Introductory Fortran Programming

Gunnar Wollan[1]

Dept. of Geosciences, University of Oslo[1]

January 27th, 2006

Outline

List of Topics

Contents

Contents

- Gentle introduction to Fortran 77 and 95 programming
- File I/O
- Arrays and loops
- Functions and subroutines
- Detailed explanation of modules
- Computational efficiency aspects
- Using modules as objects

Contents

Contents

- Gentle introduction to Fortran 77 and 95 programming
- File I/O
- Arrays and loops
- Functions and subroutines
- Detailed explanation of modules
- Computational efficiency aspects
- Using modules as objects

Contents

- Gentle introduction to Fortran 77 and 95 programming
- File I/O
- Arrays and loops
- Functions and subroutines
- Detailed explanation of modules
- Computational efficiency aspects
- Using modules as objects

Contents

- Gentle introduction to Fortran 77 and 95 programming
- File I/O
- Arrays and loops
- Functions and subroutines
- Detailed explanation of modules
- Computational efficiency aspects
- Using modules as objects

Contents

- Gentle introduction to Fortran 77 and 95 programming
- File I/O
- Arrays and loops
- Functions and subroutines
- Detailed explanation of modules
- Computational efficiency aspects
- Using modules as objects

Required background

- Programming experience with either C++, Java or Matlab
- Interest in numerical computing using Fortran
- Interest in writing efficient programs utilizing low-level details of the computer

Required background

- Programming experience with either C++, Java or Matlab
- Interest in numerical computing using Fortran
- Interest in writing efficient programs utilizing low-level details of the computer

Required background

- Programming experience with either C++, Java or Matlab
- Interest in numerical computing using Fortran
- Interest in writing efficient programs utilizing low-level details of the computer

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

About learning Fortran

- Fortran is a less complicated language than C++ and Java
- Even so it takes time to master the advanced details of Fortran 95
- At least 6 months to a year working with Fortran 95 before you are familiar with most of the details
- Four days can only get you started
- You need to use Fortran 95 in your own projects to master the language
- Fortran 77 code is not the main topic here, but you need to have some knowledge of it

List of Topics

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- The predecessor Fortran IV was replaced by Fortran 77 in the early eighties
- At IBM in 1954 a group of people started to design the FORmula TRANslator System, or FORTRAN0
- The first version of Fortran was released in 1957 and the language has evolved over time
- Like many procedural languages Fortran has a fairly simple syntax
- Fortran is good for only one thing: NUMBERCRUNCHING

Fortran 95

- Fortran 95 extends Fortran 77 with
  - Nicer syntax, free format instead of fixed format
  - User defined datatypes using the TYPE declaration
  - Modules containing data definitions and procedure declarations
  - No implicit variable declarations, avoiding typing errors

Fortran versus other languages

- C is low level and close to the machine, but can be error prone
- C++ is a superset of C and more reliable
- Java is simpler and more reliable than C++
- Python is more high-level than Java
- Fortran 95 is more reliable than Fortran 77

Fortran versus other languages

- C is low level and close to the machine, but can be error prone
- C++ is a superset of C and more reliable
- Java is simpler and more reliable than C++
- Python is more high-level than Java
- Fortran 95 is more reliable than Fortran 77

Fortran versus other languages

- C is low level and close to the machine, but can be error prone
- C++ is a superset of C and more reliable
- Java is simpler and more reliable than C++
- Python is more high-level than Java
- Fortran 95 is more reliable than Fortran 77

Fortran versus other languages

- C is low level and close to the machine, but can be error prone
- C++ is a superset of C and more reliable
- Java is simpler and more reliable than C++
- Python is more high-level than Java
- Fortran 95 is more reliable than Fortran 77

Fortran versus other languages

- C is low level and close to the machine, but can be error prone
- C++ is a superset of C and more reliable
- Java is simpler and more reliable than C++
- Python is more high-level than Java
- Fortran 95 is more reliable than Fortran 77

Speed of Fortran versus other languages

- Fortran 77 is regarded as very fast
- C yield slightly slower code
- C++ and fortran 95 are slower than Fortran 77
- Java is much slower

Speed of Fortran versus other languages

- Fortran 77 is regarded as very fast
- C yield slightly slower code
- C++ and fortran 95 are slower than Fortran 77
- Java is much slower

Speed of Fortran versus other languages

- Fortran 77 is regarded as very fast
- C yield slightly slower code
- C++ and fortran 95 are slower than Fortran 77
- Java is much slower

Speed of Fortran versus other languages

- Fortran 77 is regarded as very fast
- C yield slightly slower code
- C++ and fortran 95 are slower than Fortran 77
- Java is much slower

Why these differences

- There are some reasons why som languages are faster than others
  - The structure and complexity of the language
  - The complexity of the CPU and the experience of the compiler developers
  - Compilation vs. interpretation

Why these differences

- There are some reasons why som languages are faster than others
  - The structure and complexity of the language
  - The complexity of the CPU and the experience of the compiler developers
  - Compilation vs. interpretation

Some guidelines

- Fortran 77 gives very fast programs, but the source code is less readable and more error prone due to implicit declarations
- Use Fortran 95 for your main program and if speed is critical use Fortran 77 functions
- Sometimes the best solution is a combination of languages, e.g. Fortran with Python or C++
- Use the language best suited for your problem

Some guidelines

- Fortran 77 gives very fast programs, but the source code is less readable and more error prone due to implicit declarations
- Use Fortran 95 for your main program and if speed is critical use Fortran 77 functions
- Sometimes the best solution is a combination of languages, e.g. Fortran with Python or C++
- Use the language best suited for your problem

Some guidelines

- Fortran 77 gives very fast programs, but the source code is less readable and more error prone due to implicit declarations
- Use Fortran 95 for your main program and if speed is critical use Fortran 77 functions
- Sometimes the best solution is a combination of languages, e.g. Fortran with Python or C++
- Use the language best suited for your problem

Some guidelines

- Fortran 77 gives very fast programs, but the source code is less readable and more error prone due to implicit declarations
- Use Fortran 95 for your main program and if speed is critical use Fortran 77 functions
- Sometimes the best solution is a combination of languages, e.g. Fortran with Python or C++
- Use the language best suited for your problem

List of Topics

Our first Fortran 77 program

- Goal: make a program writing the text “Hello World
- Implementation
  - Without declaring a text string variable
  - With a text string variable declaration

Without declaring a string variable

- Fortran fixed format

```
C234567
      PROGRAM hw1
         WRITE(*,*) 'Hello World'
      END PROGRAM hw1
```

With declaring a string variable

- Fortran fixed format

```
C234567
      PROGRAM hw1
         CHARACTER*11 str = 'Hello World'
         WRITE(*,*) str
      END PROGRAM hw1
```

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

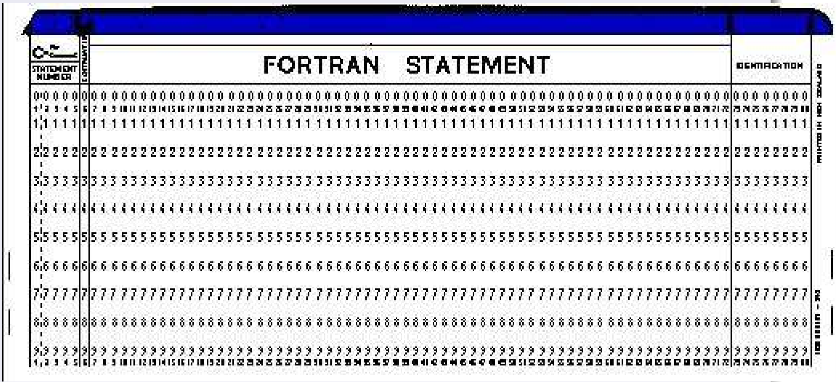
- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

Some comments to the “Hello World program

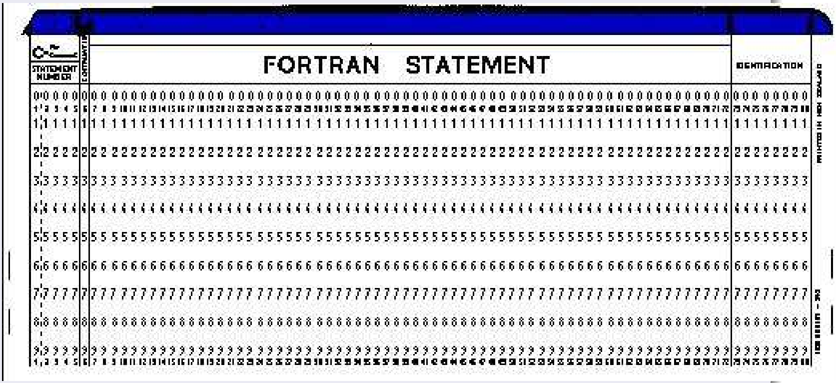
- Fortran 77 uses fixed format
- The source code is divided into positions on the line
- This is a heritage from the old days when communication with the computer was by punched cards
- A character in the first column identifies to the compiler that the rest of the line is a comment
- The coumns 2 to 5 is for jump labels and format specifiers
- Column 6 is for continuation of the previous line
- The column 7 to 72 is for the source code
- Column 73 to 80 is for comments

An example of a punched card for Fortran

- the layout of a Fortran card



An example of a punched card for Fortran

- the layout of a Fortran card
  - 

List of Topics

Scientific Hello World in Fortran 95

- Usage:

```
./hw1 2.3
```

- Output of the program hw1

```
Hello, World! sin(2.3)=0.745705
```

- What to learn
  - Store the first command-line argument in a floating-point variable
  - Call the sine function
  - Write a combination of text and numbers to the screen

Scientific Hello World in Fortran 95

- Usage:

```
./hw1 2.3
```

- Output of the program hw1

```
Hello, World! sin(2.3)=0.745705
```

- What to learn
  - Store the first command-line argument in a floating-point variable
  - Call the sine function
  - Write a combination of text and numbers to the screen

Scientific Hello World in Fortran 95

- Usage:

```
./hw1 2.3
```

- Output of the program hw1

```
Hello, World! sin(2.3)=0.745705
```

- What to learn
  - Store the first command-line argument in a floating-point variable
  - Call the sine function
  - Write a combination of text and numbers to the screen

Scientific Hello World in Fortran 95

- Usage:

```
./hw1 2.3
```

- Output of the program hw1

```
Hello, World! sin(2.3)=0.745705
```

- What to learn
  - Store the first command-line argument in a floating-point variable
  - Call the sine function
  - Write a combination of text and numbers to the screen

The code

- The hw1 program

```
PROGRAM hw1
   IMPLICIT NONE
   DOUBLE PRECISION  :: r, s
   CHARACTER(LEN=80) :: argv ! Input argument
   CALL getarg(1,argv)       ! A C-function
   r = a2d(argv)             ! Our own ascii to
                             ! double
   s = SIN(r)                ! The intrinsic
                             ! SINE function
   PRINT *, 'Hello Word sin(',r,')=',s
END PROGRAM hw1
```

Dissection(1)

- Contrary to C++ the compiler does not need to se a declaration of subroutines and intrinsic functions
- Only external functions must be declared
- Comments in Fortran 95 are the exclamation mark *!* on a line
- The code is free format unlike Fortran 77

Dissection(1)

- Contrary to C++ the compiler does not need to se a declaration of subroutines and intrinsic functions
- Only external functions must be declared
- Comments in Fortran 95 are the exclamation mark *!* on a line
- The code is free format unlike Fortran 77

Dissection(1)

- Contrary to C++ the compiler does not need to se a declaration of subroutines and intrinsic functions
- Only external functions must be declared
- Comments in Fortran 95 are the exclamation mark *!* on a line
- The code is free format unlike Fortran 77

Dissection(1)

- Contrary to C++ the compiler does not need to se a declaration of subroutines and intrinsic functions
- Only external functions must be declared
- Comments in Fortran 95 are the exclamation mark *!* on a line
- The code is free format unlike Fortran 77

Dissection(2)

- Floating point variables in Fortran
  - REAL: single precision
  - DOUBLE PRECISION: double precision
- a2d: your own ascii string to double conversion function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself or you can use the C/C++ atof() if you declare it as external real function
- Automatic type conversion: DOUBLE PRECISION = REAL
- The SIN() function is an intrinsic function and does not need a specific declaration

Dissection(2)

- Floating point variables in Fortran
  - REAL: single precision
  - DOUBLE PRECISION: double precision
- a2d: your own ascii string to double conversion function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself or you can use the C/C++ atof() if you declare it as external real function
- Automatic type conversion: DOUBLE PRECISION = REAL
- The SIN() function is an intrinsic function and does not need a specific declaration

Dissection(2)

- Floating point variables in Fortran
  - REAL: single precision
  - DOUBLE PRECISION: double precision
- a2d: your own ascii string to double conversion function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself or you can use the C/C++ atof() if you declare it as external real function
- Automatic type conversion: DOUBLE PRECISION = REAL
- The SIN() function is an intrinsic function and does not need a specific declaration

Dissection(2)

- Floating point variables in Fortran
  - REAL: single precision
  - DOUBLE PRECISION: double precision
- a2d: your own ascii string to double conversion function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself or you can use the C/C++ atof() if you declare it as external real function
- Automatic type conversion: DOUBLE PRECISION = REAL
- The SIN() function is an intrinsic function and does not need a specific declaration

Dissection(2)

- Floating point variables in Fortran
  - REAL: single precision
  - DOUBLE PRECISION: double precision
- a2d: your own ascii string to double conversion function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself or you can use the C/C++ atof() if you declare it as external real function
- Automatic type conversion: DOUBLE PRECISION = REAL
- The SIN() function is an intrinsic function and does not need a specific declaration

An interactive version

- Let us ask the user for the real number instead of reading it from the command line

```
WRITE(*.FMT='(A)',ADVANCE='NO') 'Give a number: '
READ(*,*) r
s = SIN(r)
! etc.
```

- The keyword *ADVANCE='NO'* suppress the linefeed Fortran put at the end of each *WRITE* statement

Scientific Hello World in Fortran 77

- The f77 code

```
C234567
      PROGRAM hw1
        REAL*8 r,s
        CHARACTER*80 argv
        CALL getarg(1,argv)
        r = a2d(argv)
        s = SIN(r)
        WRITE(*,*)'Hello World! sin(',r')=',s
      END PROGRAM hw1
```

Differences from the Fortran 95 version

- Fortran 77 uses REAL*8 instead of DOUBLE PRECISION
- Fortran 77 lacks IMPLICIT NONE directive
- A double precision variable has to be declared in Fortran 77 since default real numbers are single precision

Differences from the Fortran 95 version

- Fortran 77 uses REAL*8 instead of DOUBLE PRECISION
- Fortran 77 lacks IMPLICIT NONE directive
- A double precision variable has to be declared in Fortran 77 since default real numbers are single precision

Differences from the Fortran 95 version

- Fortran 77 uses REAL*8 instead of DOUBLE PRECISION
- Fortran 77 lacks IMPLICIT NONE directive
- A double precision variable has to be declared in Fortran 77 since default real numbers are single precision

List of Topics

How to compile and link (Fortran 95) on a unix/linux system

- One step (compiling and linking):

```
unix> f90 -O3 -o hw1 hw1.f90
```

- Two steps:

```
unix> f90  -O3 -c hw1.f90
unix> f90 -O3 -o hw1 hw1.o
```

- A linux system with Intel Fortran Compiler:

```
linux> ifort -O3 -o hw1 hw1.f90
```

How to compile and link (Fortran 95) on a unix/linux system

- One step (compiling and linking):

```
unix> f90 -O3 -o hw1 hw1.f90
```

- Two steps:

```
unix> f90  -O3 -c hw1.f90
unix> f90 -O3 -o hw1 hw1.o
```

- A linux system with Intel Fortran Compiler:

```
linux> ifort -O3 -o hw1 hw1.f90
```

How to compile and link (Fortran 95) on a unix/linux system

- One step (compiling and linking):

```
unix> f90 -O3 -o hw1 hw1.f90
```

- Two steps:

```
unix> f90  -O3 -c hw1.f90
unix> f90 -O3 -o hw1 hw1.o
```

- A linux system with Intel Fortran Compiler:

```
linux> ifort -O3 -o hw1 hw1.f90
```

How to compile and link (Fortran 95) on a windows system

- On a windows system one usually uses an Integrated Development Environment (*IDE*)
- This *IDE* contains drop down menus to compile and run the program
- An integrated debugger is also available in such an *IDE*

How to compile and link (Fortran 95) on a windows system

- On a windows system one usually uses an Integrated Development Environment (*IDE*)
- This *IDE* contains drop down menus to compile and run the program
- An integrated debugger is also available in such an *IDE*

How to compile and link (Fortran 95) on a windows system

- On a windows system one usually uses an Integrated Development Environment (*IDE*)
- This *IDE* contains drop down menus to compile and run the program
- An integrated debugger is also available in such an *IDE*

Using the make utility to compile a program

- What is the make utility?
  - The make utility reads a file containing the name(s) of the file(s) to be compiled togehter with the name of the executable program
- The makefile is either called makefile or Makefile as default and is available on all unix/linux systems
- Invoking the make utiltity:

```
linux> make
unix> make
unix> gmake
```

- On unix machines there are often both the gnu make utility and a native make. The native make utility can often have a different syntax than the gnu make

Using the make utility to compile a program

- What is the make utility?
  - The make utility reads a file containing the name(s) of the file(s) to be compiled togehter with the name of the executable program
- The makefile is either called makefile or Makefile as default and is available on all unix/linux systems
- Invoking the make utiltity:

```
linux> make
unix> make
unix> gmake
```

- On unix machines there are often both the gnu make utility and a native make. The native make utility can often have a different syntax than the gnu make

Using the make utility to compile a program

- What is the make utility?
  - The make utility reads a file containing the name(s) of the file(s) to be compiled togehter with the name of the executable program
- The makefile is either called makefile or Makefile as default and is available on all unix/linux systems
- Invoking the make utiltity:

```
linux> make
unix> make
unix> gmake
```

- On unix machines there are often both the gnu make utility and a native make. The native make utility can often have a different syntax than the gnu make

Using the make utility to compile a program

- What is the make utility?
  - The make utility reads a file containing the name(s) of the file(s) to be compiled togehter with the name of the executable program
- The makefile is either called makefile or Makefile as default and is available on all unix/linux systems
- Invoking the make utiltity:

```
linux> make
unix> make
unix> gmake
```

- On unix machines there are often both the gnu make utility and a native make. The native make utility can often have a different syntax than the gnu make

Using the make utility to compile a program

- What is the make utility?
  - The make utility reads a file containing the name(s) of the file(s) to be compiled togehter with the name of the executable program
- The makefile is either called makefile or Makefile as default and is available on all unix/linux systems
- Invoking the make utiltity:

```
linux> make
unix> make
unix> gmake
```

- On unix machines there are often both the gnu make utility and a native make. The native make utility can often have a different syntax than the gnu make

A short example of a makefile

- Makefile

```
$(shell ls *.f90 ./srclist)
SRC=$(shell cat ./srclist)
OBJECTS= $(SRC:.f90=.o)
prog : $(OBJECTS)
	$(FC) -o $@ $(OBJECTS)
%.o : %.f90
	$(FC) -c $?
```

Rolling your own make script

- The main feature of a makefile is to check time stamps in files and only recompile the required files
- Since the syntax of a makefile is kind of awkward and each flavour of unix has its own specialities you can make your own script doing almost the same

Rolling your own make script

- The main feature of a makefile is to check time stamps in files and only recompile the required files
- Since the syntax of a makefile is kind of awkward and each flavour of unix has its own specialities you can make your own script doing almost the same

The looks of a make.sh script(1)

- make.sh

```
#!/bin/sh
if [ ! -n ``$F90_CMPL'' ]; then
  case `uname -s` in
    Linux)
       F90_CMPL=ifort
       F90_OPTS= -static
       ;;
    *)
       F90_CMPL=f90
       F90_OPTS=
  esac
fi
```

The looks of the make.sh script(2)

- make.sh

```
files=‘/bin/ls *.f90‘
for file in files; do
 stem=‘echo $file | sed ’s/\.f90//’‘
 echo $F90_CMPL $F90_OPTS -I. -o $stem $file
 $F90_CMPL $F90_OPTS -I. -o $stem $file
 ls -s stem
```

How to compile and link (Fortran 77)(1)

- Either use the f90 compiler or if present the f77 compiler
- Rememeber that Fortran 77 is s subset of Fortran 95
- An example:

```
f90 -o prog prog.f or
f77 -o prog prog.f
```

How to compile and link (Fortran 77)(1)

- Either use the f90 compiler or if present the f77 compiler
- Rememeber that Fortran 77 is s subset of Fortran 95
- An example:

```
f90 -o prog prog.f or
f77 -o prog prog.f
```

How to compile and link (Fortran 77)(1)

- Either use the f90 compiler or if present the f77 compiler
- Rememeber that Fortran 77 is s subset of Fortran 95
- An example:

```
f90 -o prog prog.f or
f77 -o prog prog.f
```

How to compile and link (Fortran 77) (2)

- NOTE! The file extension of the Fortran source code is important
- A file with the extension *.f90* is automatically a Fortran 90/95 free format file
- If the file has the extension *.f* the compier sees this as a Fortran 77 fixed format file

How to compile and link (Fortran 77) (2)

- NOTE! The file extension of the Fortran source code is important
- A file with the extension *.f90* is automatically a Fortran 90/95 free format file
- If the file has the extension *.f* the compier sees this as a Fortran 77 fixed format file

How to compile and link (Fortran 77) (2)

- NOTE! The file extension of the Fortran source code is important
- A file with the extension *.f90* is automatically a Fortran 90/95 free format file
- If the file has the extension *.f* the compier sees this as a Fortran 77 fixed format file

How to compile and link in general

- We compile a set of programs in Fortran and C++
- Compile each set of files with the right compiler:

```
unix> f90 -O3 -c *.f90
unix> g++ -O3 -c *.cpp
```

- Then link:

```
unix> f90 -o exec_file  *.o -L/some/libdir \
      -L/other/libdir -lmylib -lyourlib
```

- Library type: lib*.a: static; lib*.so: dynamic

How to compile and link in general

- We compile a set of programs in Fortran and C++
- Compile each set of files with the right compiler:

```
unix> f90 -O3 -c *.f90
unix> g++ -O3 -c *.cpp
```

- Then link:

```
unix> f90 -o exec_file  *.o -L/some/libdir \
      -L/other/libdir -lmylib -lyourlib
```

- Library type: lib*.a: static; lib*.so: dynamic

How to compile and link in general

- We compile a set of programs in Fortran and C++
- Compile each set of files with the right compiler:

```
unix> f90 -O3 -c *.f90
unix> g++ -O3 -c *.cpp
```

- Then link:

```
unix> f90 -o exec_file  *.o -L/some/libdir \
      -L/other/libdir -lmylib -lyourlib
```

- Library type: lib*.a: static; lib*.so: dynamic

How to compile and link in general

- We compile a set of programs in Fortran and C++
- Compile each set of files with the right compiler:

```
unix> f90 -O3 -c *.f90
unix> g++ -O3 -c *.cpp
```

- Then link:

```
unix> f90 -o exec_file  *.o -L/some/libdir \
      -L/other/libdir -lmylib -lyourlib
```

- Library type: lib*.a: static; lib*.so: dynamic

List of Topics

Example: Data transformation

- Suppose we have a file with an xy-data pair

```
0.1 1.1
0.2 1.8
0.3 2.2
0.4 1.8
```

- We want to transform the y value using some mathematical function f(y)
- Goal: write a Fortran 95 program that reads the xy-data pair from the file, transforms the y value and write the new xy-data pair to a new file

Example: Data transformation

- Suppose we have a file with an xy-data pair

```
0.1 1.1
0.2 1.8
0.3 2.2
0.4 1.8
```

- We want to transform the y value using some mathematical function f(y)
- Goal: write a Fortran 95 program that reads the xy-data pair from the file, transforms the y value and write the new xy-data pair to a new file

Example: Data transformation

- Suppose we have a file with an xy-data pair

```
0.1 1.1
0.2 1.8
0.3 2.2
0.4 1.8
```

- We want to transform the y value using some mathematical function f(y)
- Goal: write a Fortran 95 program that reads the xy-data pair from the file, transforms the y value and write the new xy-data pair to a new file

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
- Close the files

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
  - read x and y from the input file
  - set y=myfunc(y)
  - write x and y to the output file
- Close the files

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
  - read x and y from the input file
  - set y=myfunc(y)
  - write x and y to the output file
- Close the files

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
  - read x and y from the input file
  - set y=myfunc(y)
  - write x and y to the output file
- Close the files

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
  - read x and y from the input file
  - set y=myfunc(y)
  - write x and y to the output file
- Close the files

Program structure

- Read the names of input and output files as command-line arguments
- Print error/usage message if less than two command-line arguments are given
- Open the files
- While more data in the file:
    - read x and y from the input file
    - set y=myfunc(y)
    - write x and y to the output file
- Close the files

The fortran 95 code(1)

- Code

```
FUNCTION myfunc(y) RESULT(r)
   IMPLICIT NONE
   DOUBLE PRECISION, INTENT(IN) :: y
   DOUBLE PRECISION             :: r
   IF(y>=0.) THEN
      r = y**0.5*EXP(-y)
   ELSE
      r = 0.
   END IF
END FUNCTION myfunc
```

The fortran 95 code(2)

- Code

```
PROGRAM dtrans
   IMPLICIT NONE
   INTEGER                :: argc, rstat
   DOUBLE PRECISION       :: x, y
   CHARACTER(LEN=80)      :: infilename, outfilename
   INTEGER,PARAMETER      :: ilun = 10
   INTEGER,PARAMETER      :: olun = 11
   INTEGER, EXTERNAL      :: iargc
   argc = iargc()
   IF (argc < 2) THEN
      PRINT *, 'Usage: dtrans infile outfile'
      STOP
   END IF
   CALL getarg(1,infilename)
   CALL getarg(2,outfilename)
```

The fortran 95 code(3)

- Code

```
OPEN(UNIT=ilun,FILE=infilename, &
  FORM='FORMATTED', IOSTAT=rstat)
  OPEN(UNIT=olun,FILE=outfilename,&
  FORM='FORMATTED', IOSTAT=rstat)
  rstat = 0
  DO WHILE(rstat == 0)
    READ(UNIT=ilun,FMT='(F3.1,X,F3.1)',&
         IOSTAT=rstat) x, y
    IF(rstat /= 0) THEN
      CLOSE(ilun); CLOSE(olun)
      STOP
    END IF
    y = myfunc(y)
    WRITE(UNIT=olun,FMT='(F3.1,X,F3.1)',&
          IOSTAT=rstat) x, y
  END DO
END PROGRAM dtrans
```

List of Topics

Fortran file opening

- Open a file for reading

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open a file for writing

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open for appending data

```
OPEN(UNIT=ilun,FORM='FORMATTED',&
     POSITION='APPEND',IOSTAT=rstat)
```

Fortran file opening

- Open a file for reading

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open a file for writing

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open for appending data

```
OPEN(UNIT=ilun,FORM='FORMATTED',&
     POSITION='APPEND',IOSTAT=rstat)
```

Fortran file opening

- Open a file for reading

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open a file for writing

```
OPEN(UNIT=ilun,FORM='FORMATTED',IOSTAT=rstat)
```

- Open for appending data

```
OPEN(UNIT=ilun,FORM='FORMATTED',&
     POSITION='APPEND',IOSTAT=rstat)
```

Fortran file reading and writing

- Read a double precision number

```
READ(UNIT=ilun,FMT='(F10.6)',IOSTAT=rstat) x
```

- Test if the reading was successful

```
IF(rstat /= 0) STOP
```

- Write a double precision number

```
WRITE(UNIT=olun,FMT='(F20.12)',IOSTAT=rstat) x
```

Fortran file reading and writing

- Read a double precision number

```
READ(UNIT=ilun,FMT='(F10.6)',IOSTAT=rstat) x
```

- Test if the reading was successful

```
IF(rstat /= 0) STOP
```

- Write a double precision number

```
WRITE(UNIT=olun,FMT='(F20.12)',IOSTAT=rstat) x
```

Fortran file reading and writing

- Read a double precision number

```
READ(UNIT=ilun,FMT='(F10.6)',IOSTAT=rstat) x
```

- Test if the reading was successful

```
IF(rstat /= 0) STOP
```

- Write a double precision number

```
WRITE(UNIT=olun,FMT='(F20.12)',IOSTAT=rstat) x
```

Formatted output

- The formatted output in Fortran is selected via the FORMAT of FMT statement
- In fortran 77 the FORMAT statement is used

```
C234567
 100  FORMAT(F15.8)
      WRITE(*,100) x
```

- In Fortran 95 the FMT statement is used

```
WRITE(*,FMT='(F15.8)') x
```

Formatted output

- The formatted output in Fortran is selected via the FORMAT of FMT statement
- In fortran 77 the FORMAT statement is used

```
C234567
 100   FORMAT(F15.8)
       WRITE(*,100) x
```

- In Fortran 95 the FMT statement is used

```
WRITE(*,FMT='(F15.8)') x
```

Formatted output

- The formatted output in Fortran is selected via the FORMAT of FMT statement
- In fortran 77 the FORMAT statement is used

```
C234567
 100  FORMAT(F15.8)
      WRITE(*,100) x
```

- In Fortran 95 the FMT statement is used

```
WRITE(*,FMT='(F15.8)') x
```

A convenient way of formatting in Fortran 95(1)

- Instead of writing the format in the FMT statement we can put it in a string variable

```
CHARACTER(LEN=7)  :: fmt_string
fmt_string = '(F15.8)'
WRITE(*,FMT=fmt_string) x
```

A convenient way of formatting in Fortran 95(2)

- We can use a set of such format strings

```
CHARACTER(LEN=7),DIMENSION(3)  :: fmt_string
fmt_string(1) = '(F15.8)'
fmt_string(2) = '(2I4)'
fmt_string(3) = '(3F10.2)'
WRITE(*,FMT=fmt_string(1)) x
```

Unformatted I/O in Fortran

- More often than not we use huge amount of data both for input and output
- Using formatted data increase both the filesize and the time spent reading and writing data from/to files
- We therefore use unformatted data in these cases

Unformatted I/O in Fortran

- More often than not we use huge amount of data both for input and output
- Using formatted data increase both the filesize and the time spent reading and writing data from/to files
- We therefore use unformatted data in these cases

Unformatted I/O in Fortran

- More often than not we use huge amount of data both for input and output
- Using formatted data increase both the filesize and the time spent reading and writing data from/to files
- We therefore use unformatted data in these cases

Opening and reading an unformatted file

- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,FORM='UNFORMATTED',&
     IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,IOSTAT=rstat) array
```

- the array variable can be a single variable, a vector or a multidimensional matrix

Opening and reading an unformatted file

- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,FORM='UNFORMATTED',&
     IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,IOSTAT=rstat) array
```

- the array variable can be a single variable, a vector or a multidimensional matrix

Opening and reading an unformatted file

- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,FORM='UNFORMATTED',&
     IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,IOSTAT=rstat) array
```

- the array variable can be a single variable, a vector or a multidimensional matrix

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

Using direct access file I/O

- In some cases it is advantageous to be able to read and write the same portion of a file without reading it sequentially from start
- This is performed by using direct access file I/O
- Open syntax:

```
OPEN(UNIT=ilun,FILE=infile,ACCESS='DIRECT',&
     RECL=lng,IOSTAT=rstat)
```

- Reading syntax:

```
READ(UNIT=ilun,REC=recno,IOSTAT=rstat) array
```

- The array most be of equal size to the record length and the recno variable contains the record number to be read
- The records are numbered from 1 and up

The namelist file

- A special type of file exists in Fortran 95
- It is the namelist file which is used for input of data mainly for initializing purposes
- Reading syntax:

```
INTEGER :: i, j, k
NAMELIST/index/i, j, k
READ(UNIT=ilun,NML=index,IOSTAT=rstat)
```

- This will read from the namelist file values into the variables i, j, k

The namelist file

- A special type of file exists in Fortran 95
- It is the namelist file which is used for input of data mainly for initializing purposes
- Reading syntax:

```
INTEGER :: i, j, k
NAMELIST/index/i, j, k
READ(UNIT=ilun,NML=index,IOSTAT=rstat)
```

- This will read from the namelist file values into the variables i, j, k

The namelist file

- A special type of file exists in Fortran 95
- It is the namelist file which is used for input of data mainly for initializing purposes
- Reading syntax:

```
INTEGER :: i, j, k
NAMELIST/index/i, j, k
READ(UNIT=ilun,NML=index,IOSTAT=rstat)
```

- This will read from the namelist file values into the variables i, j, k

The namelist file

- A special type of file exists in Fortran 95
- It is the namelist file which is used for input of data mainly for initializing purposes
- Reading syntax:

```
INTEGER :: i, j, k
NAMELIST/index/i, j, k
READ(UNIT=ilun,NML=index,IOSTAT=rstat)
```

- This will read from the namelist file values into the variables i, j, k

The contents of a namelist file

- Namelist file syntax:

```
&index i=10, j=20, k=4 /
```

- A namelist file can contain more than one namelist

The contents of a namelist file

- Namelist file syntax:

```
&index i=10, j=20, k=4 /
```

- A namelist file can contain more than one namelist

List of Topics

Matrix-vector product

- Goal: calculate a matrix-vector product
  - Make a simple example with known solution (simplifies debugging)
  - Declare a matrix A and vectors x and b
  - Initialize A
  - Perform $b = A * x$
  - Check that b is correct

Matrix-vector product

- Goal: calculate a matrix-vector product
  - Make a simple example with known solution (simplifies debugging)
  - Declare a matrix A and vectors x and b
  - Initialize A
  - Perform $b = A * x$
  - Check that b is correct

Basic arrays in Fortran

- Fortran 77 and 95 uses the same basic array construction
- Array indexing follows a quickly learned syntax:

  ```
  q(3,2)
  ```

- This is the same as in Matlab. Note that in C/C++ a multidimensional array is transposed

Basic arrays in Fortran

- Fortran 77 and 95 uses the same basic array construction
- Array indexing follows a quickly learned syntax:

```
q(3,2)
```

- This is the same as in Matlab. Note that in C/C++ a multidimensional array is transposed

Basic arrays in Fortran

- Fortran 77 and 95 uses the same basic array construction
- Array indexing follows a quickly learned syntax:

```
q(3,2)
```

- This is the same as in Matlab. Note that in C/C++ a multidimensional array is transposed

Declaring basic vectors

- Declaring a fixed size vector

```
INTEGER, PARAMETER                :: n = 100
DOUBLE PRECISION, DIMENSION(n)    :: x
DOUBLE PRECISION, DIMENSION(50)   :: b
```

- Vector indices starts at 1 not 0 like C/C++

Declaring basic vectors

- Declaring a fixed size vector

```
INTEGER, PARAMETER                 :: n = 100
DOUBLE PRECISION, DIMENSION(n)     :: x
DOUBLE PRECISION, DIMENSION(50)    :: b
```

- Vector indices starts at 1 not 0 like C/C++

Declaring basic matrices

- Declaring a fixed size matrix

```
INTEGER, PARAMETER                  :: m = 100
INTEGER, PARAMETER                  :: n = 100
DOUBLE PRECISION, DIMENSION(m,n)    :: x
```

- Matrix indices starts at 1 not 0 like C/C++

Declaring basic matrices

- Declaring a fixed size matrix

```
INTEGER, PARAMETER                    :: m = 100
INTEGER, PARAMETER                    :: n = 100
DOUBLE PRECISION, DIMENSION(m,n) :: x
```

- Matrix indices starts at 1 not 0 like C/C++

Looping over the matrix

- A nested loop

```
INTEGER    :: i, j
DO j = 1, n
  DO i = 1, n
    A(i,j) = f(i,j) + 3.14
  END DO
END DO
```

Matrix storage scheme

- Note: matices in fortran are stored column wise; the row index should vary fastest
- Recall that in C/C++ matrices are stored row by row and the column index should vary fastest
- Typical loop in C/C++ (2nd index in inner loop):

```
DO i = 1, m
  DO j = 1, n
    A(i,j) = f(i,j) + 3.14
  END DO
END DO
```

- We now traverse A in jumps

Matrix storage scheme

- Note: matices in fortran are stored column wise; the row index should vary fastest
- Recall that in C/C++ matrices are stored row by row and the column index should vary fastest
- Typical loop in C/C++ (2nd index in inner loop):

```
DO i = 1, m
  DO j = 1, n
    A(i,j) = f(i,j) + 3.14
  END DO
END DO
```

- We now traverse A in jumps

Matrix storage scheme

- Note: matices in fortran are stored column wise; the row index should vary fastest
- Recall that in C/C++ matrices are stored row by row and the column index should vary fastest
- Typical loop in C/C++ (2nd index in inner loop):

```
DO i = 1, m
  DO j = 1, n
    A(i,j) = f(i,j) + 3.14
  END DO
END DO
```

- We now traverse A in jumps

Matrix storage scheme

- Note: matices in fortran are stored column wise; the row index should vary fastest
- Recall that in C/C++ matrices are stored row by row and the column index should vary fastest
- Typical loop in C/C++ (2nd index in inner loop):

```
DO i = 1, m
  DO j = 1, n
    A(i,j) = f(i,j) + 3.14
  END DO
END DO
```

- We now traverse A in jumps

Dynamic memory allocation

- Very often we do not know the length of the array in advance
- By using dynamic memory allocation we can allocate the necessary chunk of memory at runtime
- You need to allocate and deallocate memory

Dynamic memory allocation

- Very often we do not know the length of the array in advance
- By using dynamic memory allocation we can allocate the necessary chunk of memory at runtime
- You need to allocate and deallocate memory

Dynamic memory allocation

- Very often we do not know the length of the array in advance
- By using dynamic memory allocation we can allocate the necessary chunk of memory at runtime
- You need to allocate and deallocate memory

Dynamic memeory allocation in Fortran 95

- There are two ways of declaring allocatable matrices in Fortran 95
- Using the ALLOCATABLE attribute
- Using a POINTER variable

Dynamic memeory allocation in Fortran 95

- There are two ways of declaring allocatable matrices in Fortran 95
- Using the ALLOCATABLE attribute
- Using a POINTER variable

Dynamic memeory allocation in Fortran 95

- There are two ways of declaring allocatable matrices in Fortran 95
- Using the ALLOCATABLE attribute
- Using a POINTER variable

Allocating memory using the ALLOCATABLE attribute

- Declare an ALLOCATABLE array variable

```
DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: x
ALLOCATE(x(100))
  ...
DEALLOCATE(x)
```

Allocating memory using a POINTER

- Declare a pointer array variable

```
DOUBLE PRECISION, POINTER  :: x(:)
ALLOCATE(x(100))
  ...
DEALLOCATE(x)
```

- Keep in mind that a Fortran 95 POINTER is not the same as a pointer in C/C++

Allocating memory using a POINTER

- Declare a pointer array variable

```
DOUBLE PRECISION, POINTER  :: x(:)
ALLOCATE(x(100))
  ...
DEALLOCATE(x)
```

- Keep in mind that a Fortran 95 POINTER is not the same as a pointer in C/C++

Declaring and initializing A, x and b

- Code

```
DOUBLE PRECISION, POINTER :: A(:,:), x(:), b(:)
CHARACTER(LEN=20)         :: str
INTEGER                   :: n, i, j
CALL getarg(1,str)
n = a2i(str)
ALLOCATE(A(n,n)); ALLOCATE(x(n))
ALLOCATE(b(n))
DO j = 1, n
  x(j) = j/2.
  DO i = 1, n
    A(i,j) = 2. + i/j
  END DO
END DO
```

Matrix-vector product loop

- Code for computation of the matrix-vector product

```
DOUBLE PRECISION          :: sum
DO j = 1, n
  sum = 0.
  DO i = 1, n
    sum = sum + A(i,j) * x(i)
  END DO
  b(j) = sum
END DO
```

- Another way to compute eth matrix-vector product is to use an intrinsic fortran function *MATMUL*

Matrix-vector product loop

- Code for computation of the matrix-vector product

```
DOUBLE PRECISION          :: sum
DO j = 1, n
  sum = 0.
  DO i = 1, n
    sum = sum + A(i,j) * x(i)
  END DO
  b(j) = sum
END DO
```

- Another way to compute eth matrix-vector product is to use an intrinsic fortran function *MATMUL*

List of Topics

Subroutines

- A subroutine does not return any value and is the same as a void function in C/C++
- In Fortran all arguments are passed as the address of the variable in the calling program
- This is the same as a call by refrence in C++
- It is easy to for a C++ programmer to forget this and then accidentally change the contents of the variable in the calling program

Subroutines

- A subroutine does not return any value and is the same as a void function in C/C++
- In Fortran all arguments are passed as the address of the variable in the calling program
- This is the same as a call by refrence in C++
- It is easy to for a C++ programmer to forget this and then accidentally change the contents of the variable in the calling program

Subroutines

- A subroutine does not return any value and is the same as a void function in C/C++
- In Fortran all arguments are passed as the address of the variable in the calling program
- This is the same as a call by refrence in C++
- It is easy to for a C++ programmer to forget this and then accidentally change the contents of the variable in the calling program

Subroutines

- A subroutine does not return any value and is the same as a void function in C/C++
- In Fortran all arguments are passed as the address of the variable in the calling program
- This is the same as a call by refrence in C++
- It is easy to for a C++ programmer to forget this and then accidentally change the contents of the variable in the calling program

An example of a subroutine

- This subroutine will calculate the square root of two arguments and returning the sum of the results in a third argument

```
SUBROUTINE dsquare(x,y,z)
  DOUBLE PRECISION, INTENT(IN)  :: x, y
  DOUBLE PRECISION, INTENT(OUT) :: z
  z = SQRT(x) + SQRT(y)
END SUBROUTINE dsquare
```

- Using the INTENT(IN) and INTENT(OUT) will prevent any accidentally changes of the variable(s) in the calling program by flagging an error at compile time

An example of a subroutine

- This subroutine will calculate the square root of two arguments and returning the sum of the results in a third argument

```
SUBROUTINE dsquare(x,y,z)
  DOUBLE PRECISION, INTENT(IN)  :: x, y
  DOUBLE PRECISION, INTENT(OUT) :: z
  z = SQRT(x) + SQRT(y)
END SUBROUTINE dsquare
```

- Using the INTENT(IN) and INTENT(OUT) will prevent any accidentally changes of the variable(s) in the calling program by flagging an error at compile time

Functions

- A function always return a value just like corresponding functions in C/C++
- The syntax of the function statement can be written in two ways depending on the fortran version
- In Fortran 77 it looks like a corresponding C++ function
- But in fortran 95 another syntax has been introduced although both versions can be used in Fortran 95

Functions

- A function always return a value just like corresponding functions in C/C++
- The syntax of the function statement can be written in two ways depending on the fortran version
- In Fortran 77 it looks like a corresponding C++ function
- But in fortran 95 another syntax has been introduced although both versions can be used in Fortran 95

Functions

- A function always return a value just like corresponding functions in C/C++
- The syntax of the function statement can be written in two ways depending on the fortran version
- In Fortran 77 it looks like a corresponding C++ function
- But in fortran 95 another syntax has been introduced although both versions can be used in Fortran 95

Functions

- A function always return a value just like corresponding functions in C/C++
- The syntax of the function statement can be written in two ways depending on the fortran version
- In Fortran 77 it looks like a corresponding C++ function
- But in fortran 95 another syntax has been introduced although both versions can be used in Fortran 95

An example of a function in Fortran 77 style

- This function will calculate the square root of two arguments and returning the sum of the results

```
C234567
      DOUBLE PRECISION, FUNCTION dsquare(x,y)
        DOUBLE PRECISION, INTENT(IN)  :: x, y
        DOUBLE PRECISION              :: z
        z = SQRT(x) + SQRT(y)
        dsquare = z
      END FUNCTION dsquare
```

An example of a function in Fortran 95 style

- This function will calculate the square root of two arguments and returning the sum of the results

```
FUNCTION dsquare(x,y), RESULT(z)
  DOUBLE PRECISION, INTENT(IN)  :: x, y
  DOUBLE PRECISION              :: z
  z = SQRT(x) + SQRT(y)
END FUNCTION dsquare
```

- It is the variable type in the RESULT statement that identifies the type of the function

An example of a function in Fortran 95 style

- This function will calculate the square root of two arguments and returning the sum of the results

```
FUNCTION dsquare(x,y), RESULT(z)
  DOUBLE PRECISION, INTENT(IN)  :: x, y
  DOUBLE PRECISION              :: z
  z = SQRT(x) + SQRT(y)
END FUNCTION dsquare
```

- It is the variable type in the RESULT statement that identifies the type of the function

List of Topics

More about pointers in Fortran 95

- As mentioned earlier a pointer in Fortran 95 *IS NOT* the same as a pointer in C/C++
- A fortran 95 pointer is used as an alias pointing to another variable, it can be a single variable, a vector or a multidimensional array
- A pointer must be associated with a target variable or another pointer and have the same shape that the target it is pointing to
- Or the pointer can have memory allocated and be treated as a regular variable or array

More about pointers in Fortran 95

- As mentioned earlier a pointer in Fortran 95 *IS NOT* the same as a pointer in C/C++
- A fortran 95 pointer is used as an alias pointing to another variable, it can be a single variable, a vector or a multidimensional array
- A pointer must be associated with a target variable or another pointer and have the same shape that the target it is pointing to
- Or the pointer can have memory allocated and be treated as a regular variable or array

More about pointers in Fortran 95

- As mentioned earlier a pointer in Fortran 95 *IS NOT* the same as a pointer in C/C++
- A fortran 95 pointer is used as an alias pointing to another variable, it can be a single variable, a vector or a multidimensional array
- A pointer must be associated with a target variable or another pointer and have the same shape that the target it is pointing to
- Or the pointer can have memory allocated and be treated as a regular variable or array

More about pointers in Fortran 95

- As mentioned earlier a pointer in Fortran 95 *IS NOT* the same as a pointer in C/C++
- A fortran 95 pointer is used as an alias pointing to another variable, it can be a single variable, a vector or a multidimensional array
- A pointer must be associated with a target variable or another pointer and have the same shape that the target it is pointing to
- Or the pointer can have memory allocated and be treated as a regular variable or array

Some examples of pointer usage(1)

- A target pointer example

```
DOUBLE PRECISION, TARGET, DIMENSION(100) :: x
DOUBLE PRECISION, POINTER                :: y(:)
 ...
y => x
 ...
y => x(20:80)
 ...
y => x(1:33)
NULLIFY(y)
```

Some examples of pointer usage(2)

- What happens when we try to access a deallocated array?

```
PROGRAM ptr
   IMPLICIT NONE
   DOUBLE PRECISION, POINTER :: x(:)
   DOUBLE PRECISION, POINTER :: y(:)
   ALLOCATE(x(100))
   x = 0.
   x(12:19) = 3.14
   y => x(10:20)
   PRINT '(A,3F10.4)', 'Y-value ', y(1:3)
   y => x(11:14)
   DEALLOCATE(x)
   PRINT '(A,3F10.4)', 'Y-value ', y(1:3)
   PRINT '(A,4F10.4)', 'X-value ', x(11:14)
END PROGRAM ptr
```

Some examples of pointer usage(3)

- This is what happened

```
bullet.uio.no$ EXAMPLES/ptr
    0.0000     0.0000     3.1400
    0.0000     3.1400     3.1400
forrtl: severe (174): SIGSEGV,
segmentation fault occurred
```

- When we try to access the x-array in the last PRINT statement we get an segmentation fault
- This means we try to access a variable which is not associated with any part of the memory the program has access to

Some examples of pointer usage(3)

- This is what happened

```
bullet.uio.no$ EXAMPLES/ptr
    0.0000    0.0000    3.1400
    0.0000    3.1400    3.1400
forrtl: severe (174): SIGSEGV,
segmentation fault occurred
```

- When we try to access the x-array in the last PRINT statement we get an segmentation fault
- This means we try to access a variable which is not associated with any part of the memory the program has access to

Some examples of pointer usage(3)

- This is what happened

```
bullet.uio.no$ EXAMPLES/ptr
    0.0000    0.0000    3.1400
    0.0000    3.1400    3.1400
forrtl: severe (174): SIGSEGV,
segmentation fault occurred
```

- When we try to access the x-array in the last PRINT statement we get an segmentation fault
- This means we try to access a variable which is not associated with any part of the memory the program has access to

Some examples of pointer usage(4)

- In our little example we clearly see that the memory pointed to by the x-array is no longer available
- On the other hand the part of the memory the y-array is pointing to is still available
- To free the last part of memory the y-array refers to we must nullify the y-array:

```
NULLIFY(y)
```

Some examples of pointer usage(4)

- In our little example we clearly see that the memory pointed to by the x-array is no longer available
- On the other hand the part of the memory the y-array is pointing to is still available
- To free the last part of memory the y-array refers to we must nullify the y-array:

```
NULLIFY(y)
```

Some examples of pointer usage(4)

- In our little example we clearly see that the memory pointed to by the x-array is no longer available
- On the other hand the part of the memory the y-array is pointing to is still available
- To free the last part of memory the y-array refers to we must nullify the y-array:

```
NULLIFY(y)
```

List of Topics

Exercise1: Modify the Fortran 95 Hello World program

- Locate the Hello World program
- Compile the program and test it
- Modification: write "Hello World!" and format it so the text and numbers are without unnecessary spaces and trailing zeroes

```
Hello World sin(    2.3000000000000      )=   0.7457052121
```

- unlike it is in this printout

Exercise1: Modify the Fortran 95 Hello World program

- Locate the Hello World program
- Compile the program and test it
- Modification: write “Hello World!” and format it so the text and numbers are without unnecessary spaces and trailing zeroes

```
Hello World sin(   2.3000000000000      )=   0.7457052121
```

- unlike it is in this printout

Exercise1: Modify the Fortran 95 Hello World program

- Locate the Hello World program
- Compile the program and test it
- Modification: write "Hello World!" and format it so the text and numbers are without unnecessary spaces and trailing zeroes

```
Hello World sin(   2.30000000000000      )=  0.7457052121
```

- unlike it is in this printout

Exercise1: Modify the Fortran 95 Hello World program

- Locate the Hello World program
- Compile the program and test it
- Modification: write “Hello World!” and format it so the text and numbers are without unnecessary spaces and trailing zeroes

```
Hello World sin(   2.30000000000000      )=  0.7457052121
```

- unlike it is in this printout

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a “usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a “usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a "usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a “usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a “usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise2: Extend the Fortran 95 Hello World program

- Locate the first Hello World program
- Read the three command-line arguments: *start*, *stop* and *inc*
- Provide a "usage message and abort the program in case there are too few command-line arguments
- Do r = loop_start, loop_stop, loop_inc and compute the sine of r and write the result
- Write and additional loop using DO WHILE construction
- Verify that the program works

Exercise3: Integrate a function(1)

- Write a function

```
DOUBLE PRECISION FUNCTION trapezoidal(f,a,b,n)
  DOUBLE PRECISION, EXTERNAL :: f
  DOUBLE PRECISION           :: a, b
  INTEGER                    :: n
  ...
END FUNCTION trapezoidal
```

- It shall integrate a user-defined function
  $\int_a^b f(x)\,\mathrm{d}x \approx h(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih)), h = \frac{b-a}{n-1}$

Exercise3: Integrate a function(1)

- Write a function

```
DOUBLE PRECISION FUNCTION trapezoidal(f,a,b,n)
  DOUBLE PRECISION, EXTERNAL :: f
  DOUBLE PRECISION           :: a, b
  INTEGER                    :: n
  ...
END FUNCTION trapezoidal
```

- It shall integrate a user-defined function $\int_a^b f(x)\,\mathrm{d}x \approx h(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih)), h = \frac{b-a}{n-1}$

Exercise3: Integrate a function(2)

- The user defined function is specified as *external* in the argument specifications of the trapezoidal function
- Any function taking a double precision as an argument and returning a double precision number can now be used as an input argument to the trapezoidal function
- Verify that *trapeziodal* is implemented correctly

Exercise3: Integrate a function(2)

- The user defined function is specified as *external* in the argument specifications of the trapezoidal function
- Any function taking a double precision as an argument and returning a double precision number can now be used as an input argument to the trapezoidal function
- Verify that *trapeziodal* is implemented correctly

Exercise3: Integrate a function(2)

- The user defined function is specified as *external* in the argument specifications of the trapezoidal function
- Any function taking a double precision as an argument and returning a double precision number can now be used as an input argument to the trapezoidal function
- Verify that *trapeziodal* is implemented correctly

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Binary format

- A number like π can be represented in ASCII format as 3.14 (4 bytes) or $3.14159E+00$ (11 bytes), for instance
- In memory, the number occupies 8 bytes (a *double*), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in a file
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary representation
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and HP Alpha: little endian

Exercise 4: Work with binary data in Fortran 77 (1)

- Scientific simulations often involve large data sets and binary storage of numbers saves space in files
- How to write numbers in binary format in Fortran 77:

```
WRITE(UNIT=olun) array
```

Exercise 4: Work with binary data in Fortran 77 (1)

- Scientific simulations often involve large data sets and binary storage of numbers saves space in files
- How to write numbers in binary format in Fortran 77:

```
WRITE(UNIT=olun) array
```

Exercise 4: Work with binary data in Fortran 77 (2)

- Create datatrans2.f (from datatrans1.f) such that the input and output data are in binary format
- To test the datatrans2.f we need utilities to create and read binary files
  - make a small Fortran 77 program that generates *n* xy-pairs of data and writes them to a file in binary format (read *n* from the command line)
  - make a small Fortran 77 program that reads xy-pairs from a binary file and writes them to the screen
- With these utiltities you can create input data to datatrans2.f and view the file produced by datatrans2.f

Exercise 4: Work with binary data in Fortran 77 (2)

- Create datatrans2.f (from datatrans1.f) such that the input and output data are in binary format
- To test the datatrans2.f we need utilities to create and read binary files
  - make a small Fortran 77 program that generates *n* xy-pairs of data and writes them to a file in binary format (read *n* from the command line)
  - make a small Fortran 77 program that reads xy-pairs from a binary file and writes them to the screen
- With these utiltities you can create input data to datatrans2.f and view the file produced by datatrans2.f

Exercise 4: Work with binary data in Fortran 77 (2)

- Create datatrans2.f (from datatrans1.f) such that the input and output data are in binary format
- To test the datatrans2.f we need utilities to create and read binary files
  - make a small Fortran 77 program that generates *n* xy-pairs of data and writes them to a file in binary format (read *n* from the command line)
  - make a small Fortran 77 program that reads xy-pairs from a binary file and writes them to the screen
- With these utiltities you can create input data to datatrans2.f and view the file produced by datatrans2.f

Exercise 4: Work with binary data in Fortran 77 (2)

- Create datatrans2.f (from datatrans1.f) such that the input and output data are in binary format
- To test the datatrans2.f we need utilities to create and read binary files
    - make a small Fortran 77 program that generates *n* xy-pairs of data and writes them to a file in binary format (read *n* from the command line)
    - make a small Fortran 77 program that reads xy-pairs from a binary file and writes them to the screen
- With these utiltities you can create input data to datatrans2.f and view the file produced by datatrans2.f

Exercise 4: Work with binary data in Fortran 77 (3)

- Modify datatrans2.f program such that the x and y numbers are stored in one long dynamic array
- The storage structure should be x1, y1, x2, y2, ...
- Read and write the array to file in binary format using one READ and one WRITE call
- Try to generate a file with a huge number (10 000 000) of pairs and use the unix *time* command to test the efficiency of reading/writing a single array in one READ/WRITE call compared with reading/writing each number separately

Exercise 4: Work with binary data in Fortran 77 (3)

- Modify datatrans2.f program such that the x and y numbers are stored in one long dynamic array
- The storage structure should be x1, y1, x2, y2, ...
- Read and write the array to file in binary format using one READ and one WRITE call
- Try to generate a file with a huge number (10 000 000) of pairs and use the unix *time* command to test the efficiency of reading/writing a single array in one READ/WRITE call compared with reading/writing each number separately

Exercise 4: Work with binary data in Fortran 77 (3)

- Modify datatrans2.f program such that the x and y numbers are stored in one long dynamic array
- The storage structure should be x1, y1, x2, y2, ...
- Read and write the array to file in binary format using one READ and one WRITE call
- Try to generate a file with a huge number (10 000 000) of pairs and use the unix *time* command to test the efficiency of reading/writing a single array in one READ/WRITE call compared with reading/writing each number separately

Exercise 4: Work with binary data in Fortran 77 (3)

- Modify datatrans2.f program such that the x and y numbers are stored in one long dynamic array
- The storage structure should be x1, y1, x2, y2, ...
- Read and write the array to file in binary format using one READ and one WRITE call
- Try to generate a file with a huge number (10 000 000) of pairs and use the unix *time* command to test the efficiency of reading/writing a single array in one READ/WRITE call compared with reading/writing each number separately

Exercise 5: Work with binary data in Fortran 95

- Do the Fortran 77 version of the exercise first!
- How to write numbers in binary format in Fortran 95

```
WRITE(UNIT=olun, IOSTAT=rstat) array
```

- Modify datatrans1.f90 program such that it works with binary input and output data (use the Fortran 77 utilities in the previous exercise to create input file and view output file)

Exercise 5: Work with binary data in Fortran 95

- Do the Fortran 77 version of the exercise first!
- How to write numbers in binary format in Fortran 95

```
WRITE(UNIT=olun, IOSTAT=rstat) array
```

- Modify datatrans1.f90 program such that it works with binary input and output data (use the Fortran 77 utilities in the previous exercise to create input file and view output file)

Exercise 5: Work with binary data in Fortran 95

- Do the Fortran 77 version of the exercise first!
- How to write numbers in binary format in Fortran 95

```
WRITE(UNIT=olun, IOSTAT=rstat) array
```

- Modify datatrans1.f90 program such that it works with binary input and output data (use the Fortran 77 utilities in the previous exercise to create input file and view output file)

Exercise 6: Efficiency of dynamic memory allocation(1)

- Write this code out in detail as a stand-alone program:

```
INTEGER, PARAMETER  ::  nrepetitions = 1000000
INTEGER             :: i, n
CHARACTER(LEN=80)   :: argv
CALL getarg(1,argv)
n = a2i(argv)
DO i = 1, nrepetitions
  ! allocate a vector of n double precision numbers
  ! set second entry to something
  ! deallocate the vector
END DO
```

Exercise 6: Efficiency of dynamic memory allocation(2)

- Write another program where each vector entry is allocated separately:

```
INTEGER          :: i, j
DOUBLE PRECISION :: sum
DO i = 1, nrepetitions
  ! allocate each of the double precision
  !numbers separately
  DO j = 1, n
    ! allocate a double precision number
    ! add the value of this new item to sum
    ! deallocate the double precision number
  END DO
END DO
```

Exercise 6: Efficiency of dynamic memory allocation(3)

- Measure the CPU time of vector allocation versus allocation of individual entries:

```
unix> time myprog1
unix> time myprog2
```

- Adjust the nrepetitions such that the CPU time of the fastest method is of order 10 seconds

Exercise 6: Efficiency of dynamic memory allocation(3)

- Measure the CPU time of vector allocation versus allocation of individual entries:

```
unix> time myprog1
unix> time myprog2
```

- Adjust the nrepetitions such that the CPU time of the fastest method is of order 10 seconds