Introductory Fortran Programming

Gunnar Wollan

Department of Geosciences
University of Oslo, N-0315 Oslo, Norway

Spring 2005

Motivation

Contents

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 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

Teaching philosophy

Intensive course

- Lectures 9 - 12
- Hands-on training 13 - 16
- Learn form dissecting examples
- Get in touch with the dirty work
- Get some overview of advances topics
- Focus on principles and generic strategies
- Continued learning on individual basis

This course just get you started - use textbooks, reference manuals and software examples from the internet for further work with projects

recommended attidude

- Dive into executable examples
- Don't try to understand everything
- Try to adapt examples to new problems
- Look up technical details in manuals/textbooks
- Learn on demand
- Keep a cool head
- Make your program small and fast - then your software long will last

About Fortran 77 and 95

Fortran 77

- Into the early/middle of the nineties Fortran 77 was the dominating language for number crunching
- It's predecessor Fortran IV was replaced by Fortran 77 in the early eighties.
- The first version of Fortran was written in 1957 and the language has evolved over time.
- Like many procedural languages Fortran has a failry 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 declaraion
- Modules containing data definitions and procedure declarations
- No implicit variable declaretions, avoiding typing errors

Fortran 77 is a subset of fortran 95

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

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

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 Fortran 77 functions where speed is critical
- Sometimes the best solution is a combination of languages, e.g. Fortran, Python and C++
- Use the language best suited for your problem

Intro to Fortran 77 programming

Our first Fortran 77 program

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

Without declaring a string variable

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

With declaring a string variable

```
C234567
      PROGRAM hw1
         CHARACTER*11 str
         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

Intro to Fortran 95 programming

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
  1. Store the first command-line argument in a floating-point variable
  2. Call the sine function
  3. Write a combination of text and numbers to the screen

The code

```
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 ! on a line
- The code is free format unlike Fortran 77

Dissection(2)

- All programs written in Fortran begins with the statement PROGRAM program_name and ends with the statement END with the optional PROGRAM program_name
- Unlike C++ and other programming language Fortran has no built in transfer of command line arguments
- A call to a C-function getarg(n,argv) transfers the n'th argument to the character-string variable argv

Dissection(2)

- Floating point variables in Fortran
  1. REAL: single precision
  2. DOUBLE PRECISION: double precision
- a2d: your own ascii string to double function, Fortran has no intrinsic functions of this kind in contrast to C/C++ so you have to write this one yourself
- 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 real number: '
READ(*,*) r
s = SIN(r)
! etc.
```

Scientific Hello World in Fortran 77

```
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

Compiling and linking Fortran programs

How to compile and link (Fortran 95)

- One step (compiling and liking):

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

- Two steps:

  unix> f90 -Wall -O3 -c hw1.f90 #Compile, result: hw1.o

  unix> f90 -o hw1 hw1.o # Link

- A linux system with Intel Fortran Compiler:

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

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
- Invoking the make utiltity:

```
unix-linux> make
```

A short example of a makefile

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

Rolling yourown 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 the make.sh script(1)

```
#!/bin/sh
if [ ! -n ‘‘$F90_COMPILER’’ ]; then
  case ‘uname -s‘ in
    Linux)
       F90_COMPILER=ifort
       F90_OPTIONS=’’-Wall -O3’’
       ;;
    *)
       F90_COMPILER=f90
       F90_OPTIONS=’’-Wall -O3’’
  esac
fi
```

The looks of the make.sh script(2)

```
files=‘/bin/ls *.f90‘
for file in files; do
  stem=‘echo $file | sed ’s/\.f90//’‘
  echo $F90_COMPILER $F90_OPTIONS -I. -o $stem $file
  $F90_COMPILER $F90_OPTIONS -I. -o $stem $file
  ls -s stem
done
```

How to compile and link (Fortran 77)

- 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 linkin 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 -L/some/libdir \\
     -L/other/libdir *.o -lmylib -lyourlib
```

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

Manipulate data files

Example: Data transformation

- Suppose we have a file with xy-data

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

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

Program structure

1. Read the names of input and output files as command-line arguments
2. Print error/usage message if less than two command-line arguments are given
3. Open the files
4. 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
5. Close the files

The fortran 95 code(1)

```
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)

```
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)

```
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
```

File handling in Fortran

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
```

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

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 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

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 inititalizing 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

Arrays and loops

Matrix-vector product

Goal: calculate a matrix-vector product

- Make s 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)`

  which is the same as in Matlab. Note that in C/C++ a multi dimensional 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++

Looping over a vector

- A simple loop

```
INTEGER    :: i
DO i = 1, n
  x(i) = f(i) + 3.14
END DO
! Definition of a function
DOUBLE PRECISION FUNCTION f(i)
   INTEGER  :: i
   ...
END FUNCTION f
```

Declaring baxic 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
```

- Note: matices are stored column wise; the row index should vary fastest
- Recall that in C/C++ matrices are stored row by row
- 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 memeory allocation in Fortran 77

- Static memory allocation (at compile time):

```
DOUBLE PRECISION, DIMENSION(100) :: x
```

- Dynamic memory allocation (at runtime):

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

Dynamic memeory allocation in Fortran 95

- Theare are two ways of declaring allocatable matrices in Fortran 95
- Using the same attrribute ALLOCATABLE like in Fortran 77
- Using a POINTER variable

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

```
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

- Computation

```
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
```

Arrays and loops

Subroutines

- A subroutine do not return any value and is the same as a void function in C/C++
- In Fortran all aguments 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 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

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

An example of a function Fortran 77 style

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

```
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 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
```

Pointers in Fortran 95

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 to another variable, it beeing a single variable, a vector or a multidimensional array
- A pointer must be associated with a target variable or another pointer

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(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)
```

Exercises

Exercise1: Modify the Fortran 95 Hello World program

- Locate the first 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

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 = start, stop, 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
```

that integrate a user-defined function f between a and b using the Trapezoidal rule with n points:
$\int_a^b f(x)\,\mathrm{d}x \approx h(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih)), h = \frac{b-a}{n-1}$

Exercise3: Integrate a function(2)

- The user defined function is specified as *external* in the argument specifications in 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 files
- Binary format (normally) saves space, and input/output is much faster since we avoid translation between ASCII characters and the binary repr.
- 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 Compaq: little endian

Exercise4: 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: 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
  1. 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)
  2. make a small Fortarn 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: 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 75

- 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
  1 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 : 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

Modules

Traditional programming

Traditional programming:

- subroutines/procedures/functions
- data structures = variables, arrays
- data are shuffled between functions

Problems with procedural approach

- Numerical codes are usually large, resulting in lots of functions with lots of arrays(large and their dimensions)
- Too many visible details
- Little correspondence between mathematical abstraction and computer code
- Redesign and reimplementation tend to be expensive

Introduction to modules

- Modules was introduced in Fortran with the Fortran 90 standard
- A module can be looked upon as some sort of a class in C++
- The module lacks some of the features of the C++ class so until Fortran 2003 is released we cannot use the OOP approach
- But we can use modules as objects and get something like the OOP style

Programming with objects

Programming with objects makes it easier to handle large and complicated code:

- Well-known in computer science/industry
- Can group large amounts of data (arrays) as a single variable
- Can make different implementation look the same for a user
- Not much explored in nummerical computing (until late 1990s)

Example: programming with matrices

Mathematical problem:

- Mattrix-matrix product: $C = MB$
- Matdix-vector product: $y = Mx$

Points to consider:

- What is a matrix
- A well defined mathematical quantity, containing a table of numbers and a set of legal operations
- How do we program with matrices?
- Do standard arrays in any computer language give good enough support for matrices?

A dense matrix in Fortran 77(1)

Fortran 77 syntax

```
c234567
      integer p, q, r
      real*8 M, B, C
      dimension(p,q) M
      dimension(q,r) B
      dimension(p,r) C
      real*8 y, x
      dimension(p) y
      dimension(q) x
C matrix-matrix product: C = M*B
      call prodm(M,p,q,B,q,r,C)
C matrix-vector product y = M*x
      call prodv(M,p,q,x,y)
```

A dense matrix in Fortran 77(2)

Drawback with this implementation

- Array sizes must be explicitly transferred
- New routines for different precisions

Working qith a dense matrix in Fortran 95

```
DOUBLE PRECISION, DIMENSION(p,q) :: M
DOUBLE PRECISION, DIMENSION(q,r) :: B
DOUBLE PRECISION, DIMENSION(p,r) :: C
DOUBLE PRECISION, DIMENSION(p)   :: x
DOUBLE PRECISION, DIMENSION(q)   :: y
M(j,k) = 3.14
C = MATMUL(M,b)
y = MATMUL(M,x)
```

Observe that

- We hide information about array sizes
- we hide storage structure
- the computer code is as compact as the mathematical notation

Array declarations in Fortran 95

- In Fortran 95 an array is in many ways like a C++ class, but with less functionality
- A Fortran 95 array contains information about the array structure and the length of each dimension
- As a part of the Fortran 95 language, functions exists to extract the shape and dimension(s) from arrays
- This means we no loger have to pass the array sizes as part of a function call

What is this module, class or object

- A module is a collection of data structures and operations on them
- It is not a new type of variable, but a *TYPE* construct is
- A module can use other modules so we can create complex units which are easy to program with

Extensions to sparse matrices

- Matrix for the discretization of $-\nabla^2 u = f$
- Only $5n$ out of n^2 entries are nonzero
- Store only the nonzero entries'!
- Many iterative solution methods for $Au = b$ can operate on the nonzeroes only

How to store sparse matrices(1)

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,4} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 & a_{2,5} \\ 0 & a_{3,2} & a_{3,3} & 0 & 0 \\ a_{4,1} & 0 & 0 & a_{4,4} & a_{4,5} \\ 0 & a_{5,2} & 0 & a_{5,5} & a_{5,5} \end{pmatrix} \quad (1)$$

- Working with nonzeroes only is important for efficiency

How to store sparse matrices(2)

- The nonzeroes can be stacked in a one-dimensional array
- Need two extra arrays to tell where a column starts and the row index of a nonzero

$$\begin{aligned} A &= (a1,1,a1,4,a2,2,a2,3,a2,5,\ldots \\ irow &= (1,3,6,8,11,14) \\ jcol &= (1,4,2,3,5,2,3,1,4,5.2.4.5) \end{aligned} \tag{2}$$

$\Rightarrow$ more complicated data structures and hence more complicated programs

Sparse matrices in Fortran 77

Code example for $y = Mx$

```
integer p, q, nnz
integer irow(p+1), jcol(nnz)
double precision M(nnz), x(q), y(p)
 ...
call prodvs(M, p, q, nnz, irow, jcol, x, y)
```

Two major drawbacks:

- Explicit transfer of storeage structure (5 args)
- Different name for two functions that perform the same task on two different matrix formats

Sparse matrix as a Fortran 95 module(1)

```
MODULE mattypes
  TYPE sparse
    DOUBLE PRECISION, POINTER :: A(:)    ! long vector with
                                         !  matrix entries
    INTEGER, POINTER          :: irow(:)! indexing array
    INTEGER, POINTER          :: jcol(:)! indexing array
    INTEGER                   :: m, n    ! A is logically
                                         ! m times n
    INTEGER                   :: nnz     ! number of nonzer
  END TYPE sparse
END MODULE mattypes
```

●

Sparse matrix as a Fortran 95 module(1)

```
MODULE matsparse
  USE mattypes
  TYPE(sparse),PRIVATE  :: hidden_sparse
CONTAINS
  SUBROUTINE prod(x, z)
      DOUBLE PRECISION, POINTER :: x(:), z(:)
      ...
  END SUBROUTINE prod
END MODULE matsparse
```

Sparse matrix as a Fortran 95 module(2)

- What has been gained?
- Users cannot see the sparse matrix data structure
- Matrix-vector product syntax remains the same
- The usage of sparse and dense matrix is the same
- Easy to switch between the two

The jungle of matrix formats

- When solving PDEs by finite element/difference methods there are numerous advantageous matrix formats:
  - dense matrix
  - banded matrix
  - tridiagonal matrix
  - general sparse matrix
  - structured sparse matrix
  - diagonal matrix
  - finite differece stencil as a matrix
- The efficiency of numerical algorithms is often strongly dependend on the matrix storage scheme
- Goal: hide the details of the storage schemes

Objects, modules and types

- Module matsparse = object
- Details of storage schemes are partially hidden
- Extensions in the module for interfaces to matrix operations
- Extensions in the module mattypes for specific storage schemes

Bad news

- Programming with modules can be a great thing, but it might be *inefficient*
- Adjusted picture:
  When indexing a matrix, one needs to know its data storage structure because of efficiency
- Module based numerics: balance between efficiency and the use of objects

A simple module

A simple module example

- We want to avoid the problems which often occurs when we need to use global variables
- We starts out showing the Fortran 77 code for global variables with an example of a problem using them
- Then we show the Fortran 95 module avoiding this particular problem

Modules and common blocks

- In Fortran 77 we had to use what is called a common block
- This common block is used to give a name to the part of the memory where we have global variables

```
INTEGER i, j
REAL     x, y
COMMON /ints/ i,j
COMMON /floats/ x, y
```

- One problem here is that the variables in a common block is position dependent

Common blocks

- An example of an error in the use of a common block

```
SUBROUTINE t1
   REAL x,y
   COMMON /floats/ y, x
   PRINT *, y
END SUBROUTINE t1
```

- Here we use the common block floats with the variables x and y
- The problem is that we have put y before x in the common declaration inside the subroutine
- What we are printing out is not the value of y, but that of x

Use a module for global variables(1)

- To avoid the previous problem we are using a module to contain the global variables
- In a module it is the name of the variable and not the position that counts
- Our global variables in a module:

```
MODULE global
   INTEGER    :: i, j
   REAL       :: x, y
END MODULE global
```

Use a module for global variables(2)

- Accessing a module and its varaibles

  ```
  SUBROUTINE t1
    USE global
    PRINT *, y
  END SUBROUTINE t1
  ```

- Now we are printing the value of variable y and not x as we did in the previous Fortran 77 example
- This is because we now are using the variable names directly and not the name of the common block

Modules and Operator Overloading

Doing arithmetic on derived datatypes

- We have a derived datatype:

```
TYPE mytype
  INTEGER                           :: i
  REAL, POINTER                     :: rvector(:)
  DOUBLE PRECISION, POINTER :: darray(:,:)
END TYPE mytype
```

- To be able to perform arithmetic operations on this derived datatype we need to create a module which does the job
- We want to overload the operators $+$, -, *, $/$, $=$

Operator overloading(1)

- What is operator overloading?
- By this we mean that we extends the functionality of the intrinsic operators $+$, -, *, $/$, $=$ to also perform the operations on other datatypes
- How do we do this in Fortran 95?

Operator overloading(2)

- This is how:

```
MODULE overload
  INTERFACE OPERATOR(+)
    TYPE(mytype) FUNCTION add(a,b)
      USE typedefs
      TYPE(mytype), INTENT(in) :: a
      TYPE(mytype), INTENT(in) :: b
    END FUNCTION add
  END INTERFACE
END MODULE overload
```

- We have now extended the traditional addition functionality to also incorporate our derived datatype mytype
- We extends the other operators in the same way except for the equal operator

Operator overloading(3)

- What the do we do to extend the equal operator?
- Not so very different than from the others

```
MODULE overload
  INTERFACE ASSIGNMENT(=)
    SUBROUTINE equals(a,b)
      USE typedefs
      TYPE(mytype), INTENT(OUT) :: a
      TYPE(mytype), INTENT(IN)  :: b
    END SUBROUTINE equals
  END INTERFACE
END MODULE overload
```

Operator overloading(4)

- Some explanations of what we have done
  - The keywords INTERFACE OPERATOR signal to the compiler that we want to extend the default operations of the operator
  - IN the same way we signal to the compiler we want to extend the default behaviour of the assignment by using the keyword it INTERFACE ASSIGNMENT
  - The difference between the assignment and operator implementation is that the first is implemented using a subroutine while the others are implemented using a function

Implementation of the multiplication operator(1)

- The multiplication operator for mytype

```
FUNCTION multiply(a,b) RESULT(c)
  USE typedefs
  TYPE(mytype), INTENT(in) :: a
  TYPE(mytype), INTENT(in) :: b
  TYPE(mytype)             :: c
  INTEGER                  :: rstat
  ALLOCATE(c%rvector(5),STAT=rstat)
  IF(rstat /= 0) THEN
    PRINT *, 'Error in allocating x.rvector ', rstat
  END IF
  ...
```

- It is important to remember that the implementation of the operators is in a separate file

Implementation of the multiplication operator(2)

- The multiplication operator for mytype

```
    ...
    ALLOCATE(c%darray(5,5),STAT=rstat)
    IF(rstat /= 0) THEN
      PRINT *, 'Error in allocating x.darray ', rstat
    END IF
    c%i = a%i * b%i
    c%rvector = a%rvector * b%rvector
    c%darray = a%darray * b%darray
END FUNCTION multiply
```

- It is important to remember to allocate the memory space for the result of the multiplication. Unless you do this the program will crash

How we implement the assignment operator

- The assignment operator for mytype

```
SUBROUTINE equals(a,b)
  USE typedefs
  TYPE(mytype), INTENT(OUT) :: a
  TYPE(mytype), INTENT(IN)  :: b
  a%i = b%i
  a%rvector = b%rvector
  a%darray = b%darray
END SUBROUTINE equals
```

- It is important to remember that the implementation of the assignment is in a separate file

What have we really done in these two examples(1)

- The multiply function takes two input arguments and returns a variable of mytype
- To avoid mistakes of changing the value of any of the two arguments both have the attibute *INTENT(IN)*
- In C++ we would typically use the keyword *const* as an attribute to the argument
- The default for attribute for arguments to functions and subroutines in Fortran is *INTENT(INOUT)* which allowes us to modify the argument
- Fortran has the nice feature that we can multiply an array with another provided they have the same shape and size. We therefore does not need to go through one or more loops to perform the multiplication

What have we really done in these two examples(2)

- The assigment is implemented using a subroutine
- Like in the multiplicaion we use the *INTENT(IN)* attribute to the second argument
- To the first argument we use the *INTENT(OUT)* attribute to signal that this argument is for the return of a value only

-