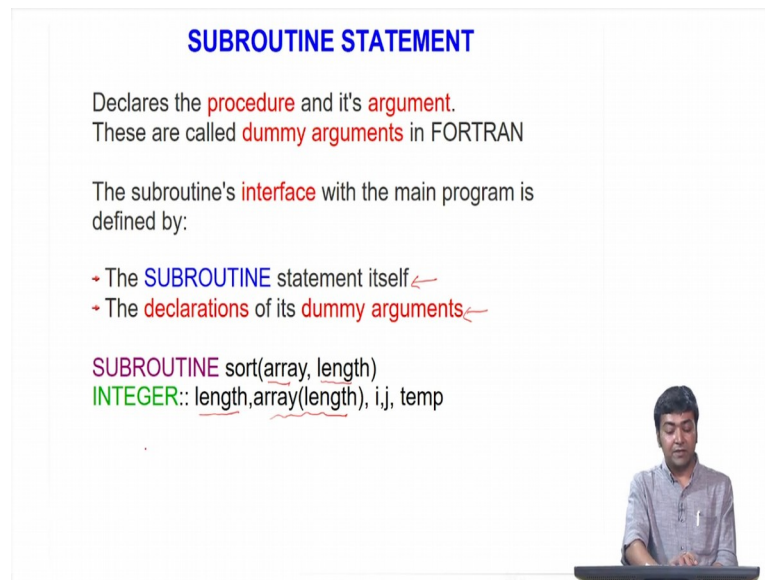
Eg. **CALL** sort(nos, n)

The slide features a small video inset in the bottom right corner showing a man with dark hair and a beard, wearing a light blue button-down shirt, sitting at a desk and speaking.

So, whenever we use a subroutine. So, what we need to use is the call statement. With the call statement evaluates the arguments of the subroutine. So, as I mentioned before that the variables and the array sections. So, if you look at it the subroutine is also like a program in itself it has an implicit none, it has a declaration part, it has a part which contains the job, it needs to do.

The only difference between a program main program on the subroutine is as you have notice the main program starts with the Fortran keyword program and it ends with the Fortran keyword end Fortran program. Instead a subroutine starts with the Fortran keyword subroutine and ends with the Fortran keyword end subroutine. The rest of it is exactly same as you have in the main program. So, this is what.... I mean these are the points which I have just discussed and I have just written listed down here in this particular slide.

(Refer Slide Time: 07:45)



SUBROUTINE STATEMENT

Declares the **procedure** and its **argument**.
These are called **dummy arguments** in FORTRAN

The subroutine's **interface** with the main program is defined by:

- The **SUBROUTINE** statement itself ←
- The **declarations** of its **dummy arguments** ←

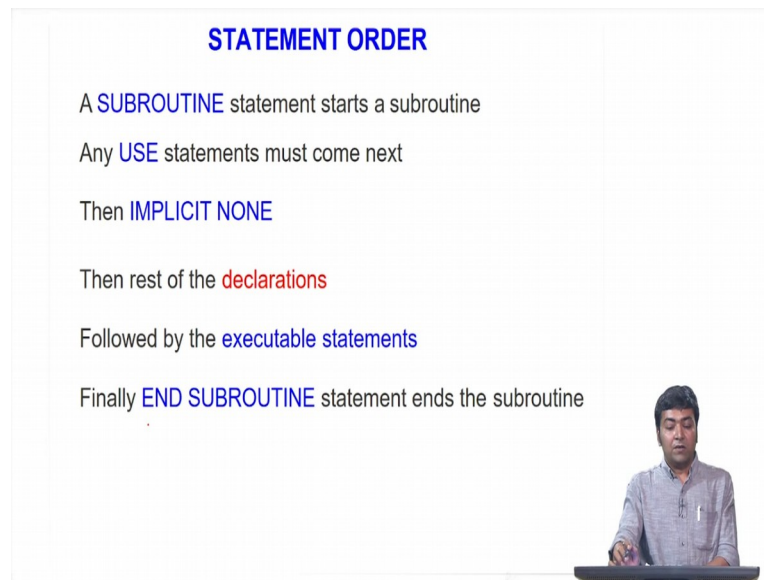
```
SUBROUTINE sort(array, length)
INTEGER:: length, array(length), i, j, temp
```

The slide also features a small video inset of a man sitting at a desk with a laptop, looking at the camera.

So, apart from the call statement as I mentioned before also there is something called a subroutine statement and the subroutine statements is used to decide the procedure and its argument. So, remember that these arguments are called dummy arguments in Fortran and then the subroutines interface is the main program it is defined by the subroutine statement itself and then the declaration of the arguments.

For example I have here, say a subroutine as we saw in the previous example which has 2 variables in array variable which is a dummy variable which is which we are declaring here as a array type and then a length variable which is a again a dummy variable which inside the subroutine we are declaring as an integer type.

(Refer Slide Time: 08:33)



STATEMENT ORDER

- A **SUBROUTINE** statement starts a subroutine
- Any **USE** statements must come next
- Then **IMPLICIT NONE**
- Then rest of the **declarations**
- Followed by the **executable statements**
- Finally **END SUBROUTINE** statement ends the subroutine


So, something to remember is that a subroutine statement starts a subroutine and often there is another keyword in Fortran which we will see slightly later this use statement. So, this use statement should come after the subroutine statement, after this subroutine statement and once you have use statement if necessary then you start writing the subroutine as a as you do your program.

So, basically you start with implicit none, then the rest of the declarations and the job you want to do that is the executable statements and then you finally, end with a end subroutine. So, this is how typical structure of a subroutine looks like.

(Refer Slide Time: 09:15)

DUMMY ARGUMENTS AND ARGUMENT MATCHING

- Their names exist only in the **procedure** ←
They are declared much like **local variables**
- Any **actual argument** names are irrelevant
- The **dummy arguments** are associated with the **actual arguments**
- Dummy & actual arguments** must match ←
The **no.** of arguments must be the same and the **type** of each argument must match
- The **size** of the **dummy array** must not exceed the size of the **actual array** argument



So, some more things to remember and it is very crucial is that the dummy arguments which we are using in the subroutine and may arguments which we are using to call the subroutine those 2 should match.

So, the dummy arguments I mean their names exist only in the procedure and they are declared as a local variable inside the subroutine. Any actual names are irrelevant and what the code does is this the code associates this dummy arguments with the actual arguments and as I mentioned before I am reiterating here again that the dummy arguments and the actual arguments must match. What I mean by that is that when you are calling the subroutine and when you are writing the subroutine.

So, in both the places the number of arguments which you have should be same and not only that the type of each argument must also match. The size of a dummy array must not exceed the size of the actual argument. This is a very important thing to remember because if this matching does not occur then your compiler will complain and the code will not compile.

(Refer Slide Time: 10:29)

EXAMPLE I

Let's assume a subroutine with an interface like:


```
SUBROUTINE normal(array,size)
INTEGER:: size
REAL, DIMENSION(size):: array
```

array and size are dummy arguments

The following calls are correct:

```
REAL, DIMENSION(1:10):: data
CALL normal(data,10)
CALL normal(data(2:5), SIZE(data(2:5)))
```

data, 10 and SIZE(data(2:5)) are actual arguments



So, let us take some examples here. So, suppose I have this particular subroutine. So, where subroutine called normal, then I have 2 arguments for this subroutine array and size where inside the subroutine I have declared size as a scalar integer quantity and an array as a real array whose dimension is determined by the value of the size. So, note here this array and sizes are these 2 are dummy arguments. Now let us see which of these how these are called in the main program.

So, let us take this example. So, for example, in the main program I declare variable one array variable which I am calling as data and I declare it as a real type and it has a dimension whose range varies from 1 to 10. So, if I call this subroutine normal in my main program and then I put the first argument as data and then the second argument as 10, so, this is a correct call, the reason this is correct call is because here if you look at the first argument and compare that with the dummy argument in the subroutine.

So, both are real type both are arrays and one dimensional, the size of the array as I determined in the main program. So, it has 10 elements in it and the in the subroutine this dummy argument size is what determines the size of this array. So, that I have associated as 10. So, very everything is matching it in the way I am calling this particular subroutine inside my main program.


(Refer Slide Time: 12:21)

EXAMPLE II

```
SUBROUTINE normal(array,size)
INTEGER:: size
REAL, DIMENSION(size):: array
```

The following calls are incorrect:

```
INTEGER, DIMENSION(1:10):: indices
REAL:: var, data(10)
⇒ CALL normal(indices, 10)
CALL normal(var,1)
CALL normal(data, 10.0)
CALL normal(data, 20)
```



Similarly, similar is the case also for the second case. Now let us look into some more examples and try to see whether these following calls to the subroutine are correct or not. So, basically I have the same subroutine as in the previous slide and now I have 3 types of variables here in my main program. One is the indices which I am declaring as an integer type of variable which has a dimension using these as range from 1 to 10 then I have 2 real variables one is a scalar quantity and another is an array whose dimension is given by 10.

Now, let us look into the first column. So, the question is. So, if I call this normal subroutine named normal in this way this particular call statement I am referring to now. So, is this a correct call or not. So, let us check the arguments first. So, here in the main program I have the first argument as indices which I have declared as a real as an integer in the main program; however, if we look at my subroutine the first argument array is declared as an integer as a real type. So, hence this there is a mismatch between this the first the data type of the first argument.

(Refer Slide Time: 13:37)


EXAMPLE II

```
SUBROUTINE normal(array,size)
INTEGER:: size
REAL, DIMENSION(size):: array
```

The following calls are incorrect:

```
INTEGER, DIMENSION(1:10):: indices
REAL:: var, data(10)
```

```
CALL normal(indices, 10)
! wrong base type
CALL normal(var,1)
! not an array
CALL normal(data, 10.0)
! wrong type
→ CALL normal(data, 20) X
! dummy array too big
```



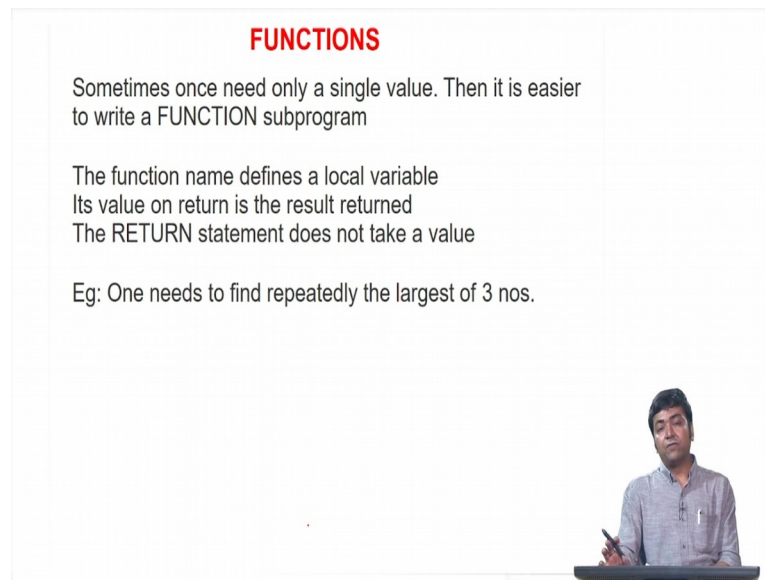
And the data type of the first argument in the subroutine and hence this is a wrong call. Similarly, if we go to the second example here that is if I call it in this fashion. So, then this is also a wrong call because I have now the variable the first argument in the main program of when I am calling this subroutine var this I have declared as a scalar quantity while in the main subroutine I have it as a array type.

So, this is also a wrong call. Now if I look at the third example here. So, in this case in the first argument I have a there is a match of the data type that is here also I have real variable and an array and in the subroutine also I have a real variable in the array, but let us look what happens to the second argument.

The second argument I am writing when I am in the main program when I am calling the subroutine as 10.0 which is a real number while in the subroutine I have declared the second argument as an integer. So, hence this there is a mismatch again of the data type here and this is also wrong.

In the similar way the first the final example this last one here that is also a wrong because here the size of this data array which I have declared to be 10 here in the main program does not match with the second argument which in the subroutine controls the size of this particular array. So, this is too big. So, hence this is also wrong.

(Refer Slide Time: 15:21)



FUNCTIONS

Sometimes once need only a single value. Then it is easier to write a FUNCTION subprogram

The function name defines a local variable
Its value on return is the result returned
The RETURN statement does not take a value

Eg: One needs to find repeatedly the largest of 3 nos.

The slide also features a small video inset in the bottom right corner showing a man in a light blue shirt speaking and gesturing with his hand.

So, this is what basically the idea of subroutines. In subroutines what you can do is you can send a single data, an array of data and you can get a single number or an array of numbers or several arrays of numbers, but sometimes what you might be interested in just getting a single value. So, and under that circumstances it is often easier to write a function sub program. So, function is another procedure for which are is available in Fortran the function name the name you give to the function it defines a local variable and when you call it in the main program the value on return is the result which is returned.

Suppose for example, if we write want to write a function which needs to find repeatedly the largest of 3 given numbers. So, what I mean by that is suppose the user gives me 3 numbers and I want to find write a code which will help me to find the largest of them amongst those 3 numbers and I want to use that repeatedly in several programs. So, basically what my code will return at the end of the day is just one single number. So, instead of using a subroutine what I can do is I can use something called the function.

(Refer Slide Time: 16:35)


EXAMPLE I

```
→ PROGRAM x
→ IMPLICIT NONE
→ INTEGER:: try1, try2, try3, total, count1, largest
total=0
count1=0

DO
PRINT*, 'Type three trial values' ←
READ*, try1, try2, try3 ←
IF (MIN(try1, try2, try3) < 0) EXIT →
count1=count1+1 ←
total=total+largest(try1, try2, try3)
END DO

→ PRINT*, 'No. of trial sets=', count1, 'total of best of 3=', total

END PROGRAM x
```



So, this is how typical program look like. So, basically, so, I have I start with say a program name then I give the implicit none then I declare my variables that these are, so, I am assuming that I get all less integer numbers and then I have some dummy variables which I am going to use in the program and then this is the part where I am executing the job of finding out the largest number and then at the end of the day I print out that how many times I need to try and then the largest of the 3 that is the total. So, this is the I mean this is how the program would have looked like when I would have used it as a simple program.

So, now what I will do is I will replace this program in and I will try to find write replace it with a function. So, what I have here in this program is basically I have this variable total and then this is my function which I am using to compute the largest of the 3 numbers try1 try2 try3. So, what I have here is in this program. So, the first step the controller enters this do loop it asks the reader the user to print 3 trial values. These are read in here then I am checking that whether these are negative numbers.

The reason I am checking whether these are negative numbers is because the way I have written the program is such that it can only work for positive values. If it is a negative number it will come out of this do loop using this exit statement and then I increase the counter. So, this gives me how many times I need to do this operation and then I here I print out the total and then I call this function here.

So, this largest here although I have declared it as a type of a variable integer type of variable.

(Refer Slide Time: 18:37)


EXAMPLE I

```
→ PROGRAM x
→ IMPLICIT NONE
→ INTEGER:: try1, try2, try3, total, count1, largest
total=0
count1=0

DO
PRINT*, 'Type three trial values' ←
READ*, try1, try2, try3 ←
IF (MIN(try1, try2, try3) < 0) EXIT →
count1=count1+1 ←
total=total+largest(try1, try2, try3)
END DO

PRINT*, 'No. of trial sets=', count1, 'total of best of 3=', total

END PROGRAM x
```



Actually it contains a set of instructions which is given by the function. So, this is how one writes function in a Fortran module. So, for example so a function procedure starts with the name function and then similar to the subroutine it also ends with the name end function. So, here I have named the function as largest and then there are 3 arguments. So, these again note that the number of arguments when you are calling the function in your main program and the number of arguments in the procedure function they should match and their data type should also match as similar to that in the subroutine.


So, here I have 3 arguments here first second and third. So, all of them I have declared as a integer and then after finding out the largest value of these 3 numbers integer numbers which is given by the user I stored them in this variable called largest. So, this is the same name as the function. So, when I call this function here. So, it is basically used as a simple variable type and it will return me just a single number.

(Refer Slide Time: 19:49)

EXAMPLE II

```
PROGRAM norm
  IMPLICIT NONE
  REAL::a,x,y,z,mynorm
  PRINT*, 'Enter the 3 components'
  READ*, x,y,z
  a=mynorm(x,y,z)
  PRINT*, a
END PROGRAM norm

REAL FUNCTION mynorm(xv,yv,zv) RESULT(res)
  IMPLICIT NONE
  REAL, INTENT(IN):: xv,yv,zv
  REAL:: a
  a=xv**2+yv**2+zv**2
  res=SQRT(a)
END FUNCTION mynorm
```



So, here I have another example. So, for example, this is an example to write a function which I am using to calculate the norm of a number sorry or a norm of a vector. Say suppose I the user gives me 3 components of a vector here x y z I want to compute it and want to know what its norm is I use this function called mynorm here.

So, another thing. So, the way this function is written is slightly different from the way this one is written. So, here if you look at, so, I am not declaring in the first line what type of function it is that is what type of number it will return whether it will be return me a real number or it will return me a imaginary number or it will return me an integer. So, that is done in the declaration part of this procedure.

In contrast here I can what I can also do is that I can declare the function as a data type. So, for example, here I have declared that this function which is my norm which will give me a number. So, that function is a real type of function and then I have 3 arguments for the functions and these match with the 3 arguments where the function is called in the data type in the main program here and I have declared the these 3 arguments as integer type in my function procedure here sorry not integer it is real type in my function procedure here and there is another new statement here which I have which is called intent in bracket in.

So, what this does is this tells the function what is the nature of these arguments that is are these inputs or are these outputs. So, the moment you set these 3 arguments as intent

in type what this part of the code will understand is that the inputs which are supplied to this particular function they are coming from these through these 3 variables which we have here in the argument and inside the program when you try to reassign some other values in these variables you will not be able to do that.

So, you can supply input to your function through this 'intent in' type of variables, but you cannot reassign them anywhere and then what it is doing is it is evaluating it and storing this in the result in this variable. So, when I am calling this function here in the main program it will return me the number.

(Refer Slide Time: 22:31)

HOW TO COMPILE PROCEDURES

Procedures may be in the same file as the one containing the Main program or they can be in different files.


If they are in the **same file** as the main program, then they can be written either before or after the main program. However, the convention is to write after the main program.

To compile:

f95 -o prog.x prog.f OR f95 prog.f ←

If they are in **different files** eg. prog.f & func.f, then to compile:

f95 -o prog.x prog.f func.f or f95 prog.f func.f
OR
f95 -o prog.x func.f prog.f or f95 func.f prog.f
The order of the file names is not important.



So, now the question is sometimes you have these functions or the subroutines in the same file as your main program, but often you have them in different files. So, in that case how do you compile them.

So, for example, suppose if you if you if you have the function or the subroutine written in the main in the same file as that in the main program then you compile it in the same way as we will compile a normal code and note one thing that in that case your subroutine or the function can either be a written before or after the main program the order does not matter; however, typically one writes it after the main at the end of the main program.

The second question is if these are in different files then how does one compile it. So, the way to compile it is that you give the name of the compiler space minus o space the name of the file where the executable will be stored, here in this case it is prog dot x then the name of the main program the file containing the main program then the name of the file containing the function or if you do not want to if you just want to use the default a dot out as your executable then you can just write it in this way, but this order whether this program name, main program name should come first or the function name should come first this does not matter.

So, for example, here in this second option here. So, I have swapped the name of the function and the program. So, this was earlier here and this was here, but now I have brought the function the name of the file containing the function in the beginning that is before the name of the file containing the program. So, come for compilation of subroutines and functions which are written in different files the order is not important.

(Refer Slide Time: 24:41)

INTENT STATEMENT

It is possible to make arguments read-only

```


SUBROUTINE normal(array, size)
IMPLICIT NONE
INTEGER, INTENT(IN):: size
REAL, DIMENSION(size):: array
    
```

This prevents to write to variable size accidentally or calling another procedure that does it

One can make arguments write-only

```

SUBROUTINE init(array, value)
IMPLICIT NONE
->REAL, DIMENSION(:), INTENT(OUT)::array
REAL, INTENT(IN):: value
array=value
END SUBROUTINE init
    
```



So, as I was telling you that this 'intent in' statements. So, typically what these statements do is that it makes the arguments either read only or it makes the arguments write only. So, as we have said say for example, here this second argument here the size if I declare it as an intent in. So, the code can only read the value inside that is stored in this size it cannot reassign the value. In the same way if I have a intent out statement or if I declare a datatype and a variable as intent out so, as for example, this array argument

here. So, what it means is that the if I want to pass some value or pass some information to the subroutine through this argument I would not be able to do. So, this argument this variable will be used only to assign new value inside the subroutine.


(Refer Slide Time: 25:49)

INTENT (INOUT)

One can use a variable both for **read** & **write** purpose:

```
SUBROUTINE sort(array, length)

IMPLICIT NONE
INTEGER, INTENT(IN):: length
INTEGER, DIMENSION(length), INTENT(INOUT):: array
```



And apart from this intent in and intent out statement there is also another statement which is called the 'intent in out'. So, from the name it is already quite clear that what is the purpose of this. The purpose of this is that you can use the same variable for both read and write purpose. So, basically you send your input to the subroutine through that variable and then the output also goes out through that variable .

(Refer Slide Time: 26:11)

MODULES

A module may contain:

- (1) variable declaration ←
- (2) functions and subroutines ←

It need not be in the same file as the main program

If it is in the same file as the main program then it must precede it

A module is invoked by “USE.<name>” at the beginning of the main program or another module

Apart from these 2 procedures that is the subroutines and my functions there is another procedure which is called modules. So, what information does the module contain? So, module typically contains these following 2 types of information. So, one is it contains a list of variable declarations now why do you need to have that. So, often it happens that you mightyou need to use the same variable in thousands of different places.

So, what it means is that if you write a program from scratch say you write thousands of programs where you need to use the same variable you need to declare it at each and every one of the of your programs. Rather than doing that what you can do is you can put this information into a file called the module file.

So, basically in this file you declare that my variable x belongs to this data type and then you call this module file in the other programs and then you can use the same variables which are listed here without further declaring them in the program which where you are using. The same thing also applies for functions and subroutines.

So, instead of having separate files for functions and subroutines you can also list them in this module file and another thing to note that the module file need not be the same file as the main program and the most importantly if you have or if you want to have the module file in the same program then it should be at the beginning of the before your main program starts and a module file module is typically used by this Fortran keyword


called use at the beginning of the main program or you can have also for example, modules in between modules.

(Refer Slide Time: 28:07)

EXAMPLE OF MODULE: FUNCTION

```
MODULE norm_mode
  IMPLICIT NONE
CONTAINS
  REAL FUNCTION mynorm(xv,yv,zv) RESULT(res)
    REAL, INTENT(IN):: xv,yv,zv
    REAL:: a
    a=xv**2+yv**2+zv**2
    res=SQRT(a)
  END FUNCTION mynorm
END MODULE norm_mode

PROGRAM norm
  USE norm_mode
  IMPLICIT NONE
  REAL::a,x,y,z
  PRINT*, 'Enter the 3 components'
  READ*, x,y,z
  a=mynorm(x,y,z)
  PRINT*, a
END PROGRAM norm
```



So, for example, here is example of a module. So, what I have done is. So, it is basically the same program which is used to calculate the norm of a vector using the function mynorm. Now in the previous case what we did is we had the main program and a function. Instead of that what I have done here is that I put the function inside a module. So, like functions and subroutines. So, a module starts with the Fortran keyword module then the name of the module and also it ends with the Fortran keyword end module followed by the name of the module.

So, and then the module part as I mentioned has 2 parts; the first part is it contains it might contain lists of different types of variables. So, the moment we declare variable in Fortran. So, we need to have this implicit none statement as I mentioned in the early lectures. So, that is why in this module I have this implicit none statement, but in this case is for this particular case I do not have any variables to declare. So, after that there is nothing written here and then it also contains as I mentioned in the previous slide the second thing which a module file contains is the different subroutines and the functions.

So, that is mentioned, but before mentioning that or before writing the subroutines and functions in the module file you need to use the contains statement. This tells the control when the controller comes to this module file this tells the list of things or the list of

subroutines and functions that are contained in the module file and then what you do is you have your function , this function is written in the same way as we had done in the earlier example. So, instead of one functions you can have n number of functions in the module file.


(Refer Slide Time: 30:15)

EXAMPLE OF MODULE: SUBROUTINE

```
MODULE sort_num
  IMPLICIT NONE
CONTAINS
  SUBROUTINE sort(array, length)
    IMPLICIT NONE
    INTEGER:: length,array(length), i,j, temp

    DO i=1,length-1
      DO j=i+1,length
        IF (array(i) > array(j)) THEN
          temp=array(j)
          array(j)=array(i)
          array(i)=temp
        END IF
      END DO
    END DO

  END SUBROUTINE sort
END MODULE sort_num
```



So, here there is another example of the module file. So, where basically I have.... So, this is the sorting program which we looked earlier. So, what I have done here is now I have 2 different files on a module file and a main program. So, this particular file printed in this slide contains the module. So, as mentioned before. So, it has the module then the module name then implicit none then it contains the subroutine and end subroutine sort and end module 'sort num'. So, this is one file and then I have my main program.

(Refer Slide Time: 30:47)

EXAMPLE OF MODULE: SUBROUTINE

```
PROGRAM sorting
USE sort_num
IMPLICIT NONE
INTEGER, DIMENSION (:), ALLOCATABLE::nos
INTEGER:: temp, i,j,n

! Reading in the size of the array nos
PRINT*, "Enter the size of the array"
READ*, n ! store it in variable n


! Allocate nos
ALLOCATE(nos(n))

! Reading in the set of n nos
PRINT*, "Type n nos, each no. on a line"
DO i=1,n
  READ*, nos(i)
END DO

! Sort the nos into ascending order of magnitudes
CALL sort(nos,n)

! Print the ordered nos
DO i=1,n
  PRINT*, "index", i, "value", nos(i)
END DO

deallocate(nos)
END PROGRAM sorting
```



So, now how do I call a module in the main program? So, in the main program if one wants to use a module, so, what one needs to do is so, this is my declaration part of the main program. So, I will start the declaration from here. So, if you look carefully that just before the declaration part before I write this implicit none statement I have used this use statement and then the name of the module use 'sort underscore num'.

So, this sort underscore num is nothing, but the module name which we see here and then the rest of the thing is the same way as we had done before. So, here we are calling the subroutine which is present in this particular module.


(Refer Slide Time: 31:33)

GENERAL STRUCTURE OF A MODULE

```
MODULE modulename
  → USE a
  → USE b
  .....
  IMPLICIT NONE
  <variables>
  .....
CONTAINS
  FUNCTION x(.....)
  END FUNCTION x

  SUBROUTINE z(.....)
  END SUBROUTINE

  .....
END MODULE modulename
```



So, this is a general structure of a module. So, again though it is a repetition, but I think it is better to repeat it once again for the sake of recapitulation. So, module file a module starts with the Fortran keyword module followed by a module name then within the module you can use other modules if you want to do so, then you can use this use statements you can use n number of modules say for example, use module 'a' here use module 'b' here and then you can declare different variables here.

So, starting with implicit none and then the variable declarations then you can contain several functions here and subroutines and that is and before you start putting your functions and subroutines you need to use the 'contains' statement and then finally, you end your module file with the end module followed by the module name. So, this is the general structure of a module.

(Refer Slide Time: 32:25)

USE OF MULTIPLE SOURCE FILES

Suppose the module & the main program are in separate files:


Eg: main.f (file containing the main program)
norm_module.f (file containing the module)

How to compile & create an executable

```
f95 -o main.exe norm_module.f main.f
```

Remember: the module source should be before the main source on the command line

A file norm_module.mod is produced in addition to main.exe



Now, in case of a when you use a module file where module in 2 in a file which is different from the main file. Say for example, I have my main program in this file called 'main dot f' and then I have my module in this file called 'norm underscore module dot f' the question is how will we compile them.

So, the compilation is done in the same way as you would have compiled your subroutine when your subroutine or the function are in 2 different are different from that of the main thing, but one thing so one should be careful of that the name of the module file should always precede the name of your main program.

So, of this you should be very careful. So, this is the primary difference between compiling subroutine in a different file along with the main program and compiling a module contained in a different file along with the main program.

So, in subroutine the ordering of the names of the files are not important, but when you do it for a module then the module name the name of the file containing the module must always come before the main program and so in this process in the compiling process what the compiler does is it produces a file called 'norm underscore module dot mod' in addition to the main executable file.


(Refer Slide Time: 34:03)

FUNCTION vs. SUBROUTINE

Both have arguments, which can be real, integer, etc. and also arrays.

- ✓FUNCTION is used like a variable in the main program.
- ✓Hence it should be assigned a type
- ✓It returns only a single value.
- ✓The output of a function is stored in the name of the function.

- SUBROUTINE is called in the main program using the CALL statement.
- It does not need to be assigned a type. ←
- Can return many values which can be stored in single variables or arrays.
- The output is stored in one of the arguments.



So in the last part of this or in the last slide what I am going to do is I am just going to sort of tell you or summarize what are the primary difference between a function and a subroutine. So, the common thing that both functions and subroutines have arguments which can be of any data type it can be real, it can be integer, it can be arrays.

So, function is typically used like a variable in the main program. In contrast a subroutine is called in the main program and since this function is used to the variable. So, it should be assigned a datatype and the most important thing to remember is that the main difference between a function and the subroutine is that the function returns only a single value while in the subroutine it can return several values.

Another thing is in the subroutine you need not assign a datatype to a particular subroutine. So, to summarize this whole module so, what we have done so far was to give you rapid idea in a very fast way to how to use Fortran language to do your programs. So, again I would like to remind you that this is not a course on Fortran. So, this is more of a crash course on Fortran where what I have done through these 3 sets of lectures is that I have listed down the most important things which we will need to get started with your program.

So, we started off with looking into the general Fortran program structure. We saw how different variables are assigned to different data types. We saw the intrinsic use of the intrinsic functions in Fortran, how mathematical operations are performed in Fortran,

how to use DO loops which will allow you to do the same task repeatedly, how to use conditional statements, how to use arrays and then finally, in this part how to use Fortran procedures namely functions subroutines and modules so

Thank you.