A Fortran 2003 introduction by examples

Gunnar Wollan

2012

1 Introduction

The purpose of this book is to give a good insight in the Fortran 2003 programming language.by going through a number of examples showing how computational problems and the reading and writing data to files can be solved.

2 Why use Fortran?

In the last 15 to 20 years Fortran has been looked upon as an old-fashioned unstructured programming language by researchers and students in the field of Informatics. Fortran has lacked most of the features found in modern programming languages like C++, Java etc. Especially the lack of object orientation has been the main drawback of Fortran. This is no longer true. Fortran 2003 has all the modern features including *OOP* (Object Oriented Programming).

The reason we still use Fortran as a programming language is because of the exeecution **speed** of the program. In the field of natural sciences, computer simulations of natural phenomena are becoming increasingly more important. Laboratory experiments are getting too complex and too costly to be performed. The only alternative is to use a computer simulation to try to solve the problem under study. Thus the need to have your code execute faster becomes more and more important when the simulations grows larger and larger. In number-crunching Fortran still has an edge in speed over C and C++. Tests has shown that an optimized Fortran program in some cases runs up to 30 percent faster than the equivalent C or C++ program. For programs with a runtime of weeks even a small increase in speed will reduce the overall time it takes to solve a problem.

2.1 Historical background

Seen in a historical perspective Fortran is an old programming language. 1954 John Backus and his team at IBM begin developing the scientific programming language Fortran. It was first introduced in 1957 for a limited set of computer architectures. In a short time the language spread to other architectures and has since been the most widely used programming language for solving numerical problems.

The name Fortran is derived from **For**mula **Tran**slation and it is still the language of choice for fast numerical computations. A couple of years later in 1959 a new version, Fortran II was introduced. This version was more advanced and among the new features was the ability to use complex numbers and splitting a program into various subroutines. In the following years Fortran was further developed to become a programming language that was fairly easy to understand and well adapted to solve numerical problems.

In 1962 a new version called Fortran IV emerged. This version had among it's features the ability to read and write direct access files and also had a new data-type called LOGICAL. This was a Boolean data-type with two states **true** or **false**. At the end of the seventies Fortran 77 was introduced. This version contained better loop and test structures. In 1992 Fortran 90 was formally introduced as an ANSI/ISO standard. This version of Fortran has made the language into a modern programming language. Fortran 95 is a small extension of Fortran 90. These latest versions of Fortran has many of the features we expect from a modern programming languages. Now we have the Fortran 2003 which incorporates object-oriented programming with type extension and inheritance, polymorphism, dynamic type allocation and type-bound procedures.

3 The Fortran syntax

As in other programming languages Fortran has it's own syntax. We shall now take a look at the Fortran 2003 syntax and also that of the Fortran 77 syntax.

4 The structure of Fortran

To start programming in Fortran a knowledge of the syntax of the language is necessary. Therefore let us start immediately to see how the Fortran syntax look like.

Fortran has, as other programming languages, a division of the code into variable declarations and instructions for manipulating the contents of the variables.

An important difference between Fortran 77 and Fortran 2003 is the way the code is written. In Fortran 77 the code is written in fixed format where each line of code is divided into 80 columns and each column has its own meaning. This division of the code lines into columns has an historically background. In the 1960s and part of the 1970s the standard media for data input was the punched cards.

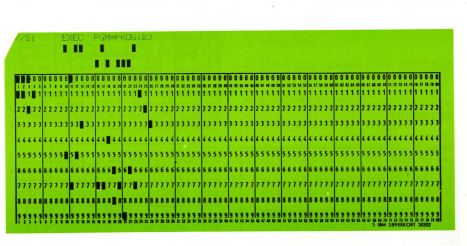

Figure 1: A punched card

They were divided into 80 columns and it was therefore naturally to set the length of each line of code to 80 characters. In table 1 an overview of the subdivision of the line of code is given.

Column number	Meaning
1	A character here means the line is a comment
2 - 5	Jump address and format number
6	A character here is a continuation from previous line
7 - 72	Program code
73 - 80	Comment

Table 1: F77 fixed format

Fortran 77 is a subset of Fortran 2003 and all programs written in Fortran 77

can be compiled using a Fortran 2003 compiler. In addition to the fixed code format from Fortran 77, Fortran 2003 also supports free format coding. This means that the division into columns are no longer necessary and the program code can be written in a more structured way which makes it more readable and easier to maintain. Today the free format is the default settings for the Fortran 2003 compiler.

4.1 Datatypes in Fortran

Traditionally Fortran has had four basic datatypes. These were INTEGER and REAL numbers, LOGICAL which is a boolean type and CHARACTER which represent the alphabet and other special non numeric types. Later the REAL data type was split into the REAL and COMPLEX data type. In addition to this a derived datatype can be used in Fortran 2003. A derived datatype can contain one or more of the basic datatypes, other derived datatypes and in addition procedures which is a part of the new OOP (object Orientee Programming) features in Fortran 2003.

4.1.1 INTEGER

An INTEGER datatype is identified with the reserved word INTEGER. It has a valid range which varies with the way it is declared and the architecture of the computer it is compiled on. When nothing else is given an INTEGER has a length of 32 bits on a typical workstation and can have a value from $[-2^{31}]$ to $[2^{30}]$ and a 64bit INTEGER with a minimum value from $[-2^{63}]$ to a maximum value of $[2^{62}]$.

4.1.2 REAL

In the same manner a REAL number can be specified with various ranges and accuracies. A REAL number is identified with the reserved word REAL and can be declared with single or double precision. In table 2 the number of bits and minimum and maximum values are given.

Precision	Sign	Exponent	Significand	Max. value	Min. value
Single	1	8	23	2^{128}	2^{-126}
Double	1	11	52	2^{1024}	2^{-1022}

Table 2: REAL numbers

A double precision real number are declared using the reserved words DOUBLE PRECISION or REAL(KIND=8) this last is now used as the preferred declaration of a double precision real number.

An extension of REAL numbers are COMPLEX numbers with their real and imaginary parts. A COMPLEX number is identified with the reserved word COMPLEX. The real part can be extracted by the function REAL() and the imaginary

part with the function AIMAG(). There is no need for writing explicit calculation functions for COMPLEX numbers like one has to do in C / C++ which lacks the COMPLEX data type.

4.1.3 LOGICAL

The Boolean datatype is identified by the reserved word LOGICAL and has only two values true or false. These values are identified with .TRUE. or .FALSE. and it is important to notice that the *point* at the beginning and end of the declaration is a necessary part of the syntax. To omit one or more points will give a compilation error.

4.1.4 CHARACTER

The CHARACTER datatype is identified by the reserved word CHARACTER and contains letters and characters in order to represent data in a readable form. Legal characters are among others *a* to *z*, *A* to *Z* and some special characters +, -, *, / and =.

4.1.5 Derived datatypes

These are datatypes which are defined for special purposes. A derived datatype is put together of components from one or more of the four basic datatypes and also of other derived datatypes. A derived datatype is always identified by the reserved word *TYPE name* as prefix and *END TYPE name* as postfix.

4.2 Declaration of variables

In Fortran there are two ways to declare a variable. The first is called implicit declaration and is inherited from the earliest versions of Fortran. Implicit declaration means that a variable is declared when needed by giving it a value anywhere in the source code. The datatype is determined by the first letter in the variable name. An INTEGER is recognized by starting with the letters *I to N* and a REAL variable by the rest of the alphabet. It is important to notice that no special characters are allowed in a variable name only the letters *A - Z*, the numbers *0 - 9* and the underscore character _. A variable cannot start with a number. In addition a LOGICAL variable is, in most compilers, identified by the letter *L*.

The other way of declaring a variable is by explicit declaration. This is in accordance with other programming languages where all variables has to be declared within a block of code before any instructions occurs.

As a general rule an implicit declaration is not a good way to program. It gives a code that is not very readable and also it is easily introduce errors in a program due to typing errors. Therefore always use explicit declaration of variables. To be certain that all variables has to be declared all programs, functions and subroutines should have as the second line in the declaration the keywords *IMPLICIT NONE*.

This tells the compiler to check that all variables has been declared. Some variables must always be declared. These are arrays in one or more dimensions and character strings.

4.2.1 Declaration of INTEGERS

First an example of how to declare an INTEGER in Fortran 95.

```
INTEGER                 :: i  ! Declaration of an INTEGER
                              ! length (32 bit)
INTEGER(KIND=2)         :: j  ! Declaration of an INTEGER (16 bit)
INTEGER(KIND=4)         :: k  ! Declaration of an INTEGER (32 bit)
INTEGER(KIND=8)         :: m  ! Declaration of an INTEGER (64 bit)
INTEGER,DIMENSION(100) :: n  ! Declaration of an INTEGER array
                              ! (100 elements)
```

As seen in the preceding examples there are certain differences in the Fortran 77 and the Fortran 95 way of declaring variables. It is less to write when the variables are declared in the Fortran 77 style but this is offset by greater readability in the Fortran 95 style. One thing to note is that in Fortran 95 a comment can start anywhere on the code line and is always preceded by an exclamation point.

4.2.2 Declaration of REAL numbers

The REAL datatype is now in most compilers confirming to the IEEE standard for floating point numbers. Declarations of single and double precision is declared like in the next example.

```
REAL                    :: x  ! Declaration of REAL
                              ! default length (32 bit)
REAL(KIND=8)            :: y  ! Declaration of REAL
                              ! double precision (64 bit)
REAL, DIMENSION(200)    :: z  ! Declaration of REAL  array
                              ! (200 elements)
```

4.2.3 Declaration of COMPLEX numbers

Fortran has, unlike C/C++, an intrinsic datatype of complex numbers. Declaration of complex variables in Fortran are shown here.

```
COMPLEX                 :: a  ! Complex number
COMPLEX, DIMENSION(100) :: b  ! Array of complex numbers
                              ! (100 elements)
```

4.2.4 Declaration of LOGICAL variables

Unlike INTEGERS and REAL numbers a LOGICAL variable has only two values, .TRUE. or .FALSE. and therefore only uses a minimum of space. The number of bits a LOGICAL variable is using depends on the architecture and the compiler. It is possible to declare a single LOGICAL variable or an array of them. The following examples shows a Fortran 77 and a Fortran 95 declaration. In other programming languages the LOGICAL variable is often called a BOOLEAN variable after Boole the mathematician.

```
LOGICAL                 :: l1  ! Single LOGICAL variable
LOGICAL, DIMENSION(100) :: l2  ! Array of LOGICAL variables
                               ! (100 elements)
```

4.2.5 Declaration of characters

Characters can either be declared as a single CHARACTER variable, a string of characters or an array of single characters or character strings.

```
CHARACTER                          :: c1  ! Single character
CHARACTER (LEN=80)                 :: c2  ! String of characters
CHARACTER, DIMENSION(10)           :: c3  ! Array of single
                                          ! characters
CHARACTER (LEN=80), DIMENSION(10)  :: c4  ! Array of character
                                          ! strings (10 elements)
```

4.2.6 Declaration of derived datatypes

The Fortran 95 syntax for the declaration of a derived datatype can be lijke the one shown here.

```
TYPE derived
   ! Internal variables
   INTEGER          :: counter
   REAL             :: number
   LOGICAL          :: used
   CHARACTER(LEN=10) :: string
END TYPE derived
! A declaration of a variable of
! the new derived datatype
TYPE (derived)      :: my_type
```

One question arises: why use derived datatypes? One answer to that is that sometimes it is desireable to group variables together and to refer to these variables under a common name. It is usually a good practice to select a name of the abstract datatype to indicate the contents and area of use.

4.3 Instructions

There are two main types of instructions. One is for program control and the other is for giving a variable a value.

4.3.1 Instructions for program control

Instructions for program control can be split into three groups, one for loops, one for tests (even though a loop usually have an implicit test) and the last for assigning values to variables and perform mathematical operations on the variables. In Fortran all loops starts with the reserved word *DO*. A short example on a simple loop is given in the following piece of code.

```
DO i = 1, 100
   !// Here instructions are performed 100 times
   !// before the loop is finished
END DO
```

The next example shows a non terminating loop where an *IF*-test inside the loop is used to exit the loop when the result of the test is true.

```
DO
   a = a * SQRT(b) + c
   IF (a > z) THEN
      !// Jump out of the loop
      EXIT
   END IF
END DO
```

This small piece of code gives the variable *a* a value from the calculation of the square root of the variable *b* and multiplied with the last value of *a* and the addition of variable *c*. When the value of *a* is greater then the value of variable *z* the program transfer control to the next instruction after the loop. We assumes here that all the variables has been initialized somewhere in the program before the loop. The various Fortran instructions will be described in the following chapters through the examples on how problems can be solved by simple Fortran programs.

5 A small program example

To make things a little clearer we shall take a small problem and program it using what we have learned so far. The problem is to write a small program calculating the daynumber in a year according to the date. We assume that the year is no a leapyear.

We start by writing the program skeleton and then fill it up with the code to solve the problem.

```
PROGRAM daynumber
  IMPLICIT NONE

END PROGRAM daynumber
```

All Fortran programs begins with the reserved word *PROGRAM* and then the program name. In our case the program name is *daynumber*. The sentence *IMPLICIT NONE* should be mandatory and is to prevent the use of implicit declarations which has been, and still is the default behavior of the Fortran compiler.

The next step is to declare some variables and constants which we are going to use calculating the daynumber.

```
PROGRAM daynumber
  IMPLICIT NONE
  INTEGER                 :: counter
  INTEGER,DIMENSION(12)   :: months
  INTEGER                 :: day, month
  INTEGER                 :: daynr
  ....
END PROGRAM daynumber
```

We have here declared four integer variables and one integer array with 12 elements. The first variable is a counter variable which will be used to traverse the array to select the number of days in the months before the given month. A variable to hold the day and month is also there together with the variable *daynr* which will contain the result of the calculations.

Then we will have to perform some initializing of the array, the day and month.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  daynr = 0
  day = 16
  month = 9
  months(:) = 30
  ....
END PROGRAM daynumber
```

Initializing the scalar variables is not difficult, but usually we would have to initialize each element of the array separately. Fortran 95 and 2003 has a built in functionality allowing us to initialize a whole array with one value. The next step is to change the number of days in the months that has 28 or 31 days.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  months(1) = 31
  months(2) = 28
  months(3) = 31
  ....
END PROGRAM daynumber
```

The rest of the *months* has to be initialized like *months(1), months(3)* and so forth for month number 5, 7,8 10 and 12.

The next step is to loop through all the elements in the *months* array up to the month minus one and sum up the number of days in each moth into the *daynr* variable. After that we just add the value from the variable *day* to the *daynr* and we have our wanted result. To show the result we can use the command *PRINT *, daynr* which will display the number of days for the given date on the display.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  DO counter = 1, month - 1
    daynr = daynr + months(counter)
  END DO
  daynr = daynr + day
  PRINT *, daynr
END PROGRAM daynumber
```

In order to have a executable program we have to compile it. The compilation process takes the source code and creates a binary file linked in with the necessary system libraries so we can run th prgram. We use an open source compiler called *gfortran* and the syntax for compiling is shown here

```
gfortran -o daynr daynr.f90
```

where *gfortran* is the name of the compiler program, the argument *-o* means that the next argument to the compiler is the name of the executable program and the last argument is the name of the file containing the source code.

The resulting output from out program with the *month = 9* and the *day = 16* is **259**. You can use a calculator and perform the calculations by hand to check that the result is correct.

So what have we learned here? We have learned to never use implicit declarations of variables which is very important. There is a story from the seventies about

implicit declarations where a typing error created an uninitialized variable causing a NASA rocket launch to fail and the rocket had to be destroyed before it could cause sever damage.

5.1 Exercises

1. Use the code in this section and fill in what is missing. Compile the code and try to run it

2. Given a radius write a program calculating the circumference of a circle, compile and run the program and check that the result is correct.

3. Given a radius write a program calculating the surface of a circle, compile and run the program and check that the result is correct.

4. Given a radius write a program calculating the volume of a sphere, compile and run the program and check that the result is correct.

s

6 Interaction with the user

In the preceding example we had the day and month values as a part of the source code. if we should change the month or the day we had to do the change in the source code and compile the program again before we could run it and get the new result. this is time consuming and absolutely not user-friendly. To remedy this we add a few lines to the program after the variable declarations shown in the code below.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  PRINT *, "Enter the day number in the month: "
  READ(*,*) day
  PRINT *, "Enter the month number: "
  READ(*,*) month
  ....
END PROGRAM daynumber
```

We use the *PRINT *,* to display a prompt and then the *READ(*,*)* to read the keyboard input into the selected variable. Now we have a much more user-friendly program which will prompt the user for an input for each variable we need. The *READ(*,*)* converts the ASCII characters into a binary number corresponding to the variable name after the *READ(*,*)* statement.

It is not a very good programming practice to have text prompts hard coded like we have done here, but we should declare a text variable using the *CARACTER(LEN=??)* syntax and put the text into the variable. Using this for all the text we have a much more readable code which is important when the program code grows larger than a few lines. So let us again change the program code utilizing this.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  CHARACTER(LEN=35)       :: day_prompt
  CHARACTER(LEN=24)       :: month_prompt
  day_prompt = "Enter the day number in the month: "
  month_prompt = "Enter the month number: "
  ....
  PRINT *, day_prompt
  READ(*,*) day
  PRINT *, month_prompt
  READ(*,*) month
```

```
  ....
END PROGRAM daynumber
```

One thing is that when we use the *PRINT*, syntax the cursor automatically advances to the next line. It is much easier to have the cursor stopping at the end of the prompt waiting for the user input. We can change this behavior by replacing the *PRINT* *, with another function *WRITE()* which allows us to suppress the automatic linefeed. The following code shows hw we can do it.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  WRITE(*,FMT='(A)',ADVANCE="NO") day_prompt
  READ(*,*) day
  WRITE(*,FMT='(A)',ADVANCE="NO") month_prompt
  READ(*,*) month
  ....
END PROGRAM daynumber
```

The syntax *WRITE(*,FMT='(A)',ADVANCE="NO")* tells the compiler that the output is ASCII text (*FMT='(A)'*) and also to suppress the linefeed (*ADVANCE="NO"*). The * in the first argument to the *WRITE* function means that the output is to the screen and not to a file.

6.1 Exercises

1. Use the first code in this section and fill in what is missing. Compile the code and try to run it.

2. Make the changes to the program so the cursor stops after the prompt, compile the program and run it. Note the in the way the program intaract with the user and see which approach you find you like best.

3. Take the programs calculating the circumference, area and volume from the exercises in the previous section and write one program where you shall use the user interface code from this section asking the user to enter a radius and the calculate all three values and display them on the screen

7 Using command line argument for running programs in batch mode

A user dialog is what we mostly use today to interact with a computer program filling in values and click on a button. This is satisfying in most cases, but if we are to run the same program with different input values the interactive use of menus and dialog boxes can take too much time. Back in the "prehistoric" time of computer science all programs were run in what is called *batch mode* that is there was no interactivity with the user. The program got the input values from the command line on the terminal. Large simulation models mostly uses this "prehistoric" approach for getting input values to the program or the values are read from a file.

In this section we will take a look at how we can pass the command line argument in to a Fortran program and run it in batch mode. So let us take the program from the previous section and modify it to accept both command line arguments and a user dialog. The code below, borrowed from the previous section where we calculated the day number for a given month an day, shows how this can be done.

```
PROGRAM daynumber
  IMPLICIT NONE
  ....
  INTEGER                 :: nargs
  CHARACTER(LEN=20)       :: buffer
  nargs = COMMAND_ARGUMENT_COUNT ()
  IF(nargs < 2) THEN
    WRITE(*,FMT='(A)',ADVANCE="NO") day_prompt
    READ(*,*) day
    WRITE(*,FMT='(A)',ADVANCE="NO") month_prompt
    READ(*,*) month
  ELSE
    buffer = ''
    CALL GET_COMMAND_ARGUMENT(1,buffer)
    READ(TRIM(buffer),FMT='(I)') day
    buffer = ''
    CALL GET_COMMAND_ARGUMENT(2,buffer)
    READ(TRIM(buffer),FMT='(I)') month
  END IF
  ....
END PROGRAM daynumber
```

A little explanation might be a good idea here. First we declare two variables which we will use to check if there is enough input arguments to the program and to store each argument as we retrieve them from the system. The function *COMMAND_ARGUMENT_COUNT()* returns the number of command line arguments which we then checks to see that we have at least two arguments. If we have enough arguments we use the subroutine *GET_COMMAND_ARGUMENT* which

has two arguments, the first is the argument number, that is the first, second etc., and the second argument is a character string to hold the argument. In our case the first argument is the day and the second argument is the month which we use the *READ* function to convert the digits to an integer.

8 Exercises

1. Take the code above and extend the test to see that we have exactly two input arguments and if the number of input arguments is greater than two print a usage message and exit the program.

9 The Basics of File Input/Output

In the previous section we was exploring the input from the keyboard and the output to the display. In most Fortran programs the input of values are read from files and the result written to another file. We shall now take an ordinary text file (*ASCII* text) and as an example on how to read the contents of a file into an array of values.

```
PROGRAM readvalues
  IMPLICIT NONE
  ....
END PROGRAM readvalues
```

As usual we start with the program skeleton and add the necessary program code as we proceeds. First we need to know how the input file looks like so we can declare the correct array to read the values into. The firs seven lines in the file looks like this:

```
Number of lines with value pairs: 12481
Date     Water-flow
717976   7.140
717977   6.570
717978   6.040
717979   5.780
717980   5.530
```

The first line tells us the number of lines which contains the date and value pairs. Note that the date is the number of days since January the first year zero and the value pairs are separated with the "tab" character.

So how do we go about getting the contents of this file into an array for the dates and another for the water-flow values? First of all we need to declare some variables which will be used for opening the file and read the contents line by line.

```
PROGRAM readvalues
  IMPLICIT NONE
  INTEGER, PARAMETER              :: lun = 10
  INTEGER                         :: res, i
  CHARACTER(LEN=80)               :: cbuffer
  INTEGER                         :: flength
  INTEGER,ALLOCATABLE,DIMENSION(:) :: dates
  REAL,ALLOCATABLE,DIMENSION(:)   :: water_flow
END PROGRAM readvalues
```

Now we have declared an integer constant "lun" with the value 10. This constant will be used as a file pointer for opening and reading the file. The next step is to open the file using the *OPEN* function.

```
PROGRAM readvalues
  IMPLICIT NONE
```

```
  ....
  OPEN(UNIT=lun,FILE='waterflow.txt',FORM='FORMATTED',IOSTAT=res)
  IF(res /= 0) THEN
    PRINT *, 'Error in opening file, status: ', res
    STOP
  END IF
  ....
END PROGRAM readvalues
```

The arguments for the *OPEN* function are first the unit number, then the filename, next is the form of the file which in this case is *FORMATTED* meaning it is a readable text file and last is the return status from the underlying operating system. If the file was opened properly the return status is zero which we used an *IF* test to test. If an error occurred the return status would be a number different from zero and we would then stop the program.

Assuming the opening of the file was ok we proceed to read the first line into the variable *cbuffer*

```
PROGRAM readvalues
  IMPLICIT NONE
  ....
  READ(UNIT=lun,FMT='(A)',IOSTAT=res) cbuffer
  IF(res /= 0) THEN
    PRINT *, 'Error in reading file, status: ', res
    CLOSE(UNIT=lun)
    STOP
  END IF
  ....
END PROGRAM readvalues
```

Again we have some arguments to the *READ* function. Firs is, as always, the unit number followed by the format which is in this case an ASCII character string (*FMT='(A)'*) and the last is the return status from the underlying operating system. Outside the parenthesis enclosing the arguments is the variable *cbuffer* where the value will be placed. If an error occurs we close the file using the *CLOSE* function and the stops the program. Now that we have the first line read into the variable we can convert the character string into an integer using the *READ* function we used to rad the first line from the file. Of course we have to extract the file length from the character string since the file length is the last part in the string.

```
PROGRAM readvalues
  IMPLICIT NONE
  ....
  INTEGER                          :: c_position, string_length
  string_length = LEN_TRIM(cbuffer)
  c_position = INDEX(cbuffer,':')
```

```
  READ(cbuffer(c_position+1:string_length),FMT='(I15)') flength
  ....
END PROGRAM readvalues
```

To extract the file length we has to find the position of the colon in the string and we use the *INDEX* function to get the character position in the string. To find the last non blank character in the string we use the *LEN_TRIM* function. Now we have the position where teh number starts and ends in the *cbuffer* string and to get the ASCII digits into an integer we use the *READ* function to read the contents of the part of the *cbuffer* into the *flength* using the internal conversion functionality in the *READ* for the conversion.

Now that we have the length of the file we can start to allocate the memory space we need to be able to read the dates and values into the allocatable variables. To get the memory space we need we use the *ALLOCATE* function on both the *dates* and *water_flow* as shown in the next example.

```
PROGRAM readvalues
  IMPLICIT NONE
  ....
  ALLOCATE(dates(flength),STAT=res)
  IF (res /= 0) THEN
    PRINT *, 'Error in allocating memory, status: ', res
    CLOSE(UNIT=lun)
    STOP
  END IF
  ....
END PROGRAM readvalues
```

Like in the file operations we get a return status from the call to the *ALLOCATE* function and any status number except zero is an error. We should always perform a test on return status to catch possible errors. Note that in the *ALLOCATE* we have two arguments, one is the allocatable variable and the second is the return status which here is preceded with the *STAT* in contrast to the *IOSTAT* for file operations.

All we have to do now is to skip the next header line and read the rest of the file into the respective variables.

```
PROGRAM readvalues
  IMPLICIT NONE
  ....
  READ(UNIT=lun,FMT='(A)',IOSTST=res) cbuffer
  DO i = 1, flength
    READ(UNIT=lun,FMT='(I6,X,F6.3)') dates(i), water_flow(i)
  END DO
END PROGRAM readvalues
```

The only difference from the previous call to the *READ* inside the lopp is that we have a more complex formatting statement. The *I6* means we have an integer with

six digits, the *X* means we skip this character and the last item tells us that we have a real number with a total of six positions including the decimal point. Also we have two variables in stead of one where the variable *i* is the index (or counter) variable telling in which position of the two arrays we want to place the values from the file. What is missing from this code is the test of the status variable and the error handling.

10 Binary files

In the previous section we took a look at ordinary text files (*ASCII* files). In this section we will take a look at the binary files which is mostly used by Fortran programs.

We start by asking what the difference is between a text file and a binary file. A text file is a file which can be displayed in a readable format on the screen and modified by using an ordinary text editor. In contrast a binary file will not be displayed in a readable format and cannot be modified by using a text editor. The example below shows how a part of a binary file would look like on the screen.

```
p=
-@>
=1@>
<BD>5@>
<BD>5@<CD><CC><CC><CC><CC><CC>8@
=<8A>7@<9A><99><99><99><99>^Y6@<A4>p=
+@<CC><CC><CC><CC><CC><CC>-@<B8>^^<85><EB>Q8<@^_<85><EB>Q<B8><9E>
A@\<8F><C2><F5>(<9C>B@<E6>@@
<CD><CC><CC><CC><CC>^LD@<85><EB>Q<B8>^^<85>B@<B8>^^<85><EB>Q<B8>@@
^W@@<AE>G<E1>z^T.@@\<8F><C2><F5>(\7@
```

The data this shows is really a set of numbers.

```
  14.7200
  17.2400
  21.7400
  21.7400
  24.8000
  23.5400
  22.1000
  13.8200
  14.9000
  28.2200
```

So why, can we ask, do we use binary files at all? The answer is storage space and the speed of writing data to a file and reading data from a file. Let us take an example of one large number like *18734991.344679912691132* where each digit uses one byte (*8 bits*) of memory and storage and therefore would use 24 bytes to

represent this number. In contrast a single precision number uses only 4 bytes and a double precision number uses 8 bytes. So each time we would read this number from a disk file or write it to a disk file we could read 3 binary numbers with the same speed as one number of *ASCII* text. In addition the computer would use time to convert the number from the *ASCII* representation to the binary counterpart. If we had a very large file (which is very common in the field of natural sciences) we waste a lot of time using *ASCII* numbers.

Let us take look at how we perform binary I/O operations using Fortran. Like the text files we have to open it before we can access the contents. The following code snippets shows how we open a binary file and read the contents into an array.

```
PROGRAM readbinary
  IMPLICIT NONE
  INTEGER, PARAMETER              :: lun = 10
  INTEGER                         :: res, i, l
  REAL,ALLOCATABLE,DIMENSION(:)   :: temperatures
....
END PROGRAM readbinary
```

We use a unit number to refer to the file once we have opened it like we did for text files. The file contains a set of temperatures. The first entry in the file is an integer number containing the size of the temperature data which is in single precision format.

```
PROGRAM readbinary
  IMPLICIT NONE
....
  OPEN(UNIT=lun,FILE='temperature.bin',FORM='UNFORMATTED',IOSTAT=res)
  IF(res /= 0) THEN
    PRINT *, 'Error in opening file'
    STOP
  END IF
....
END PROGRAM readbinary
```

After opening the file we have to allocate space for the array before we can read the data from the file into the array. To do this we first have to read the integer number to get the legnth of the array

```
PROGRAM readbinary
  IMPLICIT NONE
....
  READ(UNIT=lun,IOSTAT=res) l
  IF(res /= 0) THEN
    PRINT *, 'Error in reading file'
    CLOSE(UNIT=lun)
    STOP
```

```
  END IF
  ALLOCATE(temperatuers(l),STAT=res)
  IF(res /= 0) THEN
    PRINT *, 'Error in allocating space'
    CLOSE(UNIT=lun)
    STOP
  END IF
....
END PROGRAM readbinary
```

It is a good programming practice to test if any I/O operation failes. The same is for the allocation of memory space. It is no use to continue to run the program if we cannot get the data or allocate space for the data in memory. So now we have allocated space and can start to read the data into the array.

```
PROGRAM readbinary
  IMPLICIT NONE
....
  READ(UNIT=lun,IOSTAT=res) temperature
  IF(res /= 0) THEN
    PRINT *, 'Error in reading file file'
    STOP
  END IF
....
END PROGRAM readbinary
```

In contrast to the text file we read the whole dataset in one operation thus saving execution time. Also there is no need to convert from *ASCII* digits to binary number since the data is stored as binary numbers. Now that we have gotten the temperatures into the array we can perform the operations on the data as we wish.

In addition to ASCII files and binary files Fortran has a third type of files. it is called a *NAMELIST* file. A namelist file is used to load values to a set of variables in one read operation without specifying any variables receiving data like in an ordinary read. So how are we using this namelist construct? The code example below shows how this can be done.

```
PROGRAM nml_test
  IMPLICIT NONE
  INTEGER, PARAMETER              :: lun = 10
  INTEGER                         :: res
  INTEGER                         :: x, y, z
  INTEGER                         :: l
....
  NAMELIST /ints/ x, y, z, l
....
END PROGRAM nml_test
```

The line containing the *NAMELIST* is split into three parts. First it is the keyword namelist, then the name of the namelist */ints/* in this case and lst the variables belonging to the namelist. Note that the name of the namelist is enclised in slashes. To read the contents of the namelist file we open it as an ordinary file, but we use another use of the read function. The code below shows how this can be done.

```
PROGRAM nml_test
  IMPLICIT NONE
....
  OPEN(UNIT=lun,FILE='',STATUS='OLD',IOSTAT=res)
  READ(UNIT=lun,NML=ints,IOSTAT=res)
....
END PROGRAM nml_test
```

So how does a namelist file look like? The code below shows how an namelist file for the *ints* namelist is written.

```
&ints
x = 10
y = 14
z = 6
l = 99
/
```

The first line starts with an ampersand & followed by the name of the namelist. The next lines is the variables we have declared together with the values for each variable. Te last line is a slash / denoting the end of the namelist. A namelist file can have several namelists each namelist enclosed in the ampersand and slash.

10.1 Exercises

1. Write a program which reads the contents of a short ASCII file into an array, perform the calculation $array(i) = array(i) + i$ and save the result in a new file

2. Do the same, but this time read and write in binary format

11 Introduction to functions

In the previous section we learned how to read ASCII data from a file and allocating memory space for arrays using several intrinsic functions like *OPEN* etc. Fortran has a large amount of intrinsic functions, but sometimes it is necessary to write your own to break down a complex problem into smaller more manageable parts. In this section we shall make an introduction to writing your own functions.

Let us take the program calculating the *daynumber* and add a function deciding if we have a leap year. The program would then have an additional declaration of the function *leapyear*. As we know we have to declare our own functions as an external function. The code snippet show how we declare an external logical (*boolean*) function.

```
PROGRAM
  IMPLICIT NONE
....
  LOGICAL, EXTERNAL       :: leapyear
  INTEGER                 :: year
....
  IF(leapyear(year)) THEN
    month(2) = 29
  ELSE
    month(2) = 28
  END IF
END PROGRAM
```

For those who are familiar with *Matlab* knows that each function has to reside in a separate file with the same name as the function. In *Fortran* we can have several functions in the same file. The next code example shows how we program the *leapyear* function.

```
FUNCTION leapyear(year) RESULT(isleapyear)
  IMPLICIT NONE
  INTEGER, INTENT(IN)     :: year
  LOGICAL                 :: isleapyear

END FUNCTION leapyear
```

The declaration of a function starts with the keyword *FUNCTION* then the name of the function, the input arguments and finally the keyword *RESULT* with the variable holding the result of the function as the output argument. The type of the output argument defines the type of the function. In our case we have a *logical* function, but we can have functions returning a value from all of our datatypes. So let us program the rest of the *leapyear* function. Note the use of the construct *INTENT(IN)* which prevents us to overwrite the contents of the input argument.

```
FUNCTION leapyear(year) RESULT(isleapyear)
```

```
  IMPLICIT NONE
....
  INTEGER                 :: res1, res2, res3
  isleapyear = .FALSE.
  res1 = MOD(year,4)
  res2 = MOD(year,100)
  res3 = MOD(year,400)
  PRINT *, res1, res2, res3
  IF(res1 == 0) THEN
    IF((res1 == 0) .AND. (res2 == 0) .AND. (res3 /= 0)) THEN
      isleapyear = .FALSE.
      RETURN
    END IF
    IF((res1 == 0) .AND. (res2 == 0) .AND. (res3 == 0)) THEN
      isleapyear = .TRUE.
      RETURN
    END IF
    isleapyear = .TRUE.
    RETURN
  END IF
END FUNCTION leapyear
```

A little explanation of what we have done here might be appropriate. We have declared three help variables to contian the *modulo* division of the year and the nubers 4, 100 and 400. As we all know the formula to determine if a year is a leapyear or not is tha tif the year is divisible with 4 and not divisible with 100 we have a leapyear. If the year is divisible with 4 and also with 100, but not with 400 we have a leapyear. For all othe results we do not have a leapyear.

11.1 Exercises

1. Take the program calculating the circumference of a circle and make a function of it
2. Do the same with the area of a circle and the volume of a sphere
3. Write a main program testing the three functions and check that it is working properly

12 Introduction to subroutines

In the previous section we learned how to program a function. Using *Fortran* we have also the use of subroutines. A subroutine is in most ways the same as a function, but without returning a value like a mathematical function. So why use subroutines? A subroutine can have several input arguments like a function, but in addition a subroutine can have one or more output arguments thus allowing for a more flexible way of performing calculations.

Let us take the function *leapyear* and make a subroutine out of it. When we are using subroutines we do not have to declare the subroutine as an external procedure like we had to do with the function.

```
PROGRAM
  IMPLICIT NONE
....
  INTEGER                :: year
  LOGICAL                :: isleapyear
....
  CALL leapyear(year,isleapyear)
  IF(isleapyear) THEN
    month(2) = 29
  ELSE
    month(2) = 28
  END IF

END PROGRAM
```

Like functions we can have several subroutines in the same file. The next code example shows how we program the subroutine *leapyear*.

```
SUBROUTINE leapyear(year, isleapyear)
  IMPLICIT NONE
  INTEGER, INTENT(IN)     :: year
  LOGICAL, INTENT(OUT)    :: isleapyear

END SUBROUTINE leapyear
```

The declaration of a subroutine starts with the keyword *SUBROUTINE* then the name of the subroutine and the arguments where one or more of the arguments are used to transfer the result of the subroutine call back to the calling process. The keywords *INTENT(IN)* and *INTENT(OUT)* are used to prevent wrong use of the arguments. Using *INTENT(IN)* tells the compiler that we can only read the contents of the argument and *INTENT(OUT)* means we can only write values to the argument.

```
SUBROUTINE leapyear(year, isleapyear)
  IMPLICIT NONE
```

```
....
  INTEGER                 :: res1, res2, res3
  isleapyear = .FALSE.
  res1 = MOD(year,4)
  res2 = MOD(year,100)
  res3 = MOD(year,400)
  PRINT *, res1, res2, res3
  IF(res1 == 0) THEN
    IF((res1 == 0) .AND. (res2 == 0) .AND. (res3 /= 0)) THEN
      isleapyear = .FALSE.
      RETURN
    END IF
    IF((res1 == 0) .AND. (res2 == 0) .AND. (res3 == 0)) THEN
      isleapyear = .TRUE.
      RETURN
    END IF
    isleapyear = .TRUE.
    RETURN
  END IF
END SUBROUTINE leapyear
```

In our main program we use the syntax *CALL leapyear(year,isleapyear)* and *IF(isleapyear) THEN* instead of the construct *IF(leapyear(year)) THEN*.

So then we can ask when do we use subroutines and when functions? There is no exact asnwer to this question, but it has been more common to use subroutines which can have optional input and output arguments.

12.1 Exercises

1. Take the program calculating the circumference of a circle and make a subroutine of it
2. Do the same with the area of a circle and the volume of a sphere
3. Write a main program testing the three subroutines and check that it is working properly

13 Arrays and pointers

Let us take a closer look at arrays. An array can be a vector or a matrix in two or more dimensions. The data type can be of all the standard data types in addition to derived data types. A Fortran pointer is an alias for a variable or an array. In addition a pointer can work as an allocatable array.

For those who have a knowledge of C/C++ programming the pointer is a way to pass the address of a variable through an argument to the called function. This is known as *call by reference* in contrast to the *call by value* which is the default way of passing arguments in C/C++. In Fortran all arguments are addresses so it is *call by reference* which is the default here.

So how do we use a pointer in Fortran? First of all we declare a pointer just like any other variable. Next we have to point the pointer at some target. The target have to have the attribute *TARGET* and can be of any data type included a derived data type. The pointer has to be declared as the same data type as the target and has to have the same shape. The following code show how we declare a target variable and a corresponding pointer.

```
REAL, TARGET, DIMENSION(10,10)           :: matrix
REAL, TARGET, ALLOCATABLE,DIMENSION(:,:) :: matrix2
REAL, POINTER                            :: p(:,:)
....
ALLOCATE(matrix2(20,20))
p => matrix(2:3,2:3)
p => matrix2(10:12,18:20)
```

Here we have declared a real array, *matrix* in two dimensions which has the target attribute allowing a pointer to point at it. The pointer *p* is declared as a real pointer with the same shape as the target. We then let the pointer *p* point to a part of the *matrix* array from row 2 to 3 and column 2 to 3. To traverse the contents of the pointer we use indexes from 1 to 2 in both direction. This means that p(1,1) is the same as matrix(2,2). Next we let p point to matrix2 from row 10 to 12 and column 18 to 20. Traversing p we use the same indexes as when p pointed to matrix.

In addition to be used as an alias for an array or a part of an array the pointer can also be used as an allocatable array by itself. In contrast to the target array we used above the pointer always has to be allocated since it is pointing to nothing when we declare it.

```
REAL, POINTER                          :: p(:,:)
....
ALLOCATE(p(10,10))
```

Here we declare a pointer array in two dimensions. In order to assign values to the array we have to allocate space which is done using the *ALLOCATE* function. Now we can assign values to p just like any other array. If we declare another pointer *p2* we can use the new pointer to point to a part of p just like we used p to point to a part of matrix an matrix2.

13.1 Exercises

1. Write a program using two arrays of different dimensions and one pointer *P* which will point to different places in the two arrays and print the contents of the pointer to the screen. Let the pointer point to a very small part of the arrays so it is easier to see the contents on the screen

2. Extend the program changing the values in the arrays where the pointer is pointing and write the values to the screen

3. Extend the program by declaring a second pointer *P2*. Let P pointer point to a small part of the first array and P2 to the second array. Print the values from P and P2 to the screen. Then use the construct

   ```
   P2 = P
   ```

   and print again the contents of P and P2. Try to explain what happens.

14 Introduction to modules

In the previous sections we have used programs, functions and subroutines to solve our computing problems. With less complex problems this is an ok approach, but when the problem increases in complexity we have to change the way we break down the problem and start to look into the data structures and procedures. To illustrate this we will take the program reading water flow values and the date as the number of days after 01.01.0000 and make a global data structure with procedures working on the global data structures.

To facilitate this we will introduce the *Module* which is a structure with variable declarations and procedure declarations. We start breaking down the problem by looking at the data structure. We know that the file contains one line with the number of lines with the value pairs (date and water flow) and a line with the description of each column in the file. We need then one variable to store the number ov value pairs and two single dimension arrays to store the date and water flow. We have to have these array allocatable since we do not know in advance the number of value pairs. In addition we need a variable to hold the filename, the *ASCII*-digits, the unit number, a status variable and a loop variable. The code snippet below shows a working module containing the necessary data structure to solve our problem.

```
MODULE flowdata
  IMPLICIT NONE
  INTEGER, PARAMETER              :: lun = 10
  INTEGER                         :: res
  INTEGER                         :: flength
  CHARACTER(LEN=80)               :: filename
  CHARACTER(LEN=80)               :: cbuffer
  INTEGER                         :: c_position, string_length
  INTEGER,ALLOCATABLE,DIMENSION(:) :: dates
  REAL,ALLOCATABLE,DIMENSION(:)   :: water_flow
END MODULE flowdata
```

In addition to the global data structure we need the procedures to get the data into the variables. The code below shows how we declare the subroutine to read the number of value pairs into the arrays *dates* and *water_flow* variable. the arrays.

```
MODULE flowdata
  IMPLICIT NONE
  ....
CONTAINS
  SUBROUTINE read_data()
    IMPLICIT NONE
    INTEGER                       :: i
    OPEN(UNIT=lun,FILE=filename,FORM='FORMATTED',IOSTAT=res)
    READ(UNIT=lun,FMT='(A)',IOSTAT=res) cbuffer
```

```
    string_length = LEN_TRIM(cbuffer)
    c_position = INDEX(cbuffer,':')
    READ(cbuffer(c_position+1:string_length),FMT='(I15)') flength
    ALLOCATE(dates(flength),STAT=res)
    ALLOCATE(water_flow(flength),STAT=res)
    READ(UNIT=lun,FMT='(A)',IOSTAT=res) cbuffer
    DO i = 1, flength
      READ(UNIT=lun,FMT='(I6,X,F6.3)',IOSTAT=res) dates(i), water_flow(i)
    END DO
    CLOSE(UNIT=lun)
  END SUBROUTINE read_data
....
END MODULE flowdata
```

A little explanation is in place here. First we separate the declarations of the global variables from the procedure declarations with the keyword *CONTAINS*. Since the variables with the exception of the index variable in the do-loop are global we can use them directly in the subroutine. The subroutine is declared as usual, but is now residing inside the module. In the same manner we shall proceed with the rest of the necessary procedures in order to extract the data we need from the arrays. We will of course need a function to convert a date to an integer according to the date array in the file. The date should have the format *dd-mm-yyyy* and the result an integer containing the number of days from 01.01.0000. To make this correct we also need the function *leapyear* which we solved in the section about functions. In the code below we have the first part of the function *date2number*.

```
MODULE flowdata
  IMPLICIT NONE
....
  CONTAINS
....
  FUNCTION date2number(datestr) RESULT(datenum)
    IMPLICIT NONE
    CHARACTER(LEN=10)              :: datestr
    INTEGER                        :: datenum
    INTEGER                        :: i
    INTEGER                        :: year
    INTEGER                        :: month
    INTEGER                        :: day
    INTEGER,DIMENSION(12)          :: marray
    marray = 31
    marray(2) = 28
    marray(4) = 30
    marray(6) = 30
    marray(9) = 30
```

```
    marray(11) = 30
    datenum = 0
....
END MODULE flowdata
```

This function has a character string as an input argument and an integer as the result. The input argument is in the format of *dd-mm-yyyy*. the array *marray* will contain the number of days in each month. Note thar we use the *marray = 31* to initialize the whoel array with the value of 31. Then we just modify the elements for the moths with fewer than 31 days afterwards. We also initialize the resulting variable to zero so we can add the number of days to it for each iteration.

```
MODULE flowdata
  IMPLICIT NONE
....
    READ(datestr(1:2),FMT='(I2)') day
    READ(datestr(4:5),FMT='(I2)') month
    READ(datestr(7:10),FMT='(I4)') year
    DO i = 0, year-1
      IF(.NOT. leapyear(i)) THEN
        datenum = datenum + 365
      ELSE
        datenum = datenum + 366
      END IF
    END DO
    DO i = 1,month-1
      datenum = datenum + marray(i)
    END DO
    datenum = datenum + day
    IF(leapyear(year)) THEN
      datenum = datenum + 1
    END IF
  END FUNCTION date2number
....
END MODULE flowdata
```

Note that we use the *READ* function to extract part of the input argument and store the binary value in the variables *day*, *month* and *year*. Now we can begin to take a look at the part that extracts the indexes of a subset of the arrays *dates* and *water_flow*. We call this subroutine *find_indexes*. Using this subroutine to find the indexes of the subset we can in our main program access the subset of the *water_flow* data using the start date and end date of the subset. The code below shows how this can be done.

```
MODULE flowdata
  IMPLICIT NONE
```

```
....
  SUBROUTINE find_indexes(start_date, end_date, start_index, end_index)
    CHARACTER(LEN=10),INTENT(IN)    :: start_date
    CHARACTER(LEN=10),INTENT(IN)    :: end_date
    INTEGER, INTENT(OUT)            :: start_index
    INTEGER, INTENT(OUT)            :: end_index
    INTEGER                         :: sday
    INTEGER                         :: eday
    INTEGER                         :: i
    PRINT *, start_date
    PRINT *, end_date
    sday = date2number(start_date)
    eday = date2number(end_date)
    PRINT *, sday, eday
    start_index = 0
    end_index = 0
    DO i = 1, flength
      IF(sday == dates(i)) THEN
        start_index = i
      END IF
      IF(eday == dates(i)) THEN
        end_index = i
        EXIT
      END IF
    END DO
  END SUBROUTINE find_indexes
....
END MODULE flowdata
```

14.1 Exercises

1. Write a main program using the module *flowdata* to read the values of the input file, extract data ranging from the date 01.03.1989 to the date 31.05.1989 and display the contents of the *water flow* on the screen.

2. Take the main program and add code to write the extracted data to a new file. Then use a program to plot the extracted data (you can use any plotting program available for you).

3. Add code to the main program to have a user interface asking for the start and end date.

4. Run the new main program and extract the data from 01.01.1989 to 31.08.1990 which is what is called a hydrological year and plot the extracted data. Look at the plot and try to explain the variations in the water flow.

15 Introduction to derived data types

Using modules is a way to structure the program code making it easier to maintain and utilizing more advanced features writing more secure programs. In order to make variables easier to access and use as arguments we will now take a look at *derived* data types.

In the second section we looked at the syntax and then the *derived* data types was mentioned. To show how we can utilize a derived data type let us take some of the global data declarations from the module in the previous section and modify the code as shown below.

```
TYPE flow
  IMPLICIT NONE
  INTEGER                          :: flength
  CHARACTER(LEN=80)                :: filename
  CHARACTER(LEN=80)                :: cbuffer
  INTEGER                          :: c_position, string_length
  INTEGER,POINTER                  :: dates(:)
  REAL,POINTER                     :: water_flow(:)
END TYPE flow
```

Here we have taken the variables needed to extract the flow data and packe them into a derived data type called *flow*. Note that in contrast to the original declarations of the arrays holding the date and water flow we now use the *POINTER* attribute. This is necessary because the syntax demands the use of *POINTER* in stead of *ALLOCATABLE* inside a *TYPE* declaration. To declare a variable of their new data type we use the following code:

```
TYPE(flow)                         :: my_data
```

Here we declare a new variable named *my_data* of the *flow* data type. Note that the name of the derived data type is enclosed in a left and right parenthesis So how do we access the variables internal to the derived data type? The syntax for this is that we use the name of the derived variable, add a percent sign and then the name of the internal variable. The code below shows how this can be done.

```
my_data%filename
my_data%water_flow(1:20)
```

Even though the variable *my_data%water_flow* is a *POINTER* we access the elements just as we would an ordinary array. Using derived data types is, among other things, a way to write safer code by using one name to refer to a set of variables of different types.

A more complicated use of derived data types and modules is the case which we will look into now. We have a terrain model where the terrain is divided into a grid and the height for each grid point is stored in an ASCII file. The format of the file is four header lines where the first is the number of grid points in the

y-direction and the next is the number of grid points in the *x-direction* (i.e. number of columns and rows), the third line is cell size and the fourth is the no data value. The rest of the file is the height for each grid point. Our job is to crate a module that can read the contents of such a file and have the values stored in a derived data type created for such a data set.

So how do we go about to create such a module? As we already has seen the file is in ASCII format with header lines and terrain data. First we will create the derived data type to hold the variables we need.

```
TYPE terrain_data
   INTEGER                          :: n_cols
   INTEGER                          :: n_rows
   INTEGER                          :: cell_size
   INTEGER                          :: no_value
   REAL, POINTER                    :: heights(:,:)
END TYPE terrain_data
```

Now we have the derived data type *terrain_data* with all the necessary variables to hold all information about a specific terrain with the height in each grid point serves as a topographic map. All we have to do now is to implement the different procedures to read the data into the variables. The following code snippets illustrates how such a module can be created.

```
MODULE terrain_map
  IMPLICIT NONE
....
  TYPE(terrain_data)               :: topographic_map
CONTAINS
  SUBROUTINE read_data(f_name, res)
    CHARACTER(LEN=80),INTENT(IN)   :: f_name
    INTEGER, INTENT(OUT)           :: res
    INTEGER, PARAMETER             :: u_number = 10
    CHARACTER(LEN=512)             :: buffer
    INTEGER                        :: b_start, b_end
    INTEGER                        :: i, j, k, l
    OPEN(UNIT=u_number,FILE=f_name,FORM='FORMATTED',IOSTAT=res)
....
  END SUBROUTINE read_data
END MODULE terrain_map
```

The first steps to solve the process of reading the data into the variables is the same as earlier wit the opening of the filename. Note the *buffer* which is declared large enough to hold the longest line in the file with some extra space. Now we come to the tricky part of the code. How do we extract the values from the buffer variable into the respective places in the *heights* array? The solution is to split the problem into smaller parts. The first part is to read the header lines and extract the values

into the respective variables in the *topographic_map* type. The following code solves this problem.

```
  SUBROUTINE read_data(f_name, res)
....
    buffer = ' '
    READ(UNIT=u_number,FMT='(A)',IOSTAT=res) buffer
    b_end = LEN_TRIM(buffer)
    DO i = b_end, b_start, -1
      IF(buffer(i:i) .EQ. ' ') THEN
        READ(buffer(i+1:b_end),FMT='(I5)') topographic_map%n_cols
        EXIT
      END IF
    END DO
....
  END SUBROUTINE read_data
```

Since we know that the first line int the file is *ncols 125* with spaces between the label and the value we have to find the position in the buffer where the value is placed. To do this we use the function *LEN_TRIM* to find the position of the last non space character in the buffer. Then we loop through the characters from the found position down to where the first space character before the value. The we simply use the *READ* function to extract the value and place it into the variable. The same algorithm is used to extract the other header values.

the next item on the agenda is to extract the height for each grid point from the file and into the array. A suggestion on how this can be done is sketched below.

```
  SUBROUTINE read_data(f_name, res)
....
    DO i = 1, topographic_map%n_rows
      buffer = ' '
      READ(UNIT=u_number,FMT='(A)',IOSTAT=res) buffer
      !// Here we write the code to extract each
      !// number from the buffer
    END DO
....
  END SUBROUTINE read_data
```

First we have to clear the character buffer for the previous contents. Then we have to read each line containing the grid points. After we have read the line we have to extract the values for each grid point and store them it the correct position in the array. Since Fortran is very sensitive to the *FORMAT* we will use another way to extract the corresponding integer numbers from the buffer. The function is called *a2i* and have to be declared external since it is not an intrinsic Fortran function. Also we have to traverse the buffer to locate the comma separating one number from another and to send the digits between two commas to the *a2i* function which

will return the binary number to be stored in the array. The next code example show how this can be done.

```
 SUBROUTINE read_data(f_name, res)
....
      b_end = LEN_TRIM(buffer)
      j = 1
      l = 1
      DO k = 1, b_end
        IF(buffer(k:k) .EQ. ',') THEN
          topographic_map%heights(i,l) = a2i(buffer(j:k-1))
          j = k+1
          l = l + 1
        END IF
      END DO
....
 END SUBROUTINE read_data
```

The first thing we do here is to find the position for the last non space character in the buffer and initialize two counter variables. Next step is to traverse the buffer and find the position of the next comma in the buffer. Finding this we send the part of the buffer between the commas to the function and stores the result in the array. Note that this solution contains an error which we will fix now. In the loop we are not extracting the value after the last comma. To extract this we have to do it outside the innermost loop. The code would then look like the example below.

```
 SUBROUTINE read_data(f_name, res)
....
      b_end = LEN_TRIM(buffer)
      j = 1
      l = 1
      DO k = 1, b_end
        IF(buffer(k:k) .EQ. ',') THEN
          topographic_map%heights(i,l) = a2i(buffer(j:k-1))
          j = k+1
          l = l + 1
        END IF
      END DO
      topographic_map%heights(i,l) = a2i(buffer(j:b_end))
....
 END SUBROUTINE read_data
```

15.1 Exercises

1. Take the code in the module *flowdata* modify it to use the derived type flow and call the module *flowmodule* so we keep the original code intact.

2. Rewrite the main program utilizing the new structure and extract the same amount of data as in the last exercise in the previous section.

3. Rewrite the part of the code where we extract the data from the buffer into the variables using a subroutine with two arguments, the buffer and the target variable.

16 Introduction to Object Oriented Programming

It is very likely that you have heard the notion *Object Oriented Programming* or *OOP* for short. An excerpt from Wikipedia explains the various parts of the notion of OOP as:
Object-oriented programming (OOP) is a programming paradigm using "objects", i.e. data structures consisting of data fields and methods together with their interactions, to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, polymorphism, and inheritance.

Data abstraction is explained as:
In computer science, abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time.

Encapsulation is explained as:
In a programming language encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- *A language mechanism for restricting access to some of the object's components.*
- *A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.*

The notion of *polymorphism* is:
Subtype polymorphism, almost universally called just polymorphism in the context of object-oriented programming, is the ability to create a variable, a function, or an object that has more than one form.

Last we have the *inheritance* which is explained as:
In object-oriented programming (OOP), inheritance is a way to reuse code of existing objects, establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior (i.e., previously coded algorithms associated with a class) from pre-existing classes called base classes or superclasses or parent classes or ancestor classes. The new classes are known as derived classes or subclasses or child classes.
Having explained a few things about OOP we are now ready to try out the theory in solving a known problem by using the OOP technique.
In the previous section we used derived data types to encapsulate several variables that was connected to a data structure. We will now use this and change the code putting everything into an object which can have several instances with different

data values. The object is a part of a module where the type is declared as usual, but with one exception. The type also has a contains statement where the names of the procedures to be used is declared. The code snippet shows how this is done.

```
MODULE class_terrain
  IMPLICIT NONE
  PRIVATE
  TYPE, PUBLIC                                :: terrain
    CHARACTER(LEN=80)                         :: filename
    INTEGER, POINTER                          :: map(:,:) => null()
    INTEGER                                   :: n_cols = 0
    INTEGER                                   :: n_rows = 0
    INTEGER                                   :: cell_size = 0
    INTEGER                                   :: no_value = 0
  CONTAINS
    PROCEDURE                                 :: load => load_data
    PROCEDURE                                 :: dump => dump_data
  END TYPE terrain
CONTAINS
....
END MODULE class_terrain
```

The first difference from the usual declaration of the type is that we do not enclose the type name in a set of parenthesis, but use the syntax of an ordinary variable declaration. Note also that we have introduces the *PRIVATE* keyword which means that anything not specific declared as *PUBLIC* is not accessible from outside the module. This is done to prevent procedures not part of the module to make unintentionally changes to values inside the module. The declaration of a procedure with the construct *fload => load_data* gives an alias for the procedure *load_data* declared after the contains keyword in the module. In addition we initialize the pointer to a *NULL* value using the construct *map(:,:) => null()*. Also the other variables are getting initialized so we can use the intrinsic constructor to create an instance of the object without having to supply values for each variable in the object.

```
PROGRAM ctest
  USE class_terrain
  IMPLICIT NONE
  TYPE(terrain)                    :: td1
....
  td1 = terrain('demfile.dat')
  CALL td1%fload()
....
END PROGRAM ctest
```

Declaring a new object of the *terrain* type is done the normal way. Then we call the

implicit constructor *td1 = terrain('demfile.dat')* where we give the input filename as the only argument. If we had not initialized the internal variables in the declaration of the type we would have to use *td1 = terrain('demfile.dat', null(),0,0,0,0)*. Omitting these last arguments to the constructor and not having the internal variables initialized in advance, the program would not compile.

Having set up the object and learned how to use the constructor to create an instance of the object we will now take a look at how the rest of the module can be programmed. Using the module from the previous section we now that we need at least two procedures, one to read the file, allocate space etc. and one to take the header lines and extract the values we need to allocate space for the terrain map.

```
CONTAINS
....
  SUBROUTINE load_data(this)
    CLASS(terrain)                          :: this
    INTEGER, DIMENSION(4)                   :: vars
    INTEGER, PARAMETER                      :: u_number = 10
    CHARACTER(LEN=512),DIMENSION(4)         :: buffer
    INTEGER                                 :: b_start, b_end
    INTEGER                                 :: i, j, k, l, res
    INTEGER, EXTERNAL                       :: a2i
....
  END SUBROUTINE load_data
```

The only argument to the *load_data* subroutine is a class variable of type terrain named *this*. The variable is an instance of the class of type terrain and contains the filename, the pointer array and the four integer values. The variables local to this subroutine is used to extract the contents of the file into the variables of the class instance. So let us take a look at the rest of the code for this subroutine.

```
  SUBROUTINE load_data(this)
....
    OPEN(UNIT=u_number,FILE=this%filename,FORM='FORMATTED',IOSTAT=res)
    buffer = ' '
    DO i = 1, 4
      READ(UNIT=u_number,FMT='(A)',IOSTAT=res) buffer(i)
      IF(res /= 0) THEN
        PRINT*, 'Error in reading line 1, status ', res
        CLOSE(UNIT=u_number)
        RETURN
      END IF
    END DO
....
  END SUBROUTINE load_data
```

The only difference from this version of the subroutine and the one in the previous

section is that we precede the name of the variables with *this%* just like we would for a non object derived type. Of course the local variables for the subroutine is not preceded by *this%*. In addition we use an array of character strings to hold the header lines in stead of a single character string and another array to hold the integer values extracted from the header lines.

```
  SUBROUTINE load_data(this)
....
    CALL extract_header_lines(buffer,vars)
    this%n_cols = vars(1)
    this%n_rows = vars(2)
    this%cell_size = vars(3)
    this%no_value = vars(4)
    ALLOCATE(this%map(this%n_rows,this%n_cols),STAT=res)
    IF(res /= 0) THEN
      PRINT *, 'Allocation failure, status ', res
      CLOSE(UNIT=u_number)
    END IF
....
  END SUBROUTINE load_data
```

Here we have modified the *extract_header_lines* subroutine to take two arrays as input arguments. After extracting the values from the header lines we copy them into the various internal variables for this class instance. The rest of the subroutine is the same as the one in the previous section.

The next part we have to program is the *dump_data* subroutine. It can be constructed like the code snippet below.

```
  SUBROUTINE dump_data(this)
    CLASS(terrain)                          :: this
    INTEGER, PARAMETER                      :: u_number = 11
    INTEGER                                 :: i, j, res
    OPEN(UNIT=u_number,FILE=this%filename,FORM='FORMATTED',IOSTAT=res)
    WRITE(UNIT=u_number,FMT='(I5)',IOSTAT=res) this%n_cols
    WRITE(UNIT=u_number,FMT='(I5)',IOSTAT=res) this%n_rows
    WRITE(UNIT=u_number,FMT='(I5)',IOSTAT=res) this%cell_size
    WRITE(UNIT=u_number,FMT='(I5)',IOSTAT=res) this%no_value
    DO i = 1, this%n_rows
      DO j = 1, this%n_cols
        WRITE(UNIT=u_number,FMT='(I4,A2)',ADVANCE='NO',IOSTAT=res) &
              this%map(i,j), ', '
      END DO
      WRITE(UNIT=u_number,FMT='(A)',IOSTAT=res) ' '
    END DO
    CLOSE(UNIT=u_number)
  END SUBROUTINE dump_data
```

Like the subroutine *load_data* we have only the instance of the object as an input argument. After opening the file we write the four header variables to the file, but in this version we do not use any preceding header text. Then we loop and write each value for the columns separated with a comma to the file. Note the use of *ADVANCE='NO'* which prevents the automatic line feed after each write. To write the column values for the next line we simply write a space character to the file without the *ADVANCE='NO'* to get a new line.

16.1 Exercises

1. Change the dump_data subroutine so that the header lines contains the ncols, nrows, cellsize and NODATA_value text strings in addition to the values.

2. Write a subroutine to clear the contents of an object instance using the same type of argument as the load_data and the dump_data subroutines. Note that the map array has to be deallocated in order so have a completely clean object instance.

3. Extend the main program with code to create a new instance with a different name and copy the contents of the first instance into the new one. Hint: you have to allocate the space for the map array in the new instance before you can copy the contents from the first instance.

A Operators

Operators in Fortran 95 are for example `IF(a > b) THEN` which is a test using numerical values in the variables. For other types of variables like characters and character strings we use the construct `IF(C1 .GT. C2) THEN`.

A.1 Overview of the operators

Table3 gives an overview of the operators

Numerical	Other	Explanation
**		Exponentiation
*		Multiplication
/		Division
+		Addition
-		Subtraction
==	.EQ.	Equal
/=	.NE.	Not equal
<	.LT.	Less
>	.GT.	Greater
<=	.LE.	Less or equal
>=	.GE.	Greater or equal
	.NOT.	Negation, complement
	.AND.	Logical and
	.OR.	Logical or
	.EQV.	Logical equivalence
	.NEQV.	Logical not equivalence, exclusive or

Table 3: Logical operators

B Intrinsic functions

Intrinsic functions in Fortran 96 are functions that can be used without referring to them via include files like in other languages where functions has to be declared before being used in the program

B.1 Intrinsic Functions

Table4, table5 and table7 gives an overview of the intrinsic functions in Fortran 95

Function	Argument	Result	Explanation
ABS	Integer real complex	Integer real complex	The absolute value
ACHAR	Integer	Character	Integer to ASCII character
ACOS	Real	Real	Arcuscosine
ADJUSTL	Character string	Character string	Left adjustment
ADJUSTR	Character	Character	Right adjustment
AIMAG	Complex	Real	Imaginary part
AINT	Real	Real	Truncate to a whole number
ALL	Logical mask, dim	Logical	True if all elements == mask
ALLOCATED	Array	Logical	True if allocated in memory
ANINT	Real	Real	Round to nearest integer
ANY	Logical mask, dim	Logical	True if all elemnts == mask
ASIN	Real	Real	Arcsine
ASSOCIATED	Pointer	Logical	True if pointing to target
ATAN	Real	Real	Arctangent
ATAN2	X=Real,Y=Real	Real	Arctangent
BIT_SIZE	Integer	Integer	Number of bits in argument
BTEST	I=Integer,Pos=Integer	Logical	Test a bit of an integer
CEILING	Real	Real	Leat integer <= argument
CHAR	Integer	Character	Integer to ASCCI character
CMPLX	X=Real,Y=Real	Complex	Convert to complex number
CONJG	Complex	Complex	Conjugate the imaginary part
COS	Real Complex	Real complex	Cosine
COSH	Real	Real	Hyperbolic cosine
COUNT	Logical mask, dim	Integer	Count of true entries in mask
CPU_TIME	Real	Real	Returns the processor time
CSHIFT	Array, shift, dim	Array	Circular shift of elements
DATE_AND_TIME	Char D,T,Z,V	Character	Realtime clock
DBLE	Integer real complex	Double precision	Convert to double precision
DIGITS	Integer real	Integer	Number of bits in argument
DIM	Integer real	Integer real	Difference operator
DOT_PRODUCT	X=Real,Y=Real	Real	Dot product
DPROD	X=Real,Y=real	Double precision	Double precision dot prod.
EOSHIFT	Array-shift,boundary,dim	Array	Array element shift
EPSILON	Real	Real	Smallest positive number
EXP	Real complex	Real complex	Exponential
EXPONENT	Real	Integer	Model exponenet of argument
FLOOR	Real	Real	Integer <= argument
FRACTION	Real	Real	Fractional pert of argument
HUGE	Integer real	Integer real	Largest number

Table 4: Intrinsic functions

Function	Argument	Result	Explanation
IACHAR	Character	Integer	Integer value of argument
IAND	Integer,Integer	Integer	Bitwise logical and
IBCLR	Integer,pos	Integer	Setting bit in pos = 0
IBITS	Integer,pos,len	Integer	Extract len bits from pos
IBSET	Integer,pos	Integer	Set pos bit to one
ICHAR	Character	Integer	ASCII number of argument
IEOR	Integer,integer	Integer	Bitwise logical XOR
INDEX	String,substring	Integer	Position of substring
INT	Integer real complex	Integer	Convert to integer
IOR	Integer,integer	Integer	Bitwise logical OR
ISHFT	Integer,shift	Integer	Shift bits by shift
ISHFTC	Integer,shift	Integer	Shift circular bits in argument
KIND	Any intrinsic type	Integer	Value of the kind
LBOUND	Array,dim	Integer	Smallest subscript of dim
LEN	Character	Integer	Number of chars in argument
LEN_TRIM	Character	Integer	Length without trailing space
LGE	A,B	Logical	String A <= string B
LGT	A,B	Logical	String A > string B
LLE	A,B	Logical	String A <= string B
LLT	A,B	Logical	String A < string B
LOG	Real complex	Real complex	Natural logarithm
LOG10	Real	Real	Logarithm base 10
LOGICAL	Logical	Logical	Convert between logical
MATMUL	Matrix,matrix	Vector matrix	Matrix multiplication
MAX	a1,a2,a3,...	Integer real	Maximum value of args
MAXEXPONENT	Real	Integer	Maximum exponent
MAXLOC	Array	Integer vector	Indices in array of max value
MAXVAL	Array,dim,mask	Array element(s)	Maximum value
MERGE	Tsource,Fsource, mask	Tsource or Fsource	Chosen by mask
MIN	a1,a2,a3,...	Integer real	Minimum value
MINEXPONENT	Real	Integer	Minimum exponent
MINLOC	Array	Integer vector	Indices in array of min value
MINVAL	Array,dim,mask	Array element(s)	Minimum value
MOD	a=integer real,p	Integer real	a modulo p
MODULO	a=integer real,p	Integer real	a modulo p
MVBITS	From pos to pos	Integer	Move bits
NEAREST	Real,direction	Real	Nearest value in direction
NINT	Real,kind	Real	Round to nearest integer value
NOT	Integer	Integer	Bitwise logical complement

Table 5: Intrinsic functions

Function	Argument	Result	Explanation
PACK	Array,mask	VEctor	Vector of array elements
PRECISION	Real complex	Integer	Decimal precision of arg
PRESENT	Argument	Logical	True if optional arg is set
PRODUCT	Array,dim,mask	Integer real complex	Product along dim
RADIX	Integer real	Integer	Radix of integer or real
RANDOM_NUMBER	Harvest = real	Real $0 <= x <= 1$	Subroutine returning a random number in harvest
RANDOM_SEED	Size, put or get	Nothing	Subroutine to set a random number seed
RANGE	Integer real complex	Integer real	Decimal exponent
REAL	Integer real complex	Real	Convert to real type
REPEAT	String,ncopies	String	Concatenate n copies of string
RESHAPE	Array,shape,pad,order	Array	Reshape source array to array
RRSPACING	Real	Real	Reciprocal of relative spacing of model
SCALE	Real,integer	Real	Returns $X \cdot b^I$
SCAN	String,set,back	Integer	Position of first of set in string
SELECTED_INT_KIND	Integer	Integer	Kind number to represent digits
SELECTED_REAL_KIND	Integer	Integer	Kind number to represent digits
SET_EXPONENT	Real,integer	Resl	Set an integer as exponent of a real $X \cdot bI - e$
SHAPE	Array	Integer vector	Vector of dimension sizes
SIGN	Integer real,integer real	Integer real	Absolute value of $A \cdot B$
SIN	Real complex	Real complex	Sine of angle in radians
SINH	Real	Real	Hyperbolic sine
SIZE	Array,dim	Integer	Number of array elements in dim
SPACING	Real	Real	Spacing of model number near argument
SPREAD	Source,dim,copies	Array	Adding a dimension to source
SQRT	Real complex	Real complex	Square root
SUM	Array,dim,mask	Integer real complex	Sum of elements
SYSTEM_CLOCK	Count,count,count	Trhough the arguments	Subroutine returning integer data from a real time clock

Table 6: Intrinsic functions

Function	Argument	Result	Explanation
TAN	Real	Real	Tangent of angle in radians
TANH	Real	Real	Hyperbolic tangent
TINY	Real	Real	Smallest positive model representation
TRANSFER	Source,mold,size	Mold type	Same bits, but new type
TRANSPOSE	Matrix	Matrix	The transpose of matrix
TRIM	String	String	REmove trailing blanks
UBOUND	Array,dim	Integer	Largest subscript of dim in array
UNPACK	Vector,mask,field	Vector type, mask shape	Unpack an array of rank one into an array of mask shape
VERIFY	String,set,back	Integer	Position in string not in set

Table 7: Intrinsic functions