Introductory Fortran Programming, Part II

Gunnar Wollan

Dept. of Geosciences, University of Oslo

January 27th, 2006
Outline

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
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 large arrays and their dimensions
- Too many visible details
- Little correspondence between mathematical abstraction and computer code
- Redesign and reimplementation tend to be expensive
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 large arrays and their dimensions
- Too many visible details
- Little correspondence between mathematical abstraction and computer code
- Redesign and reimplementations tend to be expensive
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 large arrays and their dimensions
- Too many visible details
- Little correspondence between mathematical abstraction and computer code
- Redesign and reimplementation tend to be expensive
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 large arrays 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 that approaches the OOP style
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 that approaches the OOP style
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 that approaches the OOP style.
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 that approaches 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 numerical computing (until late 1990s)
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 numerical computing (until late 1990s)
Example: programming with matrices

- Mathematical problem:
  - Matrix-matrix product: $C = MB$
  - Matrix-vector product: $y = Mx$

- Points to consider:
  - What is a matrix
  - How do we program with matrices?
  - Do standard arrays in any computer language give good enough support for matrices?
Example: programming with matrices

- Mathematical problem:
  - Matrix-matrix product: \( C = MB \)
  - Matrix-vector product: \( y = Mx \)

- Points to consider:
  - What is a matrix
  - How do we program with matrices?
  - Do standard arrays in any computer language give good enough support for matrices?
Example: programming with matrices

- Mathematical problem:
  - Matrix-matrix product: $C = MB$
  - Matrix-vector product: $y = Mx$

- Points to consider:
  - What is a matrix
  - How do we program with matrices?
  - Do standard arrays in any computer language give good enough support for matrices?
Example: programming with matrices

- **Mathematical problem:**
  - Matrix-matrix product: $C = MB$
  - Matrix-vector product: $y = Mx$

- **Points to consider:**
  - What is a matrix
  - How do we program with matrices?
  - Do standard arrays in any computer language give good enough support for matrices?
Example: programming with matrices

What is a matrix?

A well defined mathematical quantity, containing a table of numbers and a set of legal operations
Example: programming with matrices

What is a matrix?
- A well defined mathematical quantity, containing a table of numbers and a set of legal operations
Example: programming with matrices

- How do we program with matrices?
  - By utilizing loops or nested loops
Example: programming with matrices

- How do we program with matrices?
  - By utilizing loops or nested loops
Example: programming with matrices

Do standard arrays in any computer language give good enough support for matrices?

- Both yes and no, we usually have to rely on using nested loops to traverse an array in 2 or more dimensions.
- If the compiler is not properly designed for optimizing loops, the result will be a slow program.
- You have to be aware of the programming language’s way of storing the matrix to avoid indexing the array the wrong way.
Example: programming with matrices

Do standard arrays in any computer language give good enough support for matrices?

- Both yes and no, we usually have to rely on using nested loops to traverse an array in 2 or more dimensions.
- If the compiler is not properly designed for optimizing loops, the result will be a slow program.
- You have to be aware of the programming language's way of storing the matrices to avoid indexing the array the wrong way.
A dense matrix in Fortran 77(1)

**Fortran 77 syntax**

```fortran
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,B,C,p,q,r)

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
A dense matrix in Fortran 77(2)

- Drawback with this implementation
  - Array sizes must be explicitly transferred
  - New routines for different precisions
Working with a dense matrix in Fortran 95

**Code**

```fortran
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
- The computer code is as compact as the mathematical notation
Working with a dense matrix in Fortran 95

**Code**

```fortran
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
- The computer code is as compact as the mathematical notation
Working with a dense matrix in Fortran 95

Code

```fortran
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
- 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 exist to extract the shape and dimension(s) from arrays
- This means we no longer have to pass the array sizes as part of a function call
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 longer have to pass the array sizes as part of a function call.
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 longer have to pass the array sizes as part of a function call
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 exist to extract the shape and dimension(s) from arrays
- This means we no longer have to pass the array sizes as part of a function call
A module is a collection of data structures and operations on them.

- The module is not a new type of variable, but the `TYPE` construct is.
- A module can use other modules so we can create complex units which are easy to program with.
What is this module, class or object

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

The module is not a new type of variable, but the \texttt{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
- Many iterative solution methods for $Au = b$ can operate on the nonzeroes only
Extensions to sparse matrices

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

- Matrix for the discretization of $-\nabla^2 u = f$
- Only $5n$ out of $n^2$ entries are nonzero
- Many iterative solution methods for $Au = b$ can operate on the nonzeros only
How to store sparse matrices (1)

An equation

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

Working with nonzeroes only is important for efficiency.
How to store sparse matrices (1)

An equation

\[
A = \begin{pmatrix}
a_{1,1} & 0 & 0 & a_{1,4} & 0 \\
0 & a_{2,2} & a_{2,3} & 0 & a_{2,5} \\
0 & a_{3,2} & a_{3,3} & 0 & 0 \\
a_{4,1} & 0 & 0 & a_{4,4} & a_{4,5} \\
0 & a_{5,2} & 0 & a_{5,5} & a_{5,5}
\end{pmatrix}
\]  

Working with nonzeroes only is important for efficiency
How to store sparse matrices (2)

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

\[
A = (a_1, 1, a_1, 4, a_2, 2, a_2, 3, a_2, 5, \ldots)
\]
\[
irow = (1, 3, 6, 8, 11, 14) \quad (2)
\]
\[
jcol = (1, 4, 2, 3, 5, 2, 3, 1, 4, 5, 2, 4, 5)
\]

- more complicated data structures and hence more complicated programs
The nonzeros can be stacked in a one-dimensional array

We need two extra arrays to tell where a column starts and the row index of a nonzero

\[
\begin{align*}
A & = (a_1, 1, a_1, 4, a_2, 2, a_2, 3, a_2, 5, \ldots) \\
irow & = (1, 3, 6, 8, 11, 14) \\
jcol & = (1, 4, 2, 3, 5, 2, 3, 1, 4, 5, 2, 4, 5)
\end{align*}
\]  (2)

more complicated data structures and hence more complicated programs
The nonzeroes can be stacked in a one-dimensional array

We need two extra arrays to tell where a column starts and the row index of a nonzero

\[
A = (a1, 1, a1, 4, a2, 2, a2, 3, a2, 5, \ldots)
\]

\[
irow = (1, 3, 6, 8, 11, 14)
\]

\[
jcol = (1, 4, 2, 3, 5, 2, 3, 1, 4, 5, 2, 4, 5)
\]

⇒ more complicated data structures and hence more complicated programs
Sparse matrices in Fortran 77

**Code example for** $y = Mx$

```fortran
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 storage structure (5 args)
- Different name for two functions that perform the same task on two different matrix formats
Sparse matrices in Fortran 77

- **Code example for** $y = Mx$
  
  ```fortran
  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 storage structure (5 args)
  - Different name for two functions that perform the same task on two different matrix formats
Sparse matrices in Fortran 77

- Code example for $y = Mx$
  
  ```fortran
  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 storage 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)

A module

```fortran
MODULE mattypes
  TYPE sparse
    DOUBLE PRECISION, POINTER :: A(:) ! long vector with nonzero 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 nonzeroes
  END TYPE sparse
END MODULE mattypes
```
A module

```
MODULE mathsparse
  USE mathtypes
  TYPE(sparse),PRIVATE :: hidden_sparse
CONTAINS
  SUBROUTINE prod(x, z)
    DOUBLE PRECISION, POINTER :: x(:), z(:)
    ...
  END SUBROUTINE prod
END MODULE mathsparse
```
Sparse matrix as a Fortran 95 module (3)

- 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
Sparse matrix as a Fortran 95 module(3)

- 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
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
Sparse matrix as a Fortran 95 module (3)

- 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
Sparse matrix as a Fortran 95 module (3)

- 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 difference 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
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 difference 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
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 difference 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
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 difference 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
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
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
Bad news

- Programming with modules can be a great thing, but it might be \textit{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.
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
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
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.
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.
In Fortran 77 we had to use what is called a common block for global variables.

This common block is used to give a name to the part of the memory where we have global variables.

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

One problem here is that until late nineties the variables in a common block was position dependent.
In Fortran 77 we had to use what is called a common block for global variables.

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 until late nineties the variables in a common block was position dependent.
In Fortran 77 we had to use what is called a common block for global variables.

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 until late nineties the variables in a common block was position dependent.
Common blocks

- An example of an error in the use of a common block
  
  ```fortran
  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
- The good news is that as mentioned the new Fortran compilers use the variable names instead of position in a common block
Common blocks

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

```fortran
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

- The good news is that as mentioned the new Fortran compilers use the variable names instead of position in a common block
Common blocks

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

```fortran
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 \( x \)

- The good news is that as mentioned the new Fortran compilers use the variable names instead of position in a common block.
Common blocks

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

```fortran
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.
- The good news is that as mentioned the new Fortran compilers use the variable names instead of position in a common block.
An example of an error in the use of a common block

```fortran
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.

The good news is that as mentioned the new Fortran compilers use the variable names instead of position in a common block.
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:

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

```fortran
MODULE global
  INTEGER :: i, j
  REAL :: x, y
END MODULE global
```
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 variables
  
  ```fortran
  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
Use a module for global variables(2)

- Accessing a module and its variables
  
  ```fortran
  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
Use a module for global variables(2)

- Accessing a module and its variables
  
  ```fortran
  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
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
We have a derived datatype:

```fortran
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 by using the same operators as for ordinary integer and real variables we need to create a module which does the job.

We want to overload the operators `+`, `-`, `*`, `/`, `=`
We have a derived datatype:

```fortran
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 by using the same operators as for ordinary integer and real variables we need to create a module which does the job.

We want to overload the operators `+`, `-`, `*`, `/`, `=`
Doing arithmetic on derived datatypes

- We have a derived datatype:

  ```fortran
  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 by using the same operators as for ordinary integer and real variables we need to create a module which does the job.

- We want to overload the operators `+`, `-`, `*`, `/`, `=`
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?
What is operator overloading?

By this we mean that we extend the functionality of the intrinsic operators +, -, *, /, = to also perform the operations on other datatypes.

How do we do this in Fortran 95?
What is operator overloading?

By this we mean that we extend the functionality of the intrinsic operators `+`, `-`, `*`, `/`, `=` to also perform the operations on other datatypes.

How do we do this in Fortran 95?
This is how:

```fortran
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.
This is how:

```fortran
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.
This is how:

```fortran
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.
What the do we do to extend the equal operator?

Not so very different than from the others

```fortran
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
```
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
```
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 INTERFACE ASSIGNMENT.
- The difference between the assignment and operator implementation is that the assignment is implemented using a subroutine while the others usually are implemented using a function.
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 INTERFACE ASSIGNMENT.
- The difference between the assignment and operator implementation is that the assignment is implemented using a subroutine while the others usually are implemented using a function.
The multiplication operator for mytype

```fortran
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 kept in a separate file.
The multiplication operator for mytype

```fortran
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 kept in a separate file.
Implementation of the multiplication operator (2)

- The multiplication operator for mytype

  ```fortran
  ... 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
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

  ```fortran
  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 kept in a separate file
How we implement the assignment operator

The assignment operator for mytype

```fortran
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 kept in a separate file.
What have we really done in these two examples(1)

- The multiply function takes two input arguments and returns the result of the multiplication in a variable of mytype.
- To avoid mistakes of changing the value of any of the two arguments both have the attribute `INTENT(IN)`.
- In C++ we would typically use the keyword `const` as an attribute to the argument.
- The default attribute for arguments to functions and subroutines in Fortran is `INTENT(INOUT)` which allows us to modify the value of 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 do not need to go through one or more loops to perform the multiplication.
What have we really done in these two examples (1)

- The multiply function takes two input arguments and returns the result of the multiplication in a variable of mytype.
- To avoid mistakes of changing the value of any of the two arguments both have the attribute \texttt{INTENT(IN)}.
- In C++ we would typically use the keyword \texttt{const} as an attribute to the argument.
- The default attribute for arguments to functions and subroutines in Fortran is \texttt{INTENT(INOUT)} which allows us to modify the value of 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 do not need to go through one or more loops to perform the multiplication.
The multiply function takes two input arguments and returns the result of the multiplication in a variable of mytype.

To avoid mistakes of changing the value of any of the two arguments both have the attribute \texttt{INTENT(IN)}.

In C++ we would typically use the keyword \texttt{const} as an attribute to the argument.

The default attribute for arguments to functions and subroutines in Fortran is \texttt{INTENT(INOUT)} which allows us to modify the value of 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 do not need to go through one or more loops to perform the multiplication.
What have we really done in these two examples(1)

- The multiply function takes two input arguments and returns the result of the multiplication in a variable of mytype.
- To avoid mistakes of changing the value of any of the two arguments both have the attribute `INTENT(IN)`.
- In C++ we would typically use the keyword `const` as an attribute to the argument.
- The default attribute for arguments to functions and subroutines in Fortran is `INTENT(INOUT)`, which allows us to modify the value of 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 do not need to go through one or more loops to perform the multiplication.
What have we really done in these two examples(1)

- The multiply function takes two input arguments and returns the result of the multiplication in a variable of mytype.
- To avoid mistakes of changing the value of any of the two arguments both have the attribute **INTENT(IN)**.
- In C++, we would typically use the keyword `const` as an attribute to the argument.
- The default attribute for arguments to functions and subroutines in Fortran is **INTENT(INOUT)** which allows us to modify the value of 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 do not need to go through one or more loops to perform the multiplication.
The assignment is implemented using a subroutine:

- Like in the multiplication 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.
What have we really done in these two examples(2)

- The assignment is implemented using a subroutine
- Like in the multiplication we use the \textit{INTENT(IN)} attribute to the second argument
- To the first argument we use the \textit{INTENT(OUT)} attribute to signal that this argument is for the return of a value only
What have we really done in these two examples (2)

- The assignment is implemented using a subroutine.
- Like in the multiplication, 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.
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
To attempt to get some OOP functionality for a Fortran 95 module we will attempt to write a *class*-like module.

- This module is based on a C++ class called *MyVector*.
- We will begin by define a set of datatypes containing vector definitions for the three standard datatypes *double precision*, *real* and *integer*.
- We keep the various parts of the code in separate files.
To attempt to get some OOP functionality for a Fortran 95 module we will attempt to write a *class*-like module.

This module is based on a C++ class called *MyVector*.

We will begin by define a set of datatypes containing vector definitions for the three standard datatypes *double precision*, *real* and *integer*.

We keep the various parts of the code in separate files.
To attempt to get some OOP functionality for a Fortran 95 module we will attempt to write a \textit{class}-like module.

This module is based on a \texttt{C++} class called \textit{MyVector}.

We will begin by define a set of datatypes containing vector definitions for the three standard datatypes \textit{double precision}, \textit{real}, \textit{and integer}.

We keep the various parts of the code in separate files.
A Fortran 95 “class”

- To attempt to get some OOP functionality for a Fortran 95 module we will attempt to write a class-like module.
- This module is based on a C++ class called MyVector.
- We will begin by define a set of datatypes containing vector definitions for the three standard datatypes double precision, real and integer.
- We keep the various parts of the code in separate files.
Vector definitions

The mytypes module

```fortran
MODULE mytypes
  TYPE dpvector
    DOUBLE PRECISION, POINTER :: v(:)
  END TYPE dpvector

  TYPE spvector
    REAL, POINTER :: v(:)
  END TYPE spvector

  TYPE ivector
    INTEGER, POINTER :: v(:)
  END TYPE ivector
END MODULE mytypes
```
We begin by define some overloading of operators

```fortran
MODULE myvectordefs
  INTERFACE myvector
    MODULE PROCEDURE myvectord
    MODULE PROCEDURE myvectors
    MODULE PROCEDURE myvectori
  END INTERFACE

  ...

  ...
```

We now can use myvector for all three types of vectors defined in mytypes
We begin by define some overloading of operators

```fortran
MODULE myvectordefs
  INTERFACE myvector
    MODULE PROCEDURE myvectord
    MODULE PROCEDURE myvectors
    MODULE PROCEDURE myvectori
  END INTERFACE

  ...
```

We now can use myvector for all three types of vectors defined in mytypes
The double precision vector allocation subroutine

CONTAINS

SUBROUTINE myvectord(p,n)
USE mytypes, ONLY: dpvector
IMPLICIT NONE
TYPE(dpvector), POINTER :: p
INTEGER, INTENT(IN) :: n
INTEGER :: rstat
ALLOCATE(p,STAT=rstat)
IF (rstat /= 0) THEN
   PRINT *, 'Error in allocating mytype, ',
   rstat
   NULLIFY(p)
RETURN
END IF
...

The double precision vector allocation subroutine cont.

... 
ALLOCATE(p%v(n),STAT=rstat)
IF (rstat /= 0) THEN
  PRINT *, 'Error in allocating vector, ',&
  rstat
  NULLIFY(p)
  RETURN
END IF
p%v = 0.
END SUBROUTINE myvectorsd
The myvector module (4)

Multiplication of two vectors

```fortran
INTERFACE OPERATOR(*)
    SUBROUTINE multiplyd(a,b)
        USE mytypes, ONLY: dpvector
        TYPE(dpvector), INTENT(IN) :: a, b
        TYPE(dpvector), INTENT(OUT) :: c
    END FUNCTION multiplyd
END INTERFACE
```

Wollan
Introductory Fortran Programming, Part II
The myvector module(5)

Copying one vector to another

```fortran
INTERFACE ASSIGNMENT (=)
  SUBROUTINE equald(a,b)
    USE mytypes, ONLY: dpvector
    TYPE(dpvector), INTENT(IN) :: b
    TYPE(dpvector), INTENT(OUT) :: a
    END SUBROUTINE equald
  END INTERFACE
```

Wollan
Introductory Fortran Programming, Part II
The myvector module (6)

The multiplication of a double precision vector

The source code in a separate file and separately compiled

```
SUBROUTINE multiplyd(a,b,c)
  USE mytypes, ONLY: dpvector
  IMPLICIT NONE
  TYPE(dpvector), INTENT(IN) :: a, b
  TYPE(dpvector), INTENT(OUT) :: c
  IF(ASSOCIATED(c%v)) DEALLOCATE(c%v)
  ALLOCATE(c%v(SIZE(a%v)))
  c%v(:) = a%v(:) * b%v(:)
END SUBROUTINE multiplyd
```
The myvector module

- The multiplication of a double precision vector
- The source code in a separate file and separately compiled

```fortran
SUBROUTINE multiplyd(a,b,c)
  USE mytypes, ONLY: dpvector
  IMPLICIT NONE
  TYPE(dpvector), INTENT(IN) :: a, b
  TYPE(dpvector), INTENT(OUT) :: c
  IF(ASSOCIATED(c%v)) DEALLOCATE(c%v)
  ALLOCATE(c%v(SIZE(a%v)))
  c%v(:) = a%v(:) * b%v(:)
END SUBROUTINE multiplyd
```
The copying of a double precision vector

```fortran
SUBROUTINE equald(a,b)
    USE mytypes, ONLY: dpvector
    IMPLICIT NONE
    TYPE(dpvector), INTENT(OUT) :: a
    TYPE(dpvector), INTENT(IN) :: b
    a%v(:) = b%v(:)
END SUBROUTINE equald
```
To see if this module does what it is meant to do we need to test it.

This is a small program testing the module:

```fortran
PROGRAM t2
  USE mytypes
  USE myvectordefs
  TYPE(dpvector), POINTER :: p1, p2, p3
  CALL myvector(p1,3)
  p1%v(:) = 1
  CALL myvector(p2,3)
  p2%v(:) = 6
  CALL myvector(p3,3)
  p3%v(:) = 3
  ...
```
To see if this module does what it meant to do we need to test it.

This is a small program testing the module

```fortran
PROGRAM t2
    USE mytypes
    USE myvectordefs
    TYPE(dpvector), POINTER :: p1, p2, p3
    CALL myvector(p1,3)
    p1%v(:) = 1
    CALL myvector(p2,3)
    p2%v(:) = 6
    CALL myvector(p3,3)
    p3%v(:) = 3
    ...
```
The rest of the program

...  
  p3 = p1 * p2  
  PRINT *, 'Mul: ', p3%v(:)  
  p3 = p1 / p2  
  PRINT *, 'Div: ', p3%v(:)  
  p3 = p1 + p2  
  PRINT *, 'Add: ', p3%v(:)  
  p3 = p1 - p2  
  PRINT *, 'Sub: ', p3%v(:)  
  p3 = p2  
  PRINT *, 'Equ: ', p3%v(:)  
  END PROGRAM t2
The result of testing myarray module

Running the program gave this result:

```
bullet.uio.no> t2
   Mul:  6.00000000000000  6.00000000000000  6.00000000000000
   Div:  0.166666666666667  0.166666666666667  0.166666666666667
   Add:  7.00000000000000  7.00000000000000  7.00000000000000
  Sub: -5.00000000000000 -5.00000000000000 -5.00000000000000
  Equ:  6.00000000000000  6.00000000000000  6.00000000000000
bullet.uio.no>
```

It seems the program works as expected
The result of testing myarray module

Running the program gave this result:

```
bullet.uio.no> t2
_mul: 6.000000000000000 6.000000000000000 6.000000000000000
_div: 0.166666666666667 0.166666666666667 0.166666666666667
_add: 7.000000000000000 7.000000000000000 7.000000000000000
_sub: -5.000000000000000 -5.000000000000000 -5.000000000000000
_equ: 6.000000000000000 6.000000000000000 6.000000000000000
bullet.uio.no>
```

It seems the program works as expected
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
Let us start with some questions

- Why do we optimise our program
- When do we optimise it
- What to optimise
- How to optimise

Now we shall look at some answers to these questions
Let us start with some questions

- Why do we optimise our program
- When do we optimise it
- What to optimise
- How to optimise

Now we shall look at some answers to these questions
Let us start with some questions

- Why do we optimise our program
- When do we optimise it
- What to optimise
- How to optimise

Now we shall look at some answers to these questions
Why do we optimise our program

- Whether we are scientists or students we need our big computer models to execute as fast as possible to get our result before the financing runs out.
- For Ph.D. and Master students this is especially important since they "live on borrowed time".
Why do we optimise our program

- Whether we are scientists or students we need our big computer models to execute as fast as possible to get our result before the financing runs out.
- For Ph.D. and Master students this is especially important since they "live on borrowed time"
Program is very time consuming

- As mentioned students and researchers both need the results of their computations as fast as possible
- Long execution time means less time to analyse the results

- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
  - The program is running on a regular schedule
When do we optimise

- Program is very time consuming
  - As mentioned students and researchers both need the results of their computations as fast as possible
  - Long execution time means less time to analyse the results
- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
- The program is running on a regular schedule
When do we optimise

- Program is very time consuming
  - As mentioned students and researchers both need the results of their computations as fast as possible
  - Long execution time means less time to analyse the results
- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
  - The program is running on a regular schedule
  - Think about a large meteorological model for weather forecast. This model is run several times a day and is time consuming
When do we optimise

- Program is very time consuming
  - As mentioned students and researchers both need the results of their computations as fast as possible
  - Long execution time means less time to analyse the results
- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
- The program is running on a regular schedule
  - Think about a large meteorological model for weather forecast
  - This model is run several times a day and is time consuming
When do we optimise

- Program is very time consuming
  - As mentioned students and researchers both need the results of their computations as fast as possible
  - Long execution time means less time to analyse the results
- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
- The program is running on a regular schedule
  - Think about a large meteorological model for weather forecast
  - This model is run several times a day and is time consuming
When do we optimise

- Program is very time consuming
  - As mentioned students and researchers both need the results of their computations as fast as possible
  - Long execution time means less time to analyse the results
- Increase the resolution of the model
  - The need to increase the resolution of the model in time and/or space
- The program is running on a regular schedule
  - Think about a large meteorological model for weather forecast
  - This model is run several times a day and is time consuming
The development cycle of a program

- Use library routines if they do the job for you, it is a waste of time to invent the wheel over and over again.
- Use the Make utility or other similar tools to speed up compile time, it is time consuming to compile parts of a program already compiled.
What to optimise (1)

- The development cycle of a program
  - Use library routines if they do the job for you, it is a waste of time to invent the wheel over and over again
  - Use the Make utility or other similar tools to speed up compile time, it is time consuming to compile parts of a program already compiled
Use the compiler. It has many options helping you find bugs in your program.

- CPU cycles, where do they go?
  - The main CPU consumption in a Fortran program is looping through arrays.
  - This is specially important for arrays with two or more dimensions.
What to optimise (2)

- Use the compiler. It has many options helping you find bugs in your program
- CPU cycles, where do they go?
  - The main CPU consumption in a Fortran program is looping through arrays
  - This is specially important for arrays with two or more dimensions
What to optimise(2)

- Use the compiler. It has many options helping you find bugs in your program.
- CPU cycles, where do they go?
  - The main CPU consumption in a Fortran program is looping through arrays.
  - This is specially important for arrays with two or more dimensions.
Memory usage can be a bottleneck
- Large programs will be slow especially if your computer has little main memory
- The organisation of your program also influence memory usage

I/O
- Only perform I/O when necessary
- Avoid debug printout if possible
- Read as large a chunk of data as possible from disk to memory
What to optimise (3)

- Memory usage can be a bottleneck
  - Large programs will be slow especially if your computer has little main memory
  - The organisation of your program also influence memory usage

- I/O
  - Only perform I/O when necessary
  - Avoid debug printout if possible
  - Read as large a chunk of data as possible from disk to memory
What to optimise (3)

- Memory usage can be a bottleneck
  - Large programs will be slow especially if your computer has little main memory
  - The organisation of your program also influence memory usage

- I/O
  - Only perform I/O when necessary
  - Avoid debug printout if possible
  - Read as large a chunk of data as possible from disk to memory
What to optimise (3)

- Memory usage can be a bottleneck
  - Large programs will be slow especially if your computer has little main memory
  - The organisation of your program also influence memory usage
- I/O
  - Only perform I/O when necessary
  - Avoid debug printout if possible
  - Read as large a chunk of data as possible from disk to memory
The compiler

- Know your compiler, it has several options to increase the speed of your program
- Use the man-pages to get information about the options
- The options can be formulated differently for compilers from two providers

Some important options

- O[n]: tells the compiler to optimise the program using the optimising level [n]
- -inline: tells the compiler to include small functions and subroutines directly into the program instead of calling them. This avoids shuffling of data in and from functions.
How to optimise(1)

- **The compiler**
  - Know your compiler, it has several options to increase the speed of your program
  - Use the man-pages to get information about the options
  - The options can be formulated differently for compilers from two providers

- **Some important options**
  - `-O[n]`, tells the compiler to optimise the program using the optimising level `[n]`
  - `-inline`, tells the compiler to include small functions and subroutines directly into the program instead of calling them. This avoids shuffling of data to and from functions
How to optimise (1)

- **The compiler**
  - Know your compiler, it has several options to increase the speed of your program
  - Use the man-pages to get information about the options
  - The options can be formulated differently for compilers from two providers

- **Some important options**
  - `-O[n]`, tells the compiler to optimise the program using the optimising level `[n]`
  - `-inline`, tells the compiler to include small functions and subroutines directly into the program instead of calling them. This avoids shuffling of data to and from functions
The compiler

- Know your compiler, it has several options to increase the speed of your program
- Use the man-pages to get information about the options
- The options can be formulated differently for compilers from two providers

Some important options

- \(-O[n]\), tells the compiler to optimise the program using the optimising level \([n]\)
- \(-inline\), tells the compiler to include small functions and subroutines directly into the program instead of calling them. This avoids shuffling of data to and from functions
Finding the bottlenecks in your program

- Sometimes our program is performing really slow, but we are unable to directly see what is the cause of this.
- A couple of useful tools are available to let us study the behaviour or our program.
- The prof utility: This is providing information per function or subroutine as a total including the call of other functions and subroutines.
- The gprof utility: Gives more detailed information on functions and subroutines.
Finding the bottlenecks in your program

- Sometimes our program is performing really slow, but we are unable to directly see what is the cause of this
- A couple of useful tools are available to let us study the behaviour or our program
  - The prof utility: This is providing information per function or subroutine as a total including the call of other functions and subroutines
  - The gprof utility: Gives more detailed information on functions and subroutines
Finding the bottlenecks in your program

- Sometimes our program is performing really slow, but we are unable to directly see what is the cause of this.
- A couple of useful tools are available to let us study the behaviour or our program.
- The prof utility: This is providing information per function or subroutine as a total including the call of other functions and subroutines.
- The gprof utility: Gives more detailed information on functions and subroutines.
Finding the bottlenecks in your program

- Sometimes our program is performing really slow, but we are unable to directly see what is the cause of this.
- A couple of useful tools are available to let us study the behaviour of our program.
- The prof utility: This is providing information per function or subroutine as a total including the call of other functions and subroutines.
- The gprof utility: Gives more detailed information on functions and subroutines.
Example of the output of the gprof

**Gprof listing**

Flat profile:
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>Time</th>
<th>Seconds</th>
<th>Self</th>
<th>Calls</th>
<th>Self/call</th>
<th>Total/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.90</td>
<td>3.05</td>
<td>3.05</td>
<td>19660800</td>
<td>0.00</td>
<td>0.00</td>
<td>f1_</td>
</tr>
<tr>
<td>17.77</td>
<td>3.72</td>
<td>0.67</td>
<td>196608</td>
<td>0.00</td>
<td>0.00</td>
<td>s1_</td>
</tr>
<tr>
<td>1.33</td>
<td>3.77</td>
<td>0.05</td>
<td>1</td>
<td>0.05</td>
<td>3.77</td>
<td>MAIN__</td>
</tr>
</tbody>
</table>
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
Exercise 7: Getting acquainted with modules (1)

Make a small program with the following code:

```fortran
MODULE x
    INTEGER, PRIVATE :: i, j
CONTAINS
    SUBROUTINE init(ni, nj)
        i = ni; j = nj
    END
    SUBROUTINE myprint()
        PRINT *, 'i=',i,' j=',j
    END
END MODULE x
```

plus the main program testing MODULE `x`

```fortran
CALL init(3,9); CALL myprint()
```

You must fill in all the missing parts in the module and the main program.
Exercise 7: Getting acquainted with modules (1)

Make a small program with the following code:

```fortran
MODULE x
    INTEGER, PRIVATE :: i, j
CONTAINS
    SUBROUTINE init(ni, nj)
        i = ni; j = nj
    END
    SUBROUTINE myprint()
        PRINT *, 'i=',i,' j=',j
    END
END MODULE x
```

plus the main program testing MODULE x

```fortran
CALL init(3,9); CALL myprint()
```

You must fill in all the missing parts in the module and the main program.
Exercise 7: Getting acquainted with modules (1)

- Make a small program with the following code:

```fortran
MODULE x
    INTEGER, PRIVATE :: i, j
CONTAINS
    SUBROUTINE init(ni, nj)
        i = ni; j = nj
    END
    SUBROUTINE myprint()
        PRINT *, 'i=',i,' j=',j
    END
END MODULE x
```

- plus the main program testing MODULE x

```fortran
CALL init(3,9); CALL myprint()
```

- You must fill in all the missing parts in the module and the main program
Exercise 7: Getting acquainted with modules (2)

- Compile and run and test that the program is working properly
- How can you change the MODULE such that the following code is legal:
  
  ```fortran
  i=5; j=10; CALL myprint()
  ```
Exercise 7: Getting acquainted with modules (2)

- Compile and run and test that the program is working properly.
- How can you change the MODULE such that the following code is legal:
  
  ```fortran
  i=5; j=10; CALL myprint()
  ```
Exercise 8: Working with including files(1)

- Consider the program from the previous exercise
- Place the module declaration in a file xmod.f90

```fortran
MODULE x
...
CONTAINS
...
END MODULE x
```
Exercise 8: Working with including files (1)

- Consider the program from the previous exercise
- Place the module declaration in a file xmod.f90

```fortran
MODULE x
  ...
  CONTAINS
  ...
END MODULE x
```
Exercise 8: Working with including files (2)

Place the main program in another file xmain.f90

```fortran
PROGRAM xmain
   include ‘xmod.f90’
   ...
END PROGRAM xmain
```
Exercise 8: Working with including files (3)

Compile the program

```
ifort -static -O2 -o xmain xmain.f90
```

- Explain why we do not have to precompile the xmod.f90 file and link it to the main program
- Is this a good programming style?
- If you think it is, explain why
- If you think it is not, explain why
Exercise 8: Working with including files (3)

- Compile the program
  
  \[ \text{ifort -static -O2 -o xmain xmain.f90} \]

- Explain why we do not have to precompile the xmod.f90 file and link it to the main program
  
  - Is this a good programming style?
  - If you think it is explain why
  - If you think it is not explain why
Exercise 8: Working with including files

Compile the program

```fortran
ifort -static -O2 -o xmain xmain.f90
```

Explain why we do not have to precompile the xmod.f90 file and link it to the main program

Is this a good programming style?

- If you think it is explain why
- If you think it is not explain why
Exercise 8: Working with including files

- Compile the program
  ```fortran
  ifort -static -O2 -o xmain xmain.f90
  ```

- Explain why we do not have to precompile the xmod.f90 file and link it to the main program

- Is this a good programming style?
  - If you think it is explain why
  - If you think it is not explain why
Exercise 8: Working with including files (3)

- Compile the program
  
  \texttt{ifort -static -O2 -o xmain xmain.f90}

- Explain why we do not have to precompile the xmod.f90 file and link it to the main program

- Is this a good programming style?
  
  If you think it is explain why

  If you think is not explain why
Exercise 9: Implement MODULE myvector

- Type in the code of the module myvector and add what is missing to perform the standard operations $+,-,/,\ast,=$
- Collect the declarations of the modules in one file
- Write the operator overloading functions in one file and the main program in another file
Exercise 9: Implement MODULE myvector

- Type in the code of the module myvector and add what is missing to perform the standard operations $+,-,/,\ast,\equiv$
- Collect the declarations of the modules in one file
- Write the operator overloading functions in one file and the main program in another file
Exercise 9: Implement MODULE myvector

- Type in the code of the module myvector and add what is missing to perform the standard operations $\div, -, \cdot, \cdot, =$
- Collect the declarations of the modules in one file
- Write the operator overloading functions in one file and the main program in another file
Exercise 10: DAXPY(1)

- The mathematical vector operation \( u \leftarrow ax + y \) where \( a \) is scalar and \( x \) and \( y \) are vectors, is often referred to as a DAXPY operation because DAXPY is the Fortran subroutine name for this operation in the standardized BLAS1 library.

- Make a Fortran 95 subroutine

  ```fortran
  SUBROUTINE daxpy (u,a,x,y)
      TYPE(dpvector) :: u, x, y
      DOUBLE PRECISION :: a
      ... 
      END SUBROUTINE daxpy
  ```

- The subroutine is performing a loop over the array entries for computing \( u \).
Exercise 10: DAXPY(1)

- The mathematical vector operation $u \leftarrow ax + y$ where $a$ is scalar and $x$ and $y$ are vectors, is often referred to as a DAXPY operation because DAXPY is the Fortran subroutine name for this operation in the standardized BLAS1 library.

- Make a Fortran 95 subroutine

```fortran
SUBROUTINE daxpy (u,a,x,y)
    TYPE(dpvector) :: u, x, y
    DOUBLE PRECISION :: a
    ...
END SUBROUTINE daxpy
```

- The subroutine is performing a loop over the array entries for computing $u$. 

Wollan
Introductory Fortran Programming, Part II
Exercise 10: DAXPY(1)

The mathematical vector operation \( u \leftarrow ax + y \) where \( a \) is scalar and \( x \) and \( y \) are vectors, is often referred to as a DAXPY operation because DAXPY is the Fortran subroutine name for this operation in the standardized BLAS1 library.

Make a Fortran 95 subroutine

```fortran
SUBROUTINE daxpy (u,a,x,y)
  TYPE(dpvector) :: u, x, y
  DOUBLE PRECISION :: a
  ...
  END SUBROUTINE daxpy

The subroutine is performing a loop over the array entries for computing \( u \)
Exercise 10: DAXPY(2)

- Make a Fortran 95 subroutine
  
  ```fortran
  SUBROUTINE daxpy (u,a,x,y)
    TYPE(dpvector) :: u, x, y
    DOUBLE PRECISION :: a
  
  using overloaded operators in the myvector module
  ```

- Compare the efficiency of the two subroutines (hint: run $10^p$ daxpy operations with vectors of length $10^q$, e.g., with $p = 4$ and $q = 5$)
Exercise 10: DAXPY(2)

- Make a Fortran 95 subroutine

  ```fortran
  SUBROUTINE daxpy (u, a, x, y)
    TYPE(dpvector) :: u, x, y
    DOUBLE PRECISION :: a
  using overloaded operators in the myvector module

  Compare the efficiency of the two subroutines (hint: run $10^p$ daxpy operations with vectors of length $10^q$, e.g., with $p = 4$ and $q = 5$)
Exercise 11: Communicate with C

- Recall one of our first fortran programs hw1.f90 where we used a fortran function \texttt{a2d(argv)} to convert a commandline argument to a double precision number.
- Suppose we wanted to use the similar C function \texttt{atof(argv)} instead.
- How can we manage to combine these two languages?
- Remember Fortran uses the address to variables as arguments, not the variable value.
- Test the use of the C function \texttt{atof} in a Fortran 95 program:
  \begin{verbatim}
  r = atof(argv)
  \end{verbatim}
Exercise 11: Communicate with C

- Recall one of our first fortran programs hw1.f90 where we used a fortran function \texttt{a2d(argv)} to convert a commandline argument to a double precision number.

- Suppose we wanted to use the similar C function \texttt{atof(argv)} instead.

  - How can we manage to combine these two languages?
  - Remember Fortran uses the address to variables as arguments, not the variable value.
  - Test the use of the C function \texttt{atof} in a Fortran 95 program:
    
    \begin{verbatim}
    r = atof(argv)
    \end{verbatim}
Exercise 11: Communicate with C

- Recall one of our first fortran programs hw1.f90 where we used a fortran function \texttt{a2d(argv)} to convert a commandline argument to a double precision number.
- Suppose we wanted to use the similar C function \texttt{atof(argv)} instead.
- How can we manage to combine these two languages?
  - Remember Fortran uses the address to variables as arguments, not the variable value.
  - Test the use of the C function \texttt{atof} in a Fortran 95 program:
    
    \begin{verbatim}
    r = atof(argv)
    \end{verbatim}
Exercise 11: Communicate with C

- Recall one of our first fortran programs hw1.f90 where we used a fortran function \texttt{a2d(argv)} to convert a commandline argument to a double precision number.
- Suppose we wanted to use the similar C function \texttt{atof(argv)} instead.
- How can we manage to combine these two languages?
- Remember Fortran uses the address to variables as arguments not the variable value.

Test the use of the C function \texttt{atof} in a Fortran 95 program:

\begin{verbatim}
r = atof(argv)
\end{verbatim}
Recall one of our first fortran programs hw1.f90 where we used a fortran function $a2d(argv)$ to convert a commandline argument to a double precision number.

Suppose we wanted to use the similar C function atof(argv) instead.

How can we manage to combine these two languages?

Remember Fortran uses the address to variables as arguments not the variable value.

Test the use of the C function atof in a Fortran 95 program

$$r = \text{atof}(argv)$$
Exercise 11: Communicate with C

- Hint: Depending on the Fortran compiler add the following option `-assume nounderscore` since most Fortran compilers adds an underscore to every function and subroutine name.
- Use the man pages for the compiler to check which option is correct for the compiler you will be using.
Exercise 11: Communicate with C

- **Hint:** Depending on the Fortran compiler add the following option `-assume nounderscore` since most Fortran compilers adds an underscore to every function and subroutine name.

- Use the man pages for the compiler to check which option is correct for the compiler you will be using.
Exercise 12: Using tools to identify bottlenecks(1)

- Write a Fortran 95 program

```fortran
PROGRAM slow
  IMPLICIT NONE
  DOUBLE PRECISION :: v1(128,64,20)
  ...
  CALL step1(v1)
END PROGRAM slow
```
Exercise 12: Using tools to identify bottlenecks (2)

- Write the subroutine step1

```fortran
SUBROUTINE step1(arg1)
   IMPLICIT NONE
   DOUBLE PRECISION :: arg1(:,:,:)
   ...
   DO k = 1, l1; DO j = 1, l2; DO i = 1, l3
      arg1(i,j,k) = f1(arg(i,j,k))
   END DO; END DO; END DO;
END SUBROUTINE step1
```

- Get the size of the array using the `SIZE(arg1, n)` function where the `n` is the dimension number starting at 1
Exercise 12: Using tools to identify bottlenecks (2)

- Write the subroutine step1
  
  ```fortran
  SUBROUTINE step1(arg1)
    IMPLICIT NONE
    DOUBLE PRECISION :: arg1(:,:,:)
    ...
    DO k = 1, l1; DO j = 1, l2; DO i = 1, l3
      arg1(i,j,k) = f1(arg(i,j,k))
    END DO; END DO; END DO;
  END SUBROUTINE step1
  ```

- Get the size of the array using the `SIZE(arg1, n)` function where the `n` is the dimension number starting at 1.
Exercise 12: Using tools to identify bottlenecks(3)

Write the function f1

```fortran
DOUBLE PRECISION FUNCTION f1(x)
   IMPLICIT NONE
   DOUBLE PRECISION :: x
   DO i = 1, 100
      y = y + x ** 3.14
   END DO
   f1 = y
END FUNCTION f1
```

Remember to always use `IMPLICIT NONE` in all your functions and subroutines
Write the function \( f_1 \)

\[
\text{DOUBLE PRECISION FUNCTION } f_1(x) \\
\text{IMPLICIT NONE} \\
\text{DOUBLE PRECISION :: } x \\
\text{DO } i = 1, 100 \\
\quad y = y + x ** 3.14 \\
\text{END DO} \\
\text{f1 = y} \\
\text{END FUNCTION f1}
\]

Remember to always use \textit{IMPLICIT NONE} in all your functions and subroutines.
Exercise 12: Using tools to identify bottlenecks

- Compile the program and run it taking the time (i.e. time prog)
- Compile the program with the -p option and run the program again
- Now run the program using the gprof utility: 
  \texttt{gprof prog > prog.out}
- Look at the contents of the prog.out file and identify the bottleneck
- Suggest improvements (if there are any) for the program in order to increase speed
Exercise 12: Using tools to identify bottlenecks

- Compile the program and run it taking the time (i.e. `time prog`)
- Compile the program with the `-p` option and run the program again
- Now run the program using the `gprof` utility:
  
  ```bash
  gprof prog > prog.out
  ```
- Look at the contents of the `prog.out` file and identify the bottleneck
- Suggest improvements (if there are any) for the program in order to increase speed
Exercise 12: Using tools to identify bottlenecks

- Compile the program and run it taking the time (i.e. time `time prog`)
- Compile the program with the `-p` option and run the program again
- Now run the program using the `gprof` utility:
  `gprof prog > prog.out`
- Look at the contents of the `prog.out` file and identify the bottleneck
- Suggest improvements (if there are any) for the program in order to increase speed
Exercise 12: Using tools to identify bottlenecks

- Compile the program and run it taking the time (i.e. `time prog`)
- Compile the program with the `-p` option and run the program again
- Now run the program using the `gprof` utility:
  
  ```
gprof prog > prog.out
  ```
- Look at the contents of the `prog.out` file and identify the bottleneck
- Suggest improvements (if there are any) for the program in order to increase speed
Exercise 12: Using tools to identify bottlenecks

- Compile the program and run it taking the time (i.e. time prog)
- Compile the program with the -p option and run the program again
- Now run the program using the gprof utility: `gprof prog > prog.out`
- Look at the contents of the prog.out file and identify the bottleneck
- Suggest improvements (if there are any) for the program in order to increase speed
1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
A vector sorting utility

- Often we need to sort a vector in ascending or descending order
- One way of sorting this is to use the quicksort algorithm which is a *divide and conquer* method
A vector sorting utility

- Often we need to sort a vector in ascending or descending order
- One way of sorting this is to use the quicksort algorithm which is a *divide and conquer* method
The quicksort algorithm

- The essence of quicksort is to sort an array by picking some key value in the array as a pivot element around which to rearrange the elements in the array.
- The idea is to permute the elements in the array so that for an index $i$ all elements with key less than the pivot value appears in the lower part of the array and those with keys greater or even with the pivot value appears in the upper part of the array.
- Then we apply the quicksort recursively on the two parts of the array.
The quicksort algorithm

- The essence of quicksort is to sort an array by picking some key value in the array as a pivot element around which to rearrange the elements in the array.
- The idea is to permute the elements in the array so that for an index $i$ all elements with key less than the pivot value appears in the lower part of the array and those with keys greater or even with the pivot value appears in the upper part of the array.
- Then we apply the quicksort recursively on the two parts of the array.
The essence of quicksort is to sort an array by picking some key value in the array as a pivot element around which to rearrange the elements in the array.

The idea is to permute the elements in the array so that for an index $i$ all elements with key less than the pivot value appears in the lower part of the array and those with keys greater or even with the pivot value appears in the upper part of the array.

Then we apply the quicksort recursively on the two parts of the array.
Finding the pivot index (1)

Let us sketch a function returning the pivot index

```fortran
INTEGER FUNCTION findpivot(array, startpoint, endpoint)
  ...
  DO WHILE(.NOT. found)
      IF(i>endpoint)
          findpivot = 0
          EXIT
      END IF
  END DO
END FUNCTION findpivot
```
Finding the pivot index(2)

The `findpivot` function

```fortran
INTEGER FUNCTION findpivot(array, startpoint, endpoint)
  
  DO WHILE(.NOT. found) !// continued
    
    IF(array(i) > array(startpoint))
      findpivot = i
      EXIT
    ELSE IF(array(i) < array(startpoint))
      findpivot = startpoint
      EXIT
    ELSE
      findpivot = 0
      EXIT
    END IF
  END DO
```
After we find the pivot index we need to partition the array before recursively calling quicksort again with the new array partitions.

```fortran
INTEGER FUNCTION partition(startpoint, endpoint, &
pivot, array)
...
DO
  tmp = array(left)
  array(left) = array(right)
  array(right) = tmp
  DO WHILE (array(left) < pivot)
    left = left + 1
  END DO
  DO WHILE (array(right) >= pivot)
    right = right - 1
  END DO
IF (left > right) EXIT
END DO
partition = left
```

Wollan
Introductory Fortran Programming, Part II
Subroutine iqsort

RECURSIVE SUBROUTINE iqsort(spoint, epoint, array)
  ...
  pivotindex = findpivot(spoint, epoint, array)
  IF (pivotindex /= 0) THEN
    pivot = array(pivotindex)
    k = partition(spoint, epoint, pivot, array)
    CALL iqsort(spoint, k-1, array)
    CALL iqsort(k, epoint, array)
  END IF
  ...

Sometimes we need to use a list structure instead of a simple vector

To create such a linked list we need to define our own list datatype

Such a type can be like this

```fortran
TYPE llist
   TYPE(llist), POINTER :: prev, next, this
   DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures

In addition we can think of writing a procedure for sorting the list in ascending or descending order
Sometimes we need to use a list structure instead of a simple vector.

To create such a linked list we need to define our own list datatype.

Such a type can be like this:

```fortran
TYPE llist
    TYPE(llist), POINTER :: prev, next, this
    DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures:

- Create an element
- Insert an element
- Delete an element
- Traverse the list

In addition we can think of writing a procedure for sorting the list in ascending or descending order.
Sometimes we need to use a list structure instead of a simple vector.

To create such a linked list we need to define our own list datatype.

Such a type can be like this:

```fortran
TYPE llist
    TYPE(llist), POINTER :: prev, next, this
    DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures:

- Create an element
- Insert an element
- Delete an element
- Traverse the list

In addition we can think of writing a procedure for sorting the list in ascending or descending order.
Sometimes we need to use a list structure instead of a simple vector.

To create such a linked list we need to define our own list datatype.

Such a type can be like this:

```fortran
TYPE llist
  TYPE(llist), POINTER :: prev, next, this
  DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures:

- Create an element
- Insert an element
- Delete an element
- Traverse the list

In addition we can think of writing a procedure for sorting the list in ascending or descending order.
Sometimes we need to use a list structure instead of a simple vector.

To create such a linked list we need to define our own list datatype.

Such a type can be like this:

```fortran
TYPE llist
  TYPE(llist), POINTER :: prev, next, this
  DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures:

- Create an element
- Insert an element
- Delete an element
- Traverse the list

In addition we can think of writing a procedure for sorting the list in ascending or descending order.
Sometimes we need to use a list structure instead of a simple vector.

To create such a linked list we need to define our own list datatype.

Such a type can be like this:

```fortran
TYPE llist
    TYPE(llist), POINTER :: prev, next, this
    DOUBLE PRECISION :: v1
END TYPE llist
```

We need to write at least four procedures:

- Create an element
- Insert an element
- Delete an element
- Traverse the list

In addition we can think of writing a procedure for sorting the list in ascending or descending order.
Let us sketch the *create_element* subroutine

We need to create an element of type `llist`

```fortran
SUBROUTINE create_element(element)
    TYPE(llist), POINTER, INTENT(out) :: element
    ALLOCATE(element, STAT=rstat)
    IF(rstat /= 0) THEN
        NULLIFY(element)
    ELSE
        element%v1 = 0.
        NULLIFY(element%prev); NULLIFY(element%next)
    END IF
END SUBROUTINE create_element
```
Let us sketch the `create_element` subroutine

We need to create an element of type `llist`

```fortran
SUBROUTINE create_element(element)
   TYPE(llist), POINTER, INTENT(out) :: element
   ALLOCATE(element,STAT=rstat)
   IF(rstat /= 0) THEN
      NULLIFY(element)
   ELSE
      element%v1 = 0.
      NULLIFY(element%prev); NULLIFY(element%next)
   END IF
END SUBROUTINE create_element
```
We sketch also the *insert* subroutine

```
SUBROUTINE insert(p,n,e)
    TYPE(llist), POINTER :: p, n, e
    e%prev => p; e%next => n
    n%prev => e; p%next => e
```
Either in the main program or in a module it is important to have a set of pointers pointing to the head and tail of the list and also a pointer pointing to the element currently in use.

Such a module can look something like this:

```fortran
MODULE listhandler
  USE listdefs
  TYPE list
    TYPE(llist), POINTER :: head, tail, current
  END TYPE list
  TYPE(list), POINTER :: the_list
CONTAINS
  SUBROUTINE create_list()
    ...
  SUBROUTINE create_element(element)
    ...
  SUBROUTINE insert(p,n,e)
    ...
END MODULE listhandler
```
Either in the main program or in a module it is important to have a set of pointers pointing to the head and tail of the list and also a pointer pointing to the element currently in use.

Such a module can look something like this:

```fortran
MODULE listhandler
  USE listdefs
  TYPE list
    TYPE(llist), POINTER :: head, tail, current
  END TYPE list
  TYPE(list), POINTER :: the_list
CONTAINS
  SUBROUTINE create_list()
    ...
  SUBROUTINE create_element(element)
    ...
  SUBROUTINE insert(p,n,e)
    ...
END MODULE listhandler
```
The subroutine *create_list* will the list pointer and initialize the *head, tail* and *current* to *NULL* pointers.

In addition to these subroutines it can sometimes be useful to sort the list elements in an ascending or descending order.

An adaption of the quicksort algorithm can be used for this purpose.
The subroutine `create_list` will the list pointer and initialize the `head`, `tail` and `current` to `NULL` pointers.

In addition to these subroutines it can sometimes be useful to sort the list elements in an ascending or descending order.

An adaption of the quicksort algorithm can be used for this purpose.
The linked list

- The subroutine `create_list` will the list pointer and initialize the `head`, `tail` and `current` to `NULL` pointers.
- In addition to these subroutines it can sometimes be useful to sort the list elements in an ascending or descending order.
- An adaption of the quicksort algorithm can be used for this purpose.
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
Exercise 13: The quicksort module

- Write the rest of the functionality of the quicksort algorithm and put it in a module.
- Test it with the following data:
  
  ```fortran
  INTEGER, TARGET, IARRAY :: = (/1,4,3,7,9,6,8,9,3,5/)
  ```

  - Check that the program works as intended.
Exercise 13: The quicksort module

- Write the rest of the functionality of the quicksort algorithm and put it in a module
- Test it with the following data:

  INTEGER, TARGET, IARRAY :: = (/1,4,3,7,9,6,8,9,3,5/)

- Check that the program works as intended.
Exercise 13: The quicksort module

- Write the rest of the functionality of the quicksort algorithm and put it in a module.
- Test it with the following data:
  
  ```fortran
  INTEGER, TARGET, IARRAY :: = (/1,4,3,7,9,6,8,9,3,5/)
  ```

- Check that the program works as intended.
Exercise 14: Linked list

- Write the functionality missing from the linked list examples
  - Write a main program that creates a list with up to 10 elements
  - Compile and link it and see that it works with traversing, inserting and deleting elements
Exercise 14: Linked list

- Write the functionality missing from the linked list examples
- Write a main program that creates a list with up to 10 elements
- Compile and link it and see that it works with traversing, inserting and deleting elements
Exercise 14: Linked list

- Write the functionality missing from the linked list examples
- Write a main program that creates a list with up to 10 elements
- Compile and link it and see that it works with traversing, inserting and deleting elements
Studies of bordered layers can often be done in approximately one dimensional models. A wellknown example of this is the Ekman layer.

The solution presented here is originally written by Jens Debernard during his Ph.D. studies at the Department of Geophysics.

We select here an oceanographic approach and looks at an ice floe drifting in an ocean with a speed $\mathbf{V}_{ls} = u\mathbf{i} + v\mathbf{j}$. Sizes in italic are vectors and $\mathbf{i}$ and $\mathbf{j}$ is unit vectors in $x$- and $y$-direction.
A mathematical problem, the Ekman Spiral(1)

- Studies of bordered layers can often be done in approximately one dimensional models. A wellknown example of this is the Ekman layer.

- The solution presented here is originally written by Jens Debernard during his Ph.D. studies at the Department of Geophysics

- We select here an oceanographic approach and looks at an icefloe drifting in an ocean with a speed $V_{is} = u\mathbf{i} + v\mathbf{j}$. Sizes in italic are vectors and $\mathbf{i}$ and $\mathbf{j}$ is unit vectors in $x$- and $y$-direction
A mathematical problem, the Ekman Spiral(1)

- Studies of bordered layers can often be done in approximately one dimensional models. A well-known example of this is the Ekman layer.

- The solution presented here is originally written by Jens Debernard during his Ph.D. studies at the Department of Geophysics.

- We select here an oceanographic approach and looks at an ice floe drifting in an ocean with a speed \( \mathbf{V}_{is} = u\mathbf{i} + v\mathbf{j} \). Sizes in italic are vectors and \( \mathbf{i} \) and \( \mathbf{j} \) is unit vectors in \( x \)- and \( y \)-direction.
The ocean is assumed homogenous and incompressible. We assumes the speed in the ocean are negligible far from the ice, more closely in a certain depth \( z = -H \). The turbulent Reynolds tensions are modelled by a constant Eddy-viscosity coefficient \( K \). This is of course a strong simplification of the physics involved, but it is of small consequence for the principles used here. The advantage is that it makes the whole problem ”cleaner”
A mathematical problem, the Ekman Spiral (3)

- Assumed homogenous horizontal conditions the problem can be written

\[-fv = K \frac{\partial^2 u}{\partial z^2} \]  \hspace{1cm} (3)

\[fu = K \frac{\partial^2 v}{\partial z^2} \]  \hspace{1cm} (4)

with the border conditions \( u = u_{is} \) and \( v = v_{is} \) with \( z = 0 \), also \( u = v = 0 \) with \( z = -H \)
When we are going to solve this system of equations it is an advantage to use complex numbers. Using the complex speed \( W = u + iv \) where \( i \) is the imaginary unit. We can then reform the system of equations to

\[
\frac{\partial^2 W}{\partial z^2} - i \frac{f}{K} W = 0
\]  

(5)

With the border conditions \( W = W_{is} = u_{is} + iv_{is} \) by \( z = 0 \) and \( W = 0 \) with \( z = -H \)
To solve the problem numerically we split the \( z \)-direction into \( N \) discrete points such that \( z_1 = -H \) and \( z_N = 0 \), with grid distance \( \Delta z \). We also let \( W_j \) be an approximation to \( W(z_j) \). The system of equations above can then be approximated by

\[
W_{j+1} - 2W_j + W_{j-1} - i\Delta z^2 \frac{f}{K} W_j = 0, \text{ for } j = 2 \ldots N-1 \tag{6}
\]

with \( W_1 = 0 \) and \( W_N = W_{is} \)
A mathematical problem, the Ekman Spiral (6)

If we let $\delta = \Delta z^2 \frac{f}{K}$, the system of equations can be written as $AW = D$ where

$$A = \begin{bmatrix}
1 & 0 & 0 & \ldots & 0 \\
-1 & 2 + i\delta & -1 & 0 & \ldots & 0 \\
0 & -1 & 2 + i\delta & -1 & 0 & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & -1 & 2 + i\delta & -1 \\
0 & \ldots & \ldots & 0 & 1 \\
\end{bmatrix} \quad (7)$$
and the right side is given with

\[ D = d_j = [0, 0, \ldots, 0, W_is]^T \quad (8) \]

This system of equations is tridiagonal but can, with complex matrix \( A \) and vectors \( D \) and \( W \), be solved with standard very efficient algorithms for tridiagonal systems. We only saves the three diagonals from the matrix \( A \).
A mathematical problem, the Ekman Spiral (8)

\[ a = [0, -1, -1, \ldots, -1, 0] \]  \hspace{1cm} (9)
\[ b = [1, 2 + i\delta, \ldots, 2 + i\delta, 1] \]  \hspace{1cm} (10)
\[ c = [0, -1, \ldots, -1, 0] \]  \hspace{1cm} (11)

Here \( a_n, b_n \) and \( c_n \) now belongs to row number \( n \) in the original matrix
A mathematical problem, the Ekman Spiral (8)

\[ a = [0, -1, -1, \ldots, -1, 0] \] \hspace{1cm} (9)

\[ b = [1, 2 + i \delta, \ldots, 2 + i \delta, 1] \] \hspace{1cm} (10)

\[ c = [0, -1, \ldots, -1, 0] \] \hspace{1cm} (11)

Here \( a_n, b_n \) and \( c_n \) now belongs to row number \( n \) in the original matrix.
A mathematical problem, the Ekman Spiral

The standard algorithm for solving this is first an elimination of all $a$.

$$c_1' = \frac{c_1}{b_1}, \quad d_1' = \frac{d_1}{b_1}$$

(12)

and the further

$$c_n' = \frac{c_n}{b_n - a_n c_{n-1}'}$$

(13)

$$d_n' = \frac{d_n - a_n d_{n-1}'}{b_n - a_n c_{n-1}'}$$

(14)
for $i = 2, \ldots, N$, and then a backwards insertion.

$$W_N = d'_N, \quad \text{and}$$

$$W_n = d'_n - W_{n+1}c'_i$$  \hspace{1cm} (15) \hspace{1cm} (16)

for $n = N - 1, N - 2, \ldots, 1$
A mathematical problem, the Ekman Spiral

To finish all up it is only to extract $u$ and $v$ as

$$u_j = \text{Re}(W_j)$$

$$v_j = \text{Im}(W_j)$$
Exercise 15: Program the Ekman Spiral

In this exercise you shall program the Ekman Spiral from the information given in the equations.

- The program needs some initializing values.
- The speed in X-direction of the ice flow: $uis = 1.0$
- The speed in Y-direction of the ice flow: $vis = 0.0$
- A calculation constant: $p = 1.0E - 3$
- A start value: $x = -30$
- The array length: $n = 101$
Exercise 15: Program the Ekman Spiral

In this exercise you shall program the Ekman Spiral from the information given in the equations:

- The speed in X-direction of the ice flow: \( u_{is} = 1.0 \)
- The speed in Y-direction of the ice flow: \( v_{is} = 0.0 \)
- A calculation constant: \( p = 1.0E - 3 \)
- A start value: \( l = -30 \)
- The array length: \( n = 101 \)
In this exercise you shall program the Ekman Spiral from the information given in the equations:

- The program needs some initializing values:
  - The speed in X-direction of the ice flow: \( u_{is} = 1.0 \)
  - The speed in Y-direction of the ice flow: \( v_{is} = 0.0 \)
  - A calculation constant: \( p = 1.0E - 3 \)
  - A start value: \( \ell = -30 \)
  - The array length: \( n = 101 \)
In this exercise you shall program the Ekman Spiral from the information given in the equations:

- The program needs some initializing values:
  - The speed in X-direction of the ice flow: $uis = 1.0$
  - The speed in Y-direction of the ice flow: $vis = 0.0$
  - A calculation constant: $p = 1.0E - 3$
  - A start value: $l = -30$
  - The array length: $n = 101$
Exercise 15: Program the Ekman Spiral

In this exercise you shall program the Ekman Spiral from the information given in the equations.

- The program needs some initializing values
- The speed in X-direction of the ice flow: \( u_{is} = 1.0 \)
- The speed in Y-direction of the ice flow: \( v_{is} = 0.0 \)
- A calculation constant: \( \rho = 1.0 \times 10^{-3} \)
- A start value: \( l = -30 \)
- The array length: \( n = 101 \)
Exercise 15: Program the Ekman Spiral(1)

- In this exercise you shall program the Ekman Spiral from the information given in the equations
- The program needs some initializing values
- The speed in X-direction of the ice flow: $u_{is} = 1.0$
- The speed in Y-direction of the ice flow: $v_{is} = 0.0$
- A calculation constant: $p = 1.0E^{-3}$
- A start value: $I = -30$
- The array length: $n = 101$
Exercise 15: Program the Ekman Spiral

In this exercise you shall program the Ekman Spiral from the information given in the equations:

- The program needs some initializing values:
  - The speed in X-direction of the ice flow: $uis = 1.0$
  - The speed in Y-direction of the ice flow: $vis = 0.0$
  - A calculation constant: $p = 1.0E - 3$
  - A start value: $l = -30$
  - The array length: $n = 101$
Exercise 15: Program the Ekman spiral (2)

- In addition we need a subroutine to initialize the start values of the complex array $z$

- The linspace subroutine:

```fortran
SUBROUTINE linspace(z, l, k)
   COMPLEX :: z(:)
   INTEGER :: l, k
   INTEGER :: n, i, d
   n = SIZE(z); d = (k-l)/n
   z(1) = REAL(l)
   DO i = 2, n
     z(i) = z(i-1) + d
   END DO
END SUBROUTINE linspace
```
Exercise 15: Program the Ekman spiral(2)

- In addition we need a subroutine to initialize the startvalues of the complex array $z$
- The linspace subroutine:

```fortran
SUBROUTINE linspace(z,l,k)
    COMPLEX :: z(:)
    INTEGER :: l, k
    INTEGER :: n, i, d
    n = SIZE(z);  d = (k-l)/n
    z(1) = REAL(l)
    DO i = 2,n
        z(i) = z(i-1) + d
    END DO
END SUBROUTINE linspace
```
List of Topics

1. Modules
2. A simple module
3. Modules and Operator Overloading
4. Modules and more modules
5. Making programs run faster
6. Exercises part 2
7. More about modules
8. Exercises part 3
9. The promise of Fortran 2003
We shall look at a small example taken from the book Fortran 95/2003 explained by Michael Metcalf, John Reid and Malcolm Cohen

The points module:

```fortran
MODULE points
  TYPE :: point
    REAL :: x, y
  END TYPE point
  INTERFACE
    REAL MODULE FUNCTION point_dist(a,b)
      TYPE(point), INTENT(IN) :: a, b
      END FUNCTION point_dist
  END INTERFACE
END MODULE points
```
We shall look at a small example taken from the book Fortran 95/2003 explained by Michael Metcalf, John Reid and Malcolm Cohen

The points module:

```
MODULE points
    TYPE :: point
        REAL :: x, y
    END TYPE point
INTERFACE
    REAL MODULE FUNCTION point_dist(a,b)
        TYPE(point), INTENT(IN) :: a, b
    END FUNCTION point_dist
END INTERFACE END MODULE points
```
The submodule points_a

```fortran
SUBMODULE (points) points_a
  CONTAINS
    REAL MODULE FUNCTION point_dist(a,b)
    TYPE(point), INTENT(IN) :: a, b
    point_dist = &
      SQRT((a%x-b%x)**2+(a%y-b%y)**2)
    END FUNCTION point_dist
END SUBMODULE points_a
```
Extensions to the type(1)

We shall look at a small example taken from the book Fortran 95/2003 explained by Michael Metcalf, John Reid and Malcolm Cohen.

The matrix type

```
TYPE matrix(real_kind,n,m)
  INTEGER, KIND :: real_kind
  INTEGER, LEN :: n,m
  REAL(real_kind) :: value(n,m)
END TYPE matrix
```
Extensions to the type(1)

- We shall look at a small example taken from the book Fortran 95/2003 explained by Michael Metcalf, John Reid and Malcolm Cohen
- The matrix type

```fortran
  TYPE matrix(real_kind,n,m)
    INTEGER, KIND :: real_kind
    INTEGER, LEN :: n,m
    REAL(real_kind) :: value(n,m)
  END TYPE matrix
```
The labelled matrix

```fortran
TYPE, EXTENDS(matrix) :: labl_matrix(max_lbl_len)
  INTEGER, LEN :: max_lbl_len
  CHARACTER(max_label_length) :: label = ''
END TYPE labelled_matrix
TYPE(labelled_matrix(kind(0.0),10,20,200)) :: x
```