|  |  |  |
| --- | --- | --- |
| **ACTIES OP HET SCHERM** | **VOICE-OVER** | **DUUR** |
|  | In scientific computing the input to your programs is very often stored in files, and the output will be written to files. Input/output or I/O is typically not the most exciting topic, but it is a very important one, also with respect to performance. In this video we will deal with very simple I/O, but there is more material on this topic later on. |  |
| 1. Add computation 2. Add open, newunit argument 3. Add file argument 4. Add access argument 5. Add action argument 6. Add status argument 7. Add form argument 8. Add close statement 9. Add write statement 10. Save file | 1. We start by writing the result of a computation to a file. We will be using a text file, since this is the easiest case. 2. The first thing to do is to open the file to write to. In Fortran, a file is represented by a unit number that is unique for every open file. You can either specify an integer as unit number, or you can have the Fortran runtime generate one that is currently not in use. The unit number will be assigned to the variable “unit\_nr”. 3. The file’s name will be “data.txt”. 4. We will write data sequentially, which is specified through the “access” argument. Other access modes will be discussed as well later on. 5. The action we want to perform is “write”. 6. The status of the file is “replace” in this case, which means that if the file already exists, it will be overwritten. If you want to avoid that, you can specify “new”. 7. As we want a text file as output, we specify that the format is “formatted”. 8. When all data has been written, we can close the file identified by its unit number. It is very important to do this, because you may loose data if you don’t 9. Finally, we use the “write” statement to actually add content to the file. The format, “fmt” specifies what the textual representation of the data will look like. In this case we want a floating point representation using 3 characters, and one digit after the decimal dot. |  |
| 1. Switch to terminal 2. Compile code 3. Run application 4. Display output file | 1. When we compile and run the code, we get a new file “data.txt”. 2. However, it doesn’t really contain the data we want. Only the first two numbers are okay, but all other data is represented as a series of asterisks. The problem is that the data doesn’t fit in the format that was specified. The width is only 3, so for instance “10.2” can not be rendered correctly. If that is the case, the Fortran runtime will simply substitute the asterisks. As you can guess, it is pretty frustrating to discover a lot of asterisks in your output after your program has been running for hours. So getting the format correctly is quite important. We will discuss the format specification in detail later. 3. Note that each invocation of “write” resulted in a line in the text file. |  |
| 1. Fix format to 8.1 2. Add iostat and iomsg to open statement 3. Test iostat for open and handle error 4. Add error handling for write and close 5. Save file | 1. Lets make sure that the data can be written properly. A width of 8 characters should be plenty. 2. It is also quite important to do error handling. All communication statements and function support that. For instance, we can provide the “iostat” and “iomsg” arguments to “open”. The “iostat” argument will be set to an integer that is non-zero if something goes wrong while opening the file. The “iomsg” argument will be set to a descriptive error message. 3. We can now test the value of “stat”, and if it is non-zero we can write the error message in “msg” and stop the execution with a non-zero status. Note that the error message is written to “error\_unit”, this is the unit number that is associated with the standard error stream. It is declared in the “iso\_fortran\_env” intrinsic module. 4. The same can be done for the “write” and “close” statements. If necessary, you can differentiate the error messages and the exit status of the application.. |  |
| 1. Switch to terminal 2. Compile code 3. Run application 4. Show data file | 1. When we compile and run the application, we get the data file we would expect. |  |
| 1. Add open statement, access and status arguments 2. Add error handling for open 3. Add read statement 4. Add iostat\_end check 5. Add iostat > 0 check 6. Add write to output\_unit 7. Add close statement 8. Save file | 1. We can now write a program that reads the file we just generated, performs some computation on each value, and writes the result to standard output. Again we use the “open”statement to get a unit number for a file. In this case the action is “read”, and the status is “old” since we can only read a file that already exists. 2. Just as before, we handle errors that could occur when opening a file. 3. The “read” statement is used to read data. The format argument “fmt” is set to an asterisk, this will usually be best when reading data. Only use a specific format code when you are 100 % sure it is identical to the one used to write the file. The textual representation of a floating point value in the file will be converted to a “real” value and stored in the variable “x”. 4. However, what happens at the end of the file? Up front, we have no idea how many values the file contains, so how do we detect the end of the file? The value obtained throught the ”iostat” argument of “read” gives us that information. When it is equal to “iostat\_end” we exit from the iteration. 5. If that is not the case, we also check whether the “iostat” argument is zero, if not, we can handle the error. 6. Although we could have used a “print” statement to write the result to standard output, we use a write to “output\_unit” instead just by way of illustration. 7. Although less critical than for a file that is open for write operations, it is good practice to close the file open for read operations as well. |  |
| 1. Switch to terminal 2. Compile code 3. Run application | 1. When we compile and run the application, we get the expected results. |  |
|  | As I already mentioned, file I/O is an important topic, so these were just the basics to get you going. We will discuss binary I/O as well as alternative access modes that can improve performance or efficiency in many cases. |  |
| **TOTALE DUUR** | | *Maak je screencast niet langer dan ca. 6 minuten.* |