

Getting started with Fortran 90/95

Jonas Lindemann och Ola Dahlblom

May 12, 2009



Contents

1	Introduction	1
2	Fortran 95/2003	3
2.1	Program structure	3
2.2	Variables	5
2.2.1	Naming of variables	5
2.2.2	Variable declarations	5
2.2.3	Arrays and matrices	7
2.2.4	Dynamic variables	8
2.2.5	Assignment of variables	9
2.3	Operatorer	10
2.4	Conditional statements	11
2.5	Repetitive statements	14
2.6	Built-in functions	17
2.7	Subroutines and functions	20
2.8	Input and output	24
2.8.1	Reading and Writing from files	26
2.9	String manipulation	27
2.10	Modules	28
3	Photran	33
3.1	Starting Photran	33
3.2	Creating a Photran Makefile project	34
3.2.1	Adding a new source file	36
3.3	Building the project	36
3.4	Running the project	37
	Litteraturförteckning	39
A	Exercises	41
A.1	Fortran	41

Chapter 1

Introduction

This book is an introduction in programming with Fortran 95/2003 i in science and technology. The book also covers methods for integrating Fortran code with other programming languages both dynamic (Python) and compiled languages (C++). An introduction in using modern development environments such as Eclipse/Photran, for debugging and development is also given.

Chapter 2

Fortran 95/2003

Fortran was the first high-level language and was developed in the fifties. The language has since developed through a number of standards Fortran IV (1966), Fortran 77, Fortran 90, Fortran 95 and the latest Fortran 2003. The advantages with standardised languages is that the code can be run on different computer architectures without modification. In every new standard the language has been extended with more modern language elements. To be compatible with previous standards older language elements are not removed. However, language elements that are considered bad or outdated can be removed after 2 standard revisions. As an example Fortran 90 is fully backwards compatible with Fortran 77, but in Fortran 95 some older language constructs were removed.

The following sections give a short introduction to the Fortran 90 language and some of the extensions in Fortran 95. The description is centered on the most important language features. A more thorough description of the language can be found in the book Fortran 95/2003 Explained [1]

2.1 Program structure

Every Fortran-program must have a main program routine. From the main routine, other subroutines that make up the program are called. The syntax for a main program is:

```
[program program-name]  
[specification statements]  
[executable statements]  
[contains]  
[subroutines]  
end [program [program-name]]
```

From the syntax it can be seen that the only identifier that must be included in main program definition is **end**.

The syntax for a subroutine and functions are defined in the same way, but the **program** identifier is replaced with **subroutine** or **function**. A proper way of organizing subroutines is to place these in separate files or place them in modules (covered in upcoming sections). Subroutines can also be placed in the main program **contains**-section, which is the preferred

method if all subroutines are placed in the same source file. The code below shows a simple example of a main program with a subroutine in Fortran.

```
program sample1

    integer, parameter :: ap=selected_real_kind(15,300)
    real(kind=ap) :: x, y
    real(kind=ap) :: k(20,20)

    x = 6.0_ap
    y = 0.25_ap

    write(*,*) x
    write(*,*) y
    write(*,*) ap

    call myproc(k)

    write(*,*) k(1,1)

contains

subroutine myproc(k)

    integer, parameter :: ap=selected_real_kind(15,300)
    real(kind=ap) :: k(20,20)

    k=0.0_ap
    k(1,1) = 42.0_ap

    return

end subroutine myproc

end program sample1

!

---


```

The program source code can contain upper and lower case letters, numbers and special characters. However, it should be noted that Fortran does not differentiate between upper and lower case letters. The program source is written starting from the first position with one statement on each line. If a row is terminated with the character `&`, this indicates that the statement is continued on the the next line. All text placed after the character `!` is a comment and wont affect the function of the program. Even if the comments don't have any function in the program they are important for source code readability. This is especially important for future modification of the program. In addition to the source code form described above there is also the possibility of writing code in fixed form, as in Fortran 77 and earlier versions. In previous version of the Fortran standard this was the only source code form available.

2.2 Variables

2.2.1 Naming of variables

Variables in Fortran 95 consists of 1 to 31 alphanumeric characters (letters except å ä and ö, underscore and numbers). The first character of a variable name must be a letter. Allowable variable names can be:

```
a
a_thing
x1
mass
q123
time_of_flight
```

Variable names can consist of both upper case and lower case letters. It should be noted that `a` and `A` references the same variable. Invalid variables names can be:

```
1a      ! First character not a letter
a thing ! Contains a space character
\ $ sign ! Contains a non-alphanumeric character
```

2.2.2 Variable declarations

There are 5 built-in data types in Fortran:

- integer, Integers
- real, Floating point numbers
- complex, Complex numbers
- logical, Boolean values
- character, Strings and characters

The syntax for a variable declaration is:

```
type [[,attribute]... ::] entity-list
```

type defines the variable type and can be `integer`, `real`, `complex`, `logical`, `character`, or `type(type-name)`. *attribute* defines additional special attributes or how the variable is to be used. The following examples shows some typical Fortran variable declarations.

```
integer :: a      ! Scalar integer variable
real    :: b      ! Scalar floating point variable
logical :: flag  ! boolean variable

real    :: D(10)  ! Floating point array consisting of 10 elements
real    :: K(20,20) ! Floating point array of 20x20 elements

integer , dimension(10) :: C      ! Integer array of 10 elements
```

```

character :: ch           ! Character
character , dimension(60) :: chv ! Array of characters
character(len=80) :: line   ! Character string
character(len=80) :: lines(60) ! Array of strings

```

Constants are declared by specifying an additional parameter, `parameter`. A declared constant can be used in following variable declarations. An example of use is shown in the following example.

```

integer , parameter :: A = 5 ! Integer constant
real :: C(A)                 ! Floating point array where
                              ! the number of elements is
                              ! specified by A

```

The precision and size of the variable type can be specified by adding a parenthesis directly after the type declaration. The variables `A` and `B` in the following example are declared as floating point scalars with different precisions. The number in the parenthesis denotes for many architectures, how many bytes a floating point variable is represented with.

```

real(8) :: A
real(4) :: B
integer(4) :: I

```

To be able to choose the correct precision for a floating point variable, Fortran has a built in function `selected_real_kind` that returns the value to be used in the declaration with a given precision. This is illustrated in the following example.

```

integer , parameter :: ap = selected_real_kind(15,300)
real(kind=ap) :: X,Y

```

In this example the floating point variable should have at least 15 significant decimals and could represent numbers from 10^{-300} to 10^{300} . For several common architectures `selected_real_kind` will return the value 8. The advantage of using the above approach is that the precision of the floating point values can be specified in a architectural independent way. The precision constant can also be used when specifying numbers in variable assignments as the following example illustrate.

```

X = 6.0_ap

```

The importance of specifying the precision for assigning scalar values to variables is illustrated in the following example.

```

program constants
  implicit none

```

```

integer , parameter :: ap = selected_real_kind(15,300)

real(ap) :: pi1 , pi2
pi1 = 3.141592653589793
pi2 = 3.141592653589793_ap

write(*,*) 'pi1_=_', pi1
write(*,*) 'pi2_=_', pi2

stop

end program constants

```

The program gives the following results:

```

pi1 = 3.14159274101257
pi2 = 3.14159265358979

```

The scalar number assigned to the variable `pi1` is chosen by the compiler to be represented by the least number of bytes floating point precision, in this case `real(4)`, which is shown in the output from the above program.

Variable declarations in Fortran always precedes the executable statements in the main program or in a subroutine. Declarations can also be placed directly after the `module` identifier in modules. Variable does not have to be declared in Fortran. The default is that variables starting I, J,..., N are defined as `integer` and variables starting with A, B,..., H or O, P,..., Z are defined as `real`. This kind of implicit variable declaration is not recommended as it can lead to programming errors when variables are misspelled. To avoid implicit variable declarations the following declaration can be placed first in a program or module:

```

implicit none

```

This statement forces the compiler to make sure that all variables are declared. If a variable is not declared the compilation is stopped with an error message. This is default for many other strongly typed languages such as, C, C++ and Java.

2.2.3 Arrays and matrices

In scientific and technical applications matrices and arrays are important concepts. As Fortran is a language mainly for technical computing, arrays and matrices play a vital role in the language.

Declaring arrays and matrices can be done in two ways. In the first method the dimensions are specified using the special attribute, `dimension`, after the data type declaration. The second method, the dimensions are specified by adding the dimensions directly after the variable name. The following code illustrate these methods of declaring arrays.

```

integer , parameter :: ap = selected_real_kind(15,300)
real(ap), dimension(20,20) :: K ! Matrix 20x20 elements

```

```
real(ap) :: fe(6) ! Array with 6 elements
```

The default starting index in arrays is 1. It is however possible to define custom indices in the declaration, as the following example shows.

```
real(ap) :: idx(-3:3)
```

This declares an array, `idx` with the indices [-3, -2, -1, 0, 1, 2, 3], which contains 7 elements.

2.2.4 Dynamic variables

In Fortran 77 and earlier versions of the standard it was not possible to dynamically allocate memory during program execution. This capability is now available in Fortran 90 and later versions. To declare an array as dynamically allocatable, the attribute `allocatable` must be added to the array declaration. The dimensions are also replaced with a colon, `:`, indicating the number of dimensions in the declared variable. A typical allocatable array declaration is shown in the following example.

```
real , dimension (:,:) , allocatable :: K
```

In this example the two-dimensional array, `K`, is defined as allocatable. To indicate that the array is two-dimensional is done by specifying `dimension(:,:)` in the variable attribute. To declare a one-dimensional array the code becomes:

```
real , dimension (:) , allocatable :: f
```

Variables with the `allocatable` attribute can't be used until memory is allocated. Memory allocation is done using the `allocate` method. To allocate the variables, `K`, `f`, in the previous examples the following code is used.

```
allocate (K(20,20))
allocate (f(20))
```

When the allocated memory is no longer needed it can be deallocated using the command, `deallocate`, as the following code illustrates.

```
deallocate (K)
deallocate (f)
```

An important issue when using dynamically allocatable variable is to make sure the application does not "leak". "Leaking" is term used by applications that allocate memory during the execution and never deallocate used memory. If unchecked the application will use more and more resources and will eventually make the operating system start swapping and perhaps become also become unstable. A rule of thumb is that an `allocate` statement should always have corresponding `deallocate`. An example of using dynamically allocated arrays is shown in section XXX.

2.2.5 Assignment of variables

The syntax for scalar variable assignment is,

```
variable = expr
```

where *variable* denotes the variable to be assigned and *expr* the expression to be assigned. The following example assign the a variable the value 5.0 with the precision defined in the constant *ap*.

```
a = 5.0_ap
```

Assignment of boolean variables are done in the same way using the keywords, *.false.* and *.true.* indicating a true or false value. A boolean expression can also be used in the assignment. In the following example the variable, *flag*, is assigned the value *.false.*.

```
flag = .false.
```

Assignment of strings are illustrated in the following example.

```
character(40) :: first_name
character(40) :: last_name
character(20) :: company_name1
character(20) :: company_name2

...

first_name = 'Jan'
last_name = "Johansson"
company_name1 = "McDonald's"
company_name2 = 'McDonald''s'
```

The first variable, *first_name*, is assigned the text "Jan", remaining characters in the string will be padded with spaces. A string is assigned using citation marks, " or apostrophes, '. This can be of help when apostrophes or citation marks is used in strings as shown in the assignment of the variables, *company_name1* och *company_name2*.

Arrays are assigned values either by explicit indices or the entire array in a single statement. The following code assigned the variable, *K*, the value 5.0 at position row 5 and column 6.

```
K(5,6) = 5.0
```

If the assignment had been written as

```
K = 5.0
```

the entire array, *K*, would have been assigned the value 5.0. This is an efficient way of assigning entire arrays initial values.

Explicit values can be assigned to arrays in a single statement using the following assignment.

```
real(ap) :: v(5) ! Array with 5 elements
v = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)
```

This is equivalent to an assignment using the following statements.

```
v(1) = 1.0
v(2) = 2.0
v(3) = 3.0
v(4) = 4.0
v(5) = 5.0
```

The number of elements in the list must be the same as the number of elements in the array variable.

Assignments to specific parts of arrays can be achieved using index-notation. The following example illustrates this concept.

```
program index_notation

    implicit none
    real :: A(4,4)
    real :: B(4)
    real :: C(4)

    B = A(2,:) ! Assigns B the values of row 2 in A
    C = A(:,1) ! Assigns C the values of column 1 in A

    stop

end program index_notation
```

Using index-notation rows or columns can be assigned in single statements as shown in the following code:

```
! Assign row 5 in matrix K the values 1, 2, 3, 4, 5
K(5,:) = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)

! Assign the array v the values 5, 4, 3, 2, 1
v = (/ 5.0, 4.0, 3.0, 2.0, 1.0 /)
```

2.3 Operatorer

The following arithmetic operators are defined in Fortran:

****** power to
***** multiplication
/ division
+ addition
- subtraction

Parenthesis are used to specify the order of different operators. If no parenthesis are given in an expression operators are evaluated in the following order:

1. Operations with ******
2. Operations with ***** or **/**
3. Operations with **+** or **-**

The following code illustrates operator precedence.

```

c = a+b/2 ! is equivalent to a + (b/2)
c = (a+b)/2 ! in this case (a + b) is evaluated and then / 2

```

Relational operators:

< or **.lt.** less than (less than)
<= or **.le.** less than or equal to (less than or equal)
> or **.gt.** greater than (greater than)
>= or **.ge.** greater than or equal to (greater than or equal)
== or **.eq.** equal to (equal)
/= or **.ne.** not equal to (not equal)

Logical operators:

.and. and
.or. or
.not. not

2.4 Conditional statements

The simplest form of if-statements in Fortran have the following syntax

```

if (scalar-logical-expr) then
  block
end if

```

where *scalar-logical-expr* is a boolean expression, that has to be evaluated as true, (**.true.**), for the statements in, *block*, to be executed. An extended version of the if-statement adds a **else**-block with the following syntax

```

if (scalar-logical-expr) then
  block1
else
  block2
end if

```

In this form the *block1* will be executed if *scalar-logical-expr* is evaluated as true, otherwise *block2* will be executed. A third form of if-statement contains one or more **else if**-statements with the following syntax:

```

if (scalar-logical-expr1) then
  block1
else if (scalar-logical-expr2) then
  block2
else
  block3
end if

```

In this form the *scalar-logical-expr1* is evaluated first. If this expression is true *block1* is executed, otherwise if *scalar-logical-expr2* evaluates as true *block2* is executed. If no other expressions are evaluated to true, *block3* is executed. An if-statement can contain several **else if**-blocks. The use of if-statements is illustrated in the following example:

```

program logic

  implicit none

  integer :: x
  logical :: flag

  ! Read an integer from standard input

  write(*,*) 'Enter an integer value.'
  read(*,*) x

  ! Correct value to the interval 0-1000

  flag = .FALSE.

  if (x>1000) then
    x = 1000
    flag = .TRUE.
  end if

  if (x<0) then
    x = 0
    flag = .TRUE.
  end if

  ! If flag is .TRUE. the input value

```



```

    ! has been corrected.

    if (flag) then
        write(*, '(a, l4)') 'Corrected_value_=_', x
    else
        write(*, '(a, l4)') 'Value_=_', x
    end if

    stop

end program logic

```

Another conditional construct is the case-statement.

```

select case (expr)
  case selector
    block
end select

```

In this statement the expression, *expr* is evaluated and the *case*-block with the corresponding *selector* is executed. To handle the case when no *case*-block corresponds to the *expr*, a *case*-block with the *default* keyword can be added. The syntax then becomes:

```

select case(expr)
  case selector
    block
  case default
    block
end select

```

Example of case-statement use is shown in the following example:

```

select case (display_mode)
  case (displacements)
    ...
  case (geometry)
    ...
end select

```

To handle the case when *display_mode* does not correspond to any of the alternatives the above code is modified to the following code.

```

select case (display_mode) case (displacements)
  ...
  case (geometry)
    ...
  case default
    ...
end select

```

The following program example illustrate how case-statements can be used.

```

program case_sample

    integer :: value

    write(*,*) 'Enter_a_value '
    read(*,*) value

    select case (value)
        case (:0)
            write(*,*) 'Greater_than_one.'
        case (1)
            write(*,*) 'Number_one!'
        case (2:9)
            write(*,*) 'Between_2_and_9.'
        case (10)
            write(*,*) 'Number_10!'
        case (11:41)
            write(*,*) 'Less_than_42_but_greater_than_10.'
        case (42)
            write(*,*) 'Meaning_of_life_or_perhaps_6*7.'
        case (43:)
            write(*,*) 'Greater_than_42.'
        case default
            write(*,*) 'This_should_never_happen!'
    end select

    stop

end program case_sample

```

2.5 Repetitive statements

The most common repetitive statement in Fortran is the **do**-statement. The syntax is:

```

do variable = expr1, expr2 [,expr3]
    block
end do

```

variable is the control-variable of the loop. *expr1* is the starting value, *expr2* is the end value and *expr3* is the step interval. If the step interval is not given it is assumed to be 1. There are two ways of controlling the execution flow in a **do**-statement. The **exit** command terminates the loop and program execution is continued after the **do**-statement. The **cycle** command terminates the execution of the current block and continues execution with the next value of the control variable. The example below illustrates the use of a **do**-statement.

```

program loop_sample

    implicit none

    integer :: i

    do i=1,20
        if (i>10) then
            write(*,*) 'Terminates_do-statement.'
            exit
        else if (i<5) then
            write(*,*) 'Cycling_to_next_value.'
            cycle
        end if
        write(*,*) i
    end do

    stop

end program loop_sample

```

The above program gives the following output:

```

Cycling to next value.
Cycling to next value.
Cycling to next value.
Cycling to next value.
5
6
7
8
9
10
Terminates do-statement.

```

Another repetitive statement available is the **do while**-statement. With this statement, the code block can execute until a certain condition is fulfilled. The syntax is:

```

do while (scalar-logical-expr)
    block
end do

```

The following code shows a simple **do while**-statement printing the function $f(x) = \sin(x)$.

```

x = 0.0
do while x<1.05
    f = sin(x)
    x = x + 0.1
    write(*,*) x, f
end do

```

Fortran 95 has added a number of new loop-statements. The **forall**-statement has been added to optimise nested loops for execution on multiprocessor machines. The syntax is:

```
forall (index = lower:upper [,index = lower:upper])
  [body]
end forall
```

The following example shows how a **do**-statement can be replaced with a **forall**-statement.

```
do i=1,n
  do j=1,m
    A(i , j)=i+j
  end do
end do
```

! Is equivalent with

```
forall (i=1:n, j=1:m)
  A(i , j)=i+j
end forall
```

Another statement optimised for multiprocessor architectures is the **where**-statement. With this statement conditional operations on an array can be achieved efficiently. The syntax comes in two versions.

```
where (logical-array-expr)
  array-assignments
end where
```

and

```
where (logical-array-expr)
  array-assignments
else where
  array-assignments
end where
```

The usage of the **where**-statement is best illustrated with an example.

```
where (A>1)
  B = 0
else where
  B = A
end where
```

In this example two arrays with the same size are used in the **where**-statement. In this case the values in the B array are assigned 0 when an element in the A array is larger than 1 otherwise the element in B is assigned the same value as in the A array.

2.6 Built-in functions

Fortran has a number of built-in functions covering a number of different areas. The following tables list a selection of these. For a more thorough description of the built-in function please see, Metcalf and Reid [1].

Function	Description
acos(x)	Returns $\arccos(x)$
asin(x)	Returns $\arcsin(x)$
atan(x)	Returns $\arctan(x)$
atan2(y,x)	Returns $\arctan(\frac{y}{x})$ from $-\pi$ till π
cos(x)	Returns $\cos(x)$
cosh(x)	Returns $\cosh(x)$
exp(x)	Returns e^x
log(x)	Returns $\ln(x)$
log10(x)	Returns $\lg(x)$
sin(x)	Returns $\sin(x)$
sinh(x)	Returns $\sinh(x)$
sqrt(x)	Returns \sqrt{x}
tan(x)	Returns $\tan(x)$
tanh(x)	Returns $\tanh(x)$

Table 2.1: Mathematical functions

Function	Description
abs(a)	Returns absolute value of a
aint(a)	Truncates a floating point value
int(a)	Converts a floating point value to an integer
nint(a)	Rounds a floating point value to the nearest integer
real(a)	Converts an integer to a floating point value
max(a1,a2[,a3 ,...])	Returns the maximum value of two or more values
min(a1,a2[,a3 ,...])	Returns the minimum value of two or more values

Table 2.2: Miscellaneous conversion functions

Function	Description
dot_product(u, v)	Returns the scalar product of $u \cdot v$
matmul(A, B)	Matrix multiplication. The result must have the same for as AB
transpose(C)	Returns the transpose \mathbf{C}^T . Elementet C_{ij}^T motsvarar C_{ji}

Table 2.3: Vector and matrix functions

Most built-in functions and operators in Fortran support arrays. The following example shows how functions and operators support operations on arrays.

```
real , dimension(20,20) :: A, B, C
```

```
C = A/B ! Division  $C_{ij} = A_{ij}/B_{ij}$ 
```

```
C = sqrt(A) ! Square root  $C_{ij} = \sqrt{A_{ij}}$ 
```

The following example shows how a stiffness matrix for a bar element easily can be created using these functions and operators. The Matrix \mathbf{K}_e is defined as follows

$$\mathbf{K}_e = (\mathbf{G}^T \mathbf{K}_{el}) \mathbf{G} \quad (2.1)$$

The \mathbf{G}^T is returned by using the Fortran function `transpose` and the matrix multiplications are performed with `matmul`. The matrices \mathbf{K}_{el} and \mathbf{G} are defined as

$$\mathbf{K}_{el} = \frac{EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad (2.2)$$

and

Function	Description
<code>all(mask)</code>	Returns true if all elements in the logical array <i>mask</i> are true. For example <code>all(A > 0)</code> returns true if all elements in A are greater than 0.
<code>any(mask)</code>	Returns true if any of the elements in <i>mask</i> are true.
<code>count(mask)</code>	Returns the number of elements in <i>mask</i> that are true.
<code>maxval(array)</code>	Returns the maximum value of the elements in the array <i>array</i> .
<code>minval(array)</code>	Returns the minimum value of the elements in the array <i>array</i> .
<code>product(array)</code>	Returns the product of the elements in the array <i>array</i> .
<code>sum(array)</code>	Returns the sum of elements in the array <i>array</i> .

Table 2.4: Array functions

$$\mathbf{G} = \begin{bmatrix} n_x & n_y & n_z & 0 & 0 & 0 \\ 0 & 0 & 0 & n_x & n_y & n_z \end{bmatrix} \quad (2.3)$$

Length and directional cosines are defined as

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \quad (2.4)$$

$$n_x = \frac{x_2 - x_1}{L}, \quad n_y = \frac{y_2 - y_1}{L}, \quad n_z = \frac{z_2 - z_1}{L} \quad (2.5)$$

In the example the input parameters are assigned the following values:

$$x_1 = 0, y_1 = 0, z_1 = 0 \quad (2.6)$$

$$x_2 = 1, y_2 = 1, z_2 = 1 \quad (2.7)$$

$$E = 1, A = 1 \quad (2.8)$$

```

program function_sample

  implicit none

  integer, parameter :: ap = selected_real_kind(15,300)

  integer :: i, j

  real(ap) :: x1, x2, y1, y2, z1, z2
  real(ap) :: nx, ny, nz
  real(ap) :: L, E, A
  real(ap) :: Kel(2,2)
  real(ap) :: Ke(6,6)
  real(ap) :: G(2,6)

  ! Initiate scalar values

  E = 1.0_ap
  A = 1.0_ap
  x1 = 0.0_ap
  x2 = 1.0_ap
  y1 = 0.0_ap
  y2 = 1.0_ap
  z1 = 0.0_ap
  z2 = 1.0_ap

  ! Calculate directional cosines

  L = sqrt( (x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2 )
  nx = (x2-x1)/L
  ny = (y2-y1)/L
  nz = (z2-z1)/L

```

```

! Calucate local stiffness matrix

Kel(1,:) = (/ 1.0_ap , -1.0_ap /)
Kel(2,:) = (/ -1.0_ap , 1.0_ap /)

Kel = Kel * (E*A/L)

G(1,:) = (/ nx, ny, nz, 0.0_ap, 0.0_ap, 0.0_ap /)
G(2,:) = (/ 0.0_ap, 0.0_ap, 0.0_ap, nx, ny, nz /)

! Calculate transformed stiffness matrix

Ke = matmul(matmul(transpose(G),Kel),G)

! Print matrix

do i=1,6
  write(*,'(6G10.3)') (Ke(i,j), j=1,6)
end do

stop

end program function_sample

```

The program produces the following output

```

0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
-0.1925   -0.1925   -0.1925    0.1925    0.1925    0.1925
-0.1925   -0.1925   -0.1925    0.1925    0.1925    0.1925
-0.1925   -0.1925   -0.1925    0.1925    0.1925    0.1925

```

For a more thorough description of matrix handling in Fortran 90/95, see Metcalf and Reid [1]

2.7 Subroutines and functions

A subroutine in Fortran 90/95 has the following syntax

```

subroutine subroutine-name[[dummy-argument-list]]
  [argument-declaration]
  ...
return
end subroutine [subroutine-name]

```

All variables in Fortran program are passed to subroutines as references to the actual variables. Modifying a parameter in a subroutine will modify the values of variables in the

calling subroutine or program. To be able to use the variables in the argument list they must be declared in the subroutine. This is done right after the subroutine declaration. When a subroutine is finished control is returned to the calling routine or program using the `return`-command. Several return statements can exist in subroutine to return control to the calling routine or program. This is illustrated in the following example.

```

subroutine myproc(a,B,C)
  implicit none
  integer :: a
  real , dimension(a,*) :: B
  real , dimension(a) :: C
  .
  .
  .
  return
end subroutine

```

A subroutine is called using the `call` statement. The above subroutine is called with the following code.

```

call myproc(a,B,C)

```

It should be noted that the names used for variables are local to each respective subroutine. Names of variables passed as arguments does not need to have the same name in the calling and called subroutines. It is the order of the arguments that determines how the variables are referenced from the calling subroutine.

In the previous example illustrates how to make the subroutines independent of problem size. The dimensions of the arrays are passed using the `a` parameter instead of using constant values. The last index of an array does not have to be specified, indicated with a `*`, as it is not needed to determine the address to array element.

Functions are subroutines with a return value, and can be used in different kinds of expressions. The syntax is

```

type function function-name([dummy-argument-list])
  [argument-declaration]
  ...
  function-name = return-value
  ...
  return
end function function-name

```

The following code shows a simple function definition returning the value of $\sin(x)$

```

real function f(x)
  real :: x
  f=sin(x)
  return
end function f

```

The return value defined by assigning the name of the function a value. As seen in the previous example. The function is called by giving the name of the function and the associated function arguments.

```
a = f(y)
```

The following example illustrates how to use subroutines to assign an element matrix for a three-dimensional bar element. The example also shows how dynamic memory allocation can be used to allocate matrices. See also the example in section XX

```
program subroutine_sample

    integer, parameter :: ap = &
        selected_real_kind(15,300)

    real(ap) :: ex(2), ey(2), ez(2), ep(2)
    real(ap), allocatable :: Ke(:, :)

    ep(1) = 1.0_ap
    ep(2) = 1.0_ap
    ex(1) = 0.0_ap
    ex(2) = 1.0_ap
    ey(1) = 0.0_ap
    ey(2) = 1.0_ap
    ez(1) = 0.0_ap
    ez(2) = 1.0_ap

    allocate(Ke(6,6))

    call bar3e(ex, ey, ez, ep, Ke)
    call writeMatrix(Ke)

    deallocate(Ke)

    stop

end program subroutine_sample

subroutine bar3e(ex, ey, ez, ep, Ke)

    implicit none

    integer, parameter :: ap = &
        selected_real_kind(15,300)

    real(ap) :: ex(2), ey(2), ez(2), ep(2)
    real(ap) :: Ke(6,6)

    real(ap) :: nxx, nyx, nzx
    real(ap) :: L, E, A
    real(ap) :: Kel(2,2)
```

```

real(ap) :: G(2,6)

! Calculate directional cosines

L = sqrt( (ex(2)-ex(1))**2 + (ey(2)-ey(1))**2 + &
          (ez(2)-ez(1))**2 )
nxx = (ex(2)-ex(1))/L
nyx = (ey(2)-ey(1))/L
nzx = (ez(2)-ez(1))/L

! Calculate local stiffness matrix

Kel(1,:) = (/ 1.0_ap , -1.0_ap /)
Kel(2,:) = (/ -1.0_ap , 1.0_ap /)

Kel = Kel * (ep(1)*ep(2)/L)

G(1,:) = (/ nxx , nyx , nzx , 0.0_ap , 0.0_ap , 0.0_ap /)
G(2,:) = (/ 0.0_ap , 0.0_ap , 0.0_ap , nxx , nyx , nzx /)

! Calculate transformed stiffness matrix

Ke = matmul(matmul(transpose(G),Kel),G)

return

end subroutine bar3e

subroutine writeMatrix(A)

integer , parameter :: ap = &
    selected_real_kind(15,300)

real(ap) :: A(6,6)

! Print matrix

do i=1,6
    write(*, '(6G10.4)') (A(i,j) , j=1,6)
end do

return

end subroutine writeMatrix

```

The program gives the following output.

```

0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
0.1925    0.1925    0.1925   -0.1925   -0.1925   -0.1925
-0.1925   -0.1925   -0.1925    0.1925    0.1925    0.1925

```

```

-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925

```

2.8 Input and output

Input and output to and from different devices, such as screen, keyboard and files are accomplished using the commands `read` and `write`. The syntax for these commands are:

```

read(u, fmt) [list]
write(u, fmt) [list]

```

u is the device that is used for reading or writing. If a star (*) is used as a device, standard output and standard input are used (screen, keyboard or pipes).

fmt is a string describing how variables should be read or written. This is often important when writing results to text files, to make it more easily readable. If a star (*) is used a so called free format is used, no special formatting is used. The format string consists of one or more format specifiers, which have the general form:

```
[repeat-count] format-descriptor w[.m]
```

where *repeat-count* is the number of variables that this format applies to. *format-descriptor* defines the type of format specifier. *w* defined the width of the output field and *m* is the number of significant numbers or decimals in the output. The following example outputs some numbers using different format specifiers and table 2.5 show the most commonly used format specifiers.

```

program formatting

  implicit none

  integer, parameter :: ap = &
    selected_real_kind(15,300)

  write (*, '(A15)') '123456789012345'
  write (*, '(G15.4)') 5.675789_ap
  write (*, '(G15.4)') 0.0675789_ap
  write (*, '(E15.4)') 0.675779_ap
  write (*, '(F15.4)') 0.675779_ap
  write (*, *)          0.675779_ap
  write (*, '(I15)') 156
  write (*, *)          156

  stop

end program formatting

```

The program produces the following output:

Kod	Beskrivning
E	Scientific notation. Values are converted to the format "-d.dddE+ddd".
F	Decimal notation. Values are converted to the format "-d ddd.ddd...".
G	Generic notation. Values are converted to the format -ddd.ddd or -d.dddE+ddd
I	Integers.
A	Strings
TR n	Move n positions right
T n	Continue at position n

Table 2.5: Formatting codes in read/write

```

123456789012345
  5.676
  0.6758E-01
  0.6758E+00
    0.6758
  0.675779000000000
    156
    156

```

During output a invisible cursor is moved from left to right. The format specifiers TR n and T n are used to move this cursor. TR n moves the cursor n positions to the right from the previous position. T n places the cursor at position n . Figure ?? shows how this can be used in a write-statement.

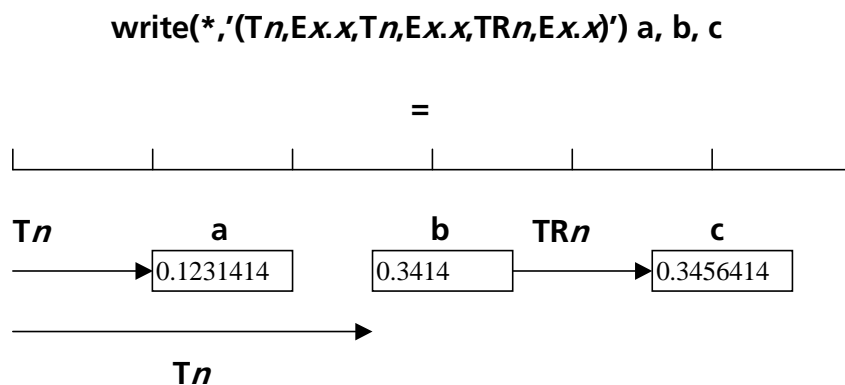


Figure 2.1: Positioning of output in Fortran 90/95

The output routines in Fortran was originally intended to be used on row printers where the first character was a control character. The effect of this is that the default behavior of these routines is that output always starts at the second position. On modern computers this is not an issue, and the first character can be used for printing. To print from the first character, the format specifier T1 can be used to position the cursor at the first position. The following code writes "Hej hopp!" starting from the first position.

```
write(*,'(T1,A)') 'Hej_hopp!'
```

A more thorough description of the available format specifiers in Fortran is given in Metcalf and Reid [1].

2.8.1 Reading and Writing from files

The input and output routines can also be used to write data to and from files. This is accomplished by associating a file in the file system with a file unit number, and using this number in the `read` and `write` statements to direct the input and output to the correct files. A file is associated, opened, with a unit number using an `open`-statement. When operations on the file is finished it is closed using the `close`-statement.

The file unit number is an integer usually between 1 and 99. On many systems the file unit number 5 is the keyboard and unit 6 the screen display. It is therefore recommended to avoid using these numbers in file operations.

In the `open`-statement the properties of the opened files are given, such as if the file already exists, how the file is accessed (reading or writing) and the filename used in the filesystem.

An example of reading and writing file is given in the following example.

```

program sample2

  implicit none
  real(8), allocatable :: infield(:, :)
  real(8), allocatable :: rowsum(:)
  integer :: rows, i, j

  ! File unit numbers

  integer, parameter :: infile = 15
  integer, parameter :: outfile = 16

  ! Allocate matrices

  rows=5
  allocate(infield(3, rows))
  allocate(rowsum(rows))

  ! Open the file 'indata.dat' for reading

  open(unit=infile, file='indata.dat', &
       access='sequential', &
       action='read')

  ! Open the file 'utdata.dat' for writing

  open(unit=outfile, file='utdata.dat', &
       access='sequential', &
       action='write')

  ! Read input from file

```

```

do i=1,rows
  read(infile,*) (infield(j,i),j=1,3)
  rowsum(i)=&
    infield(1,i)+infield(2,i)+infield(3,i)
  write(outfile,*) rowsum(i)
end do

! Close files

close(infile)
close(outfile)

! Free used memory

deallocate(infield)
deallocate(rowsum)

stop

end program sample2

```

In this example, 2 files are opened, `indata.dat` and `utdata.dat` with `open`-statements. Using the `read`-statement five rows with 3 numbers on each row are read from the file `indata.dat`. The sum of each row is calculated and is written using `write`-statements to the file `utdata.dat`. Finally the files are closed using the `close`-statements.

2.9 String manipulation

There are several ways of manipulating strings in Fortran. Strings can be concatenated with the operator, `//`, as shown in the following example:

```

c1 = 'Hej_'
c2 = 'hopp!'
c = c1 // c2 ! = 'Hej_hopp!'

```

Fortran does not have dynamic strings, so the size of the resulting string must be large enough for the concatenated string.

Substrings can be extracted using a syntax similar to the syntax used when indexing arrays.

```

c3 = c(5:8) ! Contains the string 'hopp'

```

A common task in many codes is the conversion of numbers to and from strings. Fortran does not have any explicit functions these type of conversions, instead the the `read` and `write` statements can be used together with strings to accomplish the same thing. By replacing the file unit number with a character string variable, the string can be read from and written to using `read` and `write` statements.

To convert a floating point value to a string the following code can be used.

```
character(255) :: mystring
real(8) :: myvalue
value = 42.0
write(mystring, '(G15.4)') value
! mystring now contains '      5.676'
```

To convert a value contained in string to a floating point value the read-statement is used.

```
character(255) :: mystring
real(8) :: myvalue
mystring = '42.0'
read(mystring,*) myvalue
! myvalue now contains 42.0
```

A more complete example is shown in the following listing:

```
program strings2

  implicit none

  integer :: i
  character(20) :: c

  c = '5'
  read(c, '(15)') i
  write(*,*) i

  i = 42
  write(c, '(15)') i
  write(*,*) c

  stop

end program strings2
```

The program produces the following output.

```
5
42
```

2.10 Modules

When programs become larger, they often need to be split into more manageable parts. In other languages this is often achieved using include files or packages. In Fortran 77, no such functionality exists. Source files can be grouped in files, but no standard way of including

specific libraries of subroutines or function exists in the language. The C preprocessor is often used to include code from libraries in Fortran, but is not standardised in the language itself.

In Fortran 90 the concept of modules was introduced. A Fortran 90 module can contain both variables, parameters and subroutines. This makes it possible to divide programs into well defined modules which are more easily maintained. The syntax for a module is similar to that of how a main program in Fortran is defined.

```
module module-name
  [specification-stmts]
[contains
  module-subprograms]
end module [module-name]
```

The block *specification-stmts* defines the variables that are available for programs or subroutines using the module. In the block, *module-subprograms*, subroutines in the module are declared. A module can contain only variables or only subroutines or both. One use of this, is to declare variables common to several modules in a separate module. Modules are also a good way to divide a program into logical and coherent parts. Variables and functions in a module can be made private to a module, hiding them from routines using the module. The keywords **public** and **private** can be used to control the access to a variable or a function. In the following code the variable, **a**, is hidden from subroutines or programs using this module. The variable, **b**, is however visible. When nothing is specified in the variable declaration, the variable is assumed to be public.

```
module mymodule

  integer , private :: a
  integer :: b
  ...
```

The ability to hide variables in modules enables the developer to hide the implementation details of a module, reducing the risk of accidental modification variables and use of subroutines used in the implementation.

To access the routines and variables in a module the **use** statement is used. This makes all the public variables and subroutines available in programs and other modules. In the following example illustrate how the subroutines use in the previous examples are placed in a module, **truss**, and used from a main program.

```
module truss

  ! Public variable declarations

  ! Variables that are visible for other programs
  ! and modules

  integer , public , parameter :: &
  ap = selected_real_kind(15,300)
```

```

    ! Private variables declarations

    integer , private :: my_private_variable

contains

subroutine bar3e(ex,ey,ez,ep,Ke)

    implicit none

    real(ap) :: ex(2), ey(2), ez(2), ep(2)
    real(ap) :: Ke(6,6)

    real(ap) :: nxx, nyx, nzx
    real(ap) :: L, E, A
    real(ap) :: Kel(2,2)
    real(ap) :: G(2,6)

    ! Calculate directional cosines

    L = sqrt( (ex(2)-ex(1))**2 + (ey(2)-ey(1))**2 &
             + (ez(2)-ez(1))**2 )

    nxx = (ex(2)-ex(1))/L
    nyx = (ey(2)-ey(1))/L
    nzx = (ez(2)-ez(1))/L

    ! Calculate local stiffness matrix

    Kel(1,:) = (/ 1.0_ap, -1.0_ap /)
    Kel(2,:) = (/ -1.0_ap, 1.0_ap /)

    Kel = Kel * (ep(1)*ep(2)/L)

    G(1,:) = (/ nxx, nyx, nzx, &
               0.0_ap, 0.0_ap, 0.0_ap /)
    G(2,:) = (/ 0.0_ap, 0.0_ap, 0.0_ap, &
               nxx, nyx, nzx /)

    ! Calculate transformed stiffness matrix

    Ke = matmul(matmul(transpose(G),Kel),G)

    return

end subroutine bar3e

subroutine writeMatrix(A)

```

```
    real(ap) :: A(6,6)

    ! Print matrix to standard output

    do i=1,6
        write(*, '(6G10.4)') (A(i,j), j=1,6)
    end do

    return

end subroutine writeMatrix

end module truss
```

Main program using the `truss` module.

```
program module_sample

    use truss

    implicit none

    real(ap) :: ex(2), ey(2), ez(2), ep(2)
    real(ap), allocatable :: Ke(:, :)

    ep(1) = 1.0_ap
    ep(2) = 1.0_ap
    ex(1) = 0.0_ap
    ex(2) = 1.0_ap
    ey(1) = 0.0_ap
    ey(2) = 1.0_ap
    ez(1) = 0.0_ap
    ez(2) = 1.0_ap

    allocate(Ke(6,6))

    call bar3e(ex, ey, ez, ep, Ke)
    call writeMatrix(Ke)

    deallocate(Ke)

    stop

end program module_sample
```

Please note that the declaration of `ap` in the `truss` module is used to define the precision of the variables in the main program.

Chapter 3

Photran

Photran is an integrated development environment, IDE, for Fortran based on the Eclipse project. The user interface resembles the one found in commercial alternatives such as Microsoft Visual Studio or Absoft Fortran. This chapter gives a short introduction on how to get started with this development environment

3.1 Starting Photran

Photran is started by choosing "Programs/Fortran Python Software Pack/Photran IDE" in the start-menu in Windows. When Photran has been started a dialog is shown asking for a location of a workspace directory, see figure 3.1. A workspace is a directory containing Photran configuration and project files. If the checkbox, **Use this as the default...**, is checked this question will not appear the next time Photran is started, the selected workspace will be used by default.

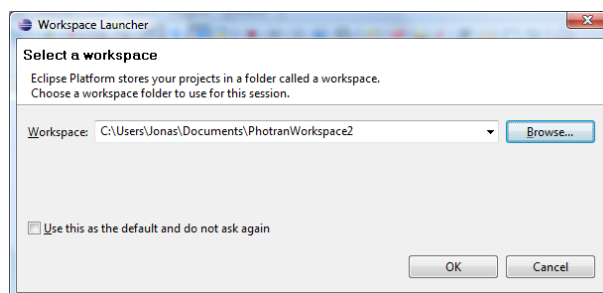
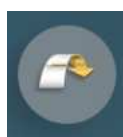


Figure 3.1: Choice of workspace directory

When Photran is started for the first time, a welcome screen is shown. This screen will not be used in this chapter. Click on



to show Photran's normal user interface layout, as shown in figure 3.2

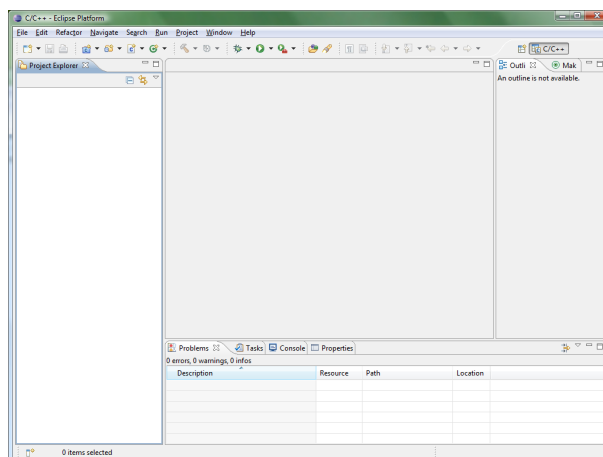


Figure 3.2: Photran default interface layout

3.2 Creating a Photran Makefile project

The Photran IDE is centered around projects. This means that source files and build files are added to projects which Photran are maintained by Photran. Unfortunately, Photran does not have as advanced project management features as its parent project Eclipse. Photran can't generate makefiles automatically from the source files contained in the project. To solve this the Fortran Python Software Pack comes with a Fortran Make File Generator that generates a Makefile from files located in the project directory. The following example shows how to create and configure a Makefile project in Photran.

First a new project is created in Photran by selecting **File/New/Ohter**. This brings up a dialog listing the available project types in Photran, see figure 3.3.

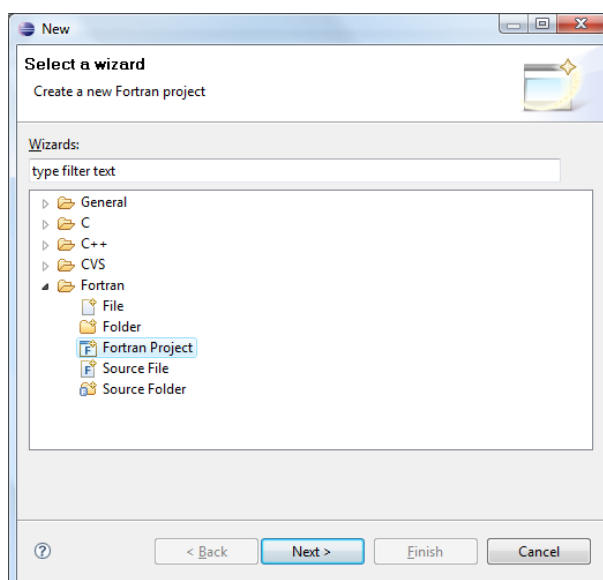


Figure 3.3: Project type

Select **Fortran/Fortran Project** in the list shown. Click **Next**.

In the next page, figure 3.4, enter the name of the project and select the "Makefile project" in the "Project type" list.

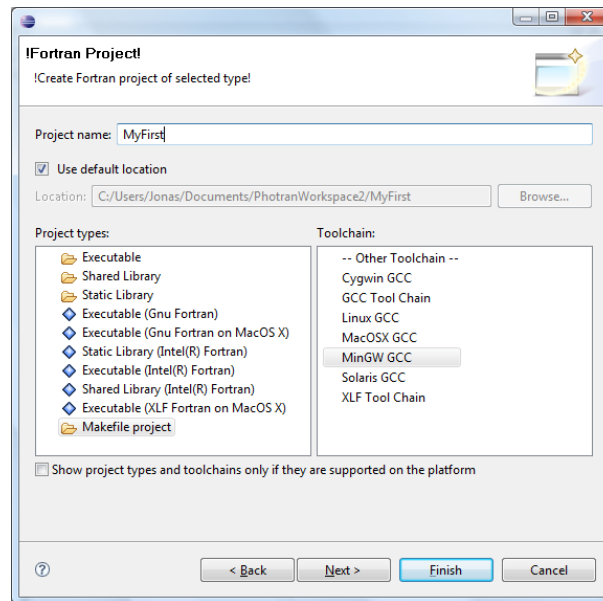


Figure 3.4: Project name and type

To be able to build a project a toolchain must be selected. A toolchain is set of tools and compilers that is used to build a project. Linux user can choose "Linux GCC" or "GCC Toolchain". On Windows the toolchains "GCC Toolchain", "MinGW GCC" or "Cygwin GCC" can be selected depending on the tools installed. If the Fortran Python Software Pack is installed "MinGW GCC" must be chosen for Windows. Windows users should uncheck the box "Show project types and toolchains only if they are supported on the platform.". This will show all available toolchains even if Photran can't detect them. Click **Next** to go to the last configuration page. In this step an error parser must be configured in the advanced settings. Click on **Advanced Settings...**. This brings up the advanced configuration dialog. Select **Fortran Build/Settings**. In the **Binary Parsers** parsers for different executable formats can be selected, see figure 3.5.

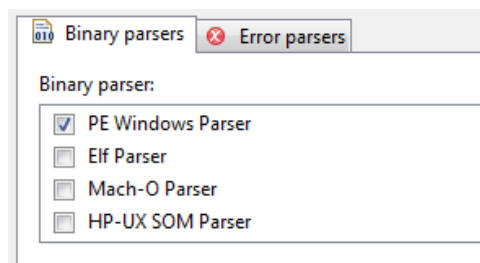


Figure 3.5: Binary parser configuration

On Windows the "PE Windows Parser" should be selected. On Linux the "Elf Parser" should be selected. In the **Error Parsers**, see figure 3.6, parsers for compiler error messages can be selected. The closest match for the gfortran compiler is the "Fortran Error Parser for G95 Fortran" selection.

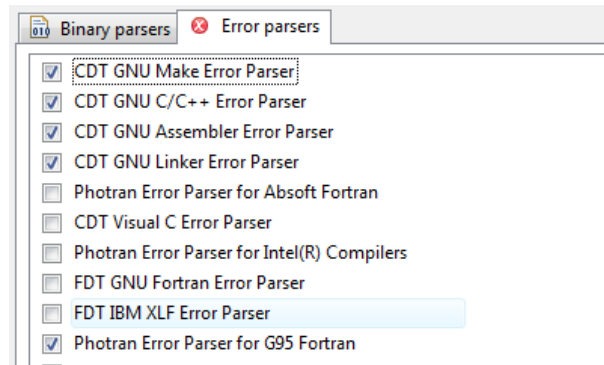


Figure 3.6: Error parser configuration

Click **OK** to save the settings and close the advanced settings dialog. The project is now ready to be created. Click **Finish** to create the project. Before the project is saved Photran shows a dialog with the option of switching to the Fortran Perspective, see figure 3.7.

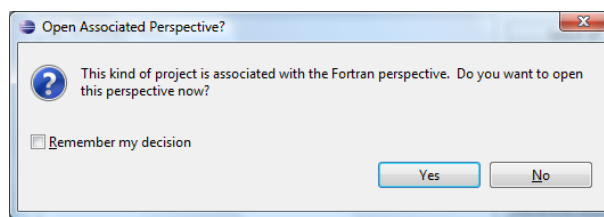


Figure 3.7: Switching to Fortran Perspective

A perspective in Photran is a pre-configured layout of the development environment. Photran comes with a Fortran perspective and a Fortran Debug perspective used when debugging Fortran applications.

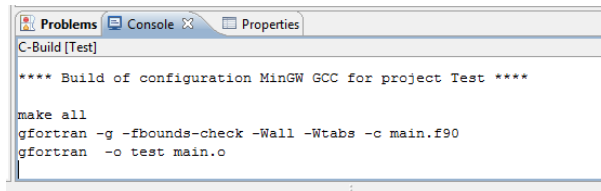
3.2.1 Adding a new source file

A new source file is added to the project by selecting **File/New/Other** and selecting "Source File" from the Fortran Folder. Click **Next**. In the next page the name of the source file is entered. Click **Finish** to create the file and add it to the project.

3.3 Building the project

When all source files have been added to the project it can be built from the **Project/Build All...** menu. This will execute the build process. Output from the process can be seen in the **Console** tab in the lower pane of the Photran window, as shown in figure 3.8.

Any errors in the build process are also shown in this tab.



```
Problems Console Properties
C-Build [Test]

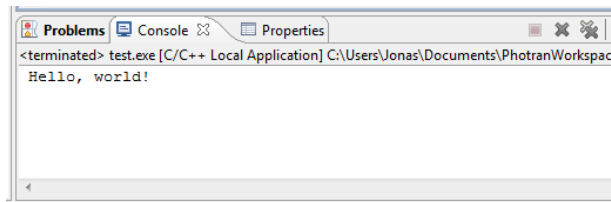
**** Build of configuration MinGW GCC for project Test ****

make all
gfortran -g -fbounds-check -Wall -Wtabs -c main.f90
gfortran -o test main.o
```

Figure 3.8: Output from the build process

3.4 Running the project

When the project has been built successfully it can be run by selecting **Run/Run** from the menu. Output from the program is redirected to the **Console** in the lower part of the window, as shown in figure 3.9.



```
Problems Console Properties
<terminated> test.exe [C/C++ Local Application] C:\Users\Jonas\Documents\PhotranWorkspac
Hello, world!
```

Figure 3.9: Output from running program

Bibliography

- [1] Michael Metcalf and John Reid, Fortran 90/95 Explained, Oxford University Press Inc, New York, 1996
- [2] Niels Ottosen & Hans Petersson, Introduction to the Finite Element Method, Prentice Hall International (UK) Ltd, 1992
- [3] Fortran 90, ISO/IEC 1539 : 1991, International Organisation for Standardization, <http://www.iso.ch/>
- [4] CALFEM – A finite element toolbox to MATLAB version 3.3, Division of Structural Mechanics, 1999

Appendix A

Exercises

A.1 Fortran

- 1-1** Which of the following names can be used as Fortran variable names?
- a) `number_of_stars`
 - b) `fortran_is_a_nice_language_to_use`
 - c) `2001_a_space_odyssey`
 - d) `more$_money`
- 1-2** Declare the following variables in Fortran: 2 scalar integers `a`, and `b`, 3 floating point scalars `c`, `d` and `e`, 2 character strings `infile` and `outfile` and a logical variable `f`.
- 1-3** Declare a floating point variable `a` that can represent values between 10^{-150} and 10^{150} with 14 significant numbers.

1-4 What is printed by the following program?

```

program precision

    implicit none

    integer, parameter :: ap = &
        selected_real_kind(15,300)

    real(ap) :: a, b

    a = 1.234567890123456
    b = 1.234567890123456_ap

    if (a==b) then
        write(* ,*) 'Values_are_equal.'
    else
        write(* ,*) 'Values_are_different.'
    endif

    stop

end program precision

```

1-5 Declare a $[3 \times 3]$ floating point array, **Ke**, and an 3 element integer array, **f**

1-6 Declare an integer array, **idx**, with the following indices [0, 1, 2, 3, 4, 5, 6, 7].

1-7 Give the following assignments:
 Floating point array, **A**, is assigned the value 5.0 at (2,3).
 Integer matrix, **C**, is assigned the value 0 at row 2.

1-8 Give the following if-statements:

If the value of the variable, **i**, is greater than 100 print 'i is greater than 100!'

If the value of the logical variable, **extra_filling**, is true print 'Extra filling is ordered.', otherwise print 'No extra filling.'

1-9 Give a case-statement for the variable, **a**, printing 'a is 1' when a is 1, 'a is between 1 and 20' for values between 1 and 20 and prints 'a is not between 1 and 20' for all other values.

- 1-10** Write a program consisting of a do-statement 1 to 20 with the control variable, i . For values, i , between 1 till 5, the value of i is printed, otherwise ' $i > 5$ ' is printed. The loop is to be terminated when i equals 15.
- 1-11** Write a program declaring a floating point matrix, I , with the dimensions $[10 \times 10]$ and initialises it with the identity matrix.
- 1-12** Give the following expressions in Fortran:
- a) $\frac{1}{\sqrt{2}}$
 - b) $e^x \sin^2 x$
 - c) $\sqrt{a^2 + b^2}$
 - d) $|x - y|$
- 1-13** Give the following matrix and vector expressions in Fortran. Also give appropriate array declarations:
- a) \mathbf{AB}
 - b) $\mathbf{A}^T \mathbf{A}$
 - c) \mathbf{ABC}
 - d) $\mathbf{a} \cdot \mathbf{b}$
- 1-14** Show expressions in Fortran calculating maximum, minimum, sum and product of the elements of an array.
- 1-15** Declare an allocatable 2-dimensional floating point array and a 1-dimensional floating point vector. Also show program-statements how memory for these variables are allocated and deallocated.
- 1-16** Create a subroutine, `identity`, initialising a arbitrary two-dimensional to the identity matrix. Write a program illustrating the use of the subroutine.
- 1-17** Implement a function returning the value of the the following expression:
- $$e^x \sin^2 x$$

- 1-18** Write a program listing $f(x) = \sin x$ from -1.0 to 1.0 in intervals of 0.1 . The output from the program should have the following format:

```

                111111111122222222223
123456789012345678901234567890
  x          f(x)
-1.000 -0.841
-0.900 -0.783
-0.800 -0.717
-0.700 -0.644
-0.600 -0.565
-0.500 -0.479
-0.400 -0.389
-0.300 -0.296
-0.200 -0.199
-0.100 -0.100
 0.000  0.000
 0.100  0.100
 0.200  0.199
 0.300  0.296
 0.400  0.389
 0.500  0.479
 0.600  0.565
 0.700  0.644
 0.800  0.717
 0.900  0.783
 1.000  0.841

```

- 1-19** Write a program calculating the total length of a piecewise linear curve. The curve is defined in a textfile `line.dat`.

The file has the following structure:

```

{number of points n in the file}
{x-coordinate point 1} {y-coordinate point 1}
{x-coordinate point 2} {y-coordinate point 2}
.
.
{x-coordinate point n} {y-coordinate point n}

```

The program must not contain any limitations regarding the number of points in the number of points in the curve read from the file.

- 1-20** Declare 3 strings, `c1`, `c2` and `c3` containing the words 'Fortran', 'is' och 'fun'. Merge these into a new string, `c4`, making a complete sentence.
- 1-21** Write a function converting a string into a floating point value. Write a program illustrating the use of the function.
- 1-22** Create a module, `conversions`, containing the function in 1-21 and a function for converting a string to an integer value. Change the program in 1-21 to use this module. The module is placed in a separate file, `conversions.f90` and the main program in `main.f90`.

