Matlab

Matlab is too slow for demanding applications:

- Statements may be interpreted (not compiled, although there is a Matlab compiler). In Matlab 6.5 (and later) there is a JIT-accelerator (JIT = Just In Time).
- The programmer has poor control over memory.
- It is easy to misuse some language constructs, e.g. dynamic memory allocation.
- Matlab is written in C, Java and Fortran.
- Matlab is not always predictable when it comes to performance.
- The first assignment contains more examples and a case study. You can start working with the Matlab assignment **now**.

Fortran90 and C

The next few pages contain the rudiments of Fortran90 and C and a glance at Fortran77. It is sufficient for the assignments, but you need more for real programming.

I have not tried to show all the different ways a program can be written. Both C and Fortran have several forms of some constructs. Professional code may have many extra details as well.

I have not shown any C++ code (but my example is available in C++-form on the web). C++ is too large and complicated and my labs are not OO. But since C++ means C = C + 1, my C-program is also a C++-program.

Some people use the C-part of C++ together with some convenient C++-constructs (e.g. `//`-comments, reference variables, simplified I/O).

Fortran90 is <u>much</u> nicer than Fortran77, almost a new language. Fortran77 is quite primitive. Fortran77 is still used for HPC.

Millions of lines are available in Fortran77 (some of them will be used in one lab) so it is necessary to understand the basics.

The example code contains one main-program one function and a procedure (void function). The function computes the inner product of two vectors and the procedure sums the elements in an array and returns the sum in a parameter.


```
program main
!
! Comments: everything after !
! Case or blanks (between keywords) are not significant
! (unless they are in strings).
!
! Fortran has an implicit type rule but
! implicit none forces you to declare everything.
!
  implicit none               ! Highly recommended!
  integer           :: k, n, in
  double precision  :: s
  double precision  :: ddot   ! a function

! Arrays start at one by default.
  double precision, dimension(100) :: a, b

  n = 100
  print*, "Type a value for in:"
  read*,  in
  print*, "This is how you write: in = ", in

  do k = 1, n             ! do when k = 1, 2, ..., n
    a(k) = k
    b(k) = -sin(dble(k))  ! using sin
  end do
!
! Call by reference for all variables.
!
  print*, "The inner product is ", ddot(a, b, n)

  call sum_array(a, s, n)    ! NOTE, call
  print*, "The sum of the array is ", s

end program main
```



```
function ddot(x, y, n) result(s)
!
! You give the function a value by assigning
! something to the result variable, s (in this case).
!
  implicit none
  integer :: n
  double precision, dimension(n) :: x, y
  double precision :: s   ! The type of the function

  integer          :: k

  s = 0.0
  do k = 1, n
    s = s + x(k) * y(k)
  end do

end function ddot



subroutine sum_array(a, s, n)
  implicit none
  integer           :: n
  double precision  :: s
  double precision, dimension(n) :: a

  integer :: k

  s = 0.0
  do k = 1, n
    s = s + a(k)
  end do

end subroutine sum_array
```

Some comments. Since Fortran90 has support for array operations the main program could have been shortened:

```
  print*, "The inner product is ", dot_product(a, b)
  print*, "The sum of the array is ", sum(a)
```

`dot_product` and `sum` are built-in.

A long statement can be broken up into several lines. The continued line should end with a **&** .

1 is an integer constant.
`1.0` is a real constant (single precision) and `1.0d0` is a double precision constant in Fortran77.

In Fortran90 it is possible to parameterize the real- and integer types and create more portable code using a module (similar to a simple class) e.g.:

```
module floating_point
! sp = at least 5 significant decimals and
! |exponent range| <= 30 which implies
! IEEE single precision.

 integer, parameter :: sp = selected_real_kind(5, 30)
 integer, parameter :: dp = selected_real_kind(10, 300)
 integer, parameter :: prec = dp ! pick one
end module floating_point

program main
  use floating_point  ! gives access to the module
  real (kind = prec)                :: x, y
  real (kind = prec), dimension(100) :: a, b

  x = 1.24_prec    ! constant
  y = 1.24e-4_prec ! constant
...
```


Here comes the Fortran77-version, but first some comments.

Fortran90 is almost a new language, but in my simple example the differences are not that striking:

- F77 has a column oriented layout dating back to the 80 column punched card.
- No result-statement in functions.
- Different type declarations:

  ```
  double precision  a(n)
  ```

  instead of

  ```
  double precision, dimension(n) :: a
  ```

  although F77-declarations are allowed in F90 as well.
  A Fortran77-program is (essentially) also a Fortran90-program, so it is possible to mix the two styles.

Fortran90 has array operations, pointers, recursion, prototypes, modules, overloading of operators (and more). Fortran77 has none of these.

The example program, coded in F77, is listed on the following two pages. It violates the ANSI-standard in several ways, the most important being the use of do/enddo. Here is the proper way of writing a F77-loop using labels (you will see it in a lab):

```
       do 10 k = 1, n
         s = s + x(k) * y(k)
10     continue
```



```
      program main
*
*     Comments:           c, C or * in column one
*                         text in columns > 72
*     !                   F90-comment
*     First five columns: labels
*     Continuation line:  non-blank in column 6
*     Statements:         columns 7 through 72
*     Case or blanks are not significant
*     (unless they are in strings).
*
*     Arrays start at one by default.
*234567890
      integer           k, n, in
      double precision  a(100), b(100), sum
      double precision  ddot   ! a function

      n = 100
      print*, "Type a value for in:"
      read*,  in
      print*, "This is how you write: in = ", in

      do k = 1, n    ! do when k = 1, 2, ..., n
        a(k) = k
        b(k) = -sin(dble(k))  ! using sin
      end do
*
*     Call by reference for all variables.
*
      print*, "The inner product is ", ddot(a, b, n)

      call sum_array(a, sum, n) ! NOTE, call
      print*, "The sum of the array is ", sum

      end
```



```
      double precision function ddot(x, y, n)
*
*     Fortran has an implicit type rule but
*     implicit none forces you to declare everything.
*     Highly recommended!
*
      implicit none
      integer           n
      double precision  x(n), y(n)

      integer           k
      double precision  sum

      sum = 0.0
      do k = 1, n
        sum = sum + x(k) * y(k)
      end do

      ddot = sum ! give the function its value

      end


      subroutine sum_array(a, sum, n)
      implicit none
      integer           n
      double precision  a(n), sum

      integer           k

      sum = 0.0   ! 0.0 is single and 0.0d0 double
      do k = 1, n
        sum = sum + a(k)
      end do

      end
```

How to compile

The Fortran compilers available on the student system are: **g77** (Fortran77), **gfortran** and **g95** (both Fortran90 and 77).
It would be interesting to use the Intel **ifort**-compiler, but we do not have a license. You can fetch a free copy for Linux (provided you have the disk space, a few hundred Mbyte). See www.

In these handouts I will use **g95** and I will assume that a Fortran90-program has the suffix **.f90**. Some examples:

```
%                  my prompt
% g95 prog.f90     if everything in one prog.f90
                   prog.f would be Fortran77

Produces the executable file a.out
% a.out            run (or ./a.out if no . in the path)

Suppose we have three files main.f90, dot.f90, sum.f90
% g95 main.f90 dot.f90 sum.f90

Can compile the files one by one.
-c means "compile only", do not link.

% g95 -c main.f90   -> object file main.o
% g95 -c dot.f90    -> object file dot.o
% g95 -c sum.f90    -> object file sum.o
% g95 main.o dot.o sum.o  link the object files

% g95 main.o dot.f90 sum.o  works as well, note .f90

Can give many options (or flags) to the compiler, e.g.

% g95 -O3 prog.f90  optimize the code
                    not standard names
```

Next comes the example program in C. See the course page for the same program in C++.


```
/* /usr/include/stdio.h, definitions for IO. */
#include <stdio.h>
#include <math.h>

/* Here are the prototypes for our procedures */
void    sum_array(double[], double *, int);
double  ddot(double[], double[], int);

int main()
{
  int      k, n, in;
  double   a[100], b[100], sum;
  /* Arrays are indexed from 0,  a[0], ..., a[99] */

  n = 100;
  printf("Type a value for in: ");
  scanf("%d", &in);  /* & = the address, the pointer */

  /* %d for int, %f or %e for float. \n is newline */
  printf("This is how you write: in =  %d\n", in);

  for (k = 0; k < n; k++) { /* for k = 0, ..., n-1 */
    a[k] = k + 1;           /* k++ means k = k + 1 */
    b[k] = -sin(k + 1);     /* requires -lm        */
  }

  /* Compute the inner product between a and b. */
  printf("The inner product is %f\n", ddot(a, b, n));

  /* Send the address of sum.
     Arrays are passed by reference automatically. */
  sum_array(a, &sum, n);

  printf("The sum of the array is %f\n", sum);

  return 0; /* return status to unix */
}
```



```
double ddot(double x[], double y[], int n)
{
  int             k;
  double          sum;

  sum = 0.0;
  for (k = 0; k < n; k++)
    sum += x[k] * y[k];  /* short for sum = sum + ... */

  return sum;    /* return the result */
}

void sum_array(double a[], double *sum, int n)
/*
 * void, i.e. not a "mathematical" function.
 * double *sum means that *sum is a double. sum
 * itself is the address, the pointer to sum.
 */
{
  int             k;  /* k is local to sum_array */

  *sum = 0;    /* sum = 0 is WRONG; it will give a
                  Segmentation Fault */

  for (k = 0; k < n; k++)
    *sum += a[k];  /* i.e. *sum = *sum + a[k] */
}
```

Compile by: **cc prog.c -lm** or **cc -O3 prog.c -lm** .
cc is a link to **gcc**.
Separate files are OK (as for Fortran).

The C++-compiler is called **g++**.
One can fetch the Intel C/C++-compiler **icc** as well.

A few words on the unix path

The location of a file or a directory is given by its path.
An absolute path starts at the root in the file tree.
The root is denoted **/** (slash). The path to the **HPC**-directory is **/chalmers/users/thomas/HPC** . The file **ex.f90**, in this directory, has the path **/chalmers/users/thomas/HPC/ex.f90**.
There are also relative paths.

Suppose the current directory is **/chalmers/users/thomas** . A path to the **ex.f90** is then **HPC/ex.f90** . Suppose your current directory is something else, then **~thomas/HPC/ex.f90** is a path to the file. **~**, by itself, denotes your home directory, **~user**, is the path to the home directory of **user**.
So I could have written, **~/HPC/ex.f90** .
.. is the level above, and **.** is the current directory. That is why we sometimes write **./a.out**, se below.

The shell (**csh**, **tcsh**, **sh**, **ksh**, **bash**, ...) keeps several variables. One important such variable is the **path**. I will concentrate on **[t]csh**. The path contains a blank-separated list of directories. When you type a command (which is not built-in, such as **cd**) the shell will search for a directory containing the command (an executable file with the given name). If the shell finds the command it will execute it, if not, it will complain:

```
% set path = ( )          no directories
% cd HPC                  cd is built-in
% ls
ls: Command not found.
% /bin/ls                 works
A.mat    ... etc

% set path = ( /bin )
% ls                      now tcsh finds ls
A.mat    ... etc
```

The **set** is local to the particular shell and lasts only the present login session.

Sometimes there are several different versions of a command. The shell will execute the command it finds first (from left to right).

```
% which ls
/bin/ls

% which gfortran
/usr/bin/gfortran        comes with the system

% which gfortran         used in the course 2006
/chalmers/users/thomas/HPC/gfortran/bin/gfortran
```

In the first **which**, **/usr/bin** comes before the HPC-directory, and in the second **/usr/bin** comes after.
If you do not have **.** in your **path**, the shell will not look for executables in the current directory.

```
% pwd                          print current directory
/chalmers/users/thomas/HPC/Test

% a.out
a.out: Command not found.      no . in the path
% ./a.out                      works
% set path = ( $path . )       add . to the path
% a.out                        works
```

$path is the value of **path**. Suppose the **path** contains **~** .

```
% cp a.out ~/a.out1
% which a.out1
a.out1: Command not found.

% rehash            rebuild the internal hash table
% which a.out1
/chalmers/users/thomas/a.out1
```


A command does not have to be a compiled program.

```
% ls -l /bin/ls
-rwxr-xr-x  1 root root 82796 Jun 20 13:52 /bin/ls

% file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.2.5,
dynamically linked (uses shared libs), stripped

% which cd
cd: shell built-in command.

% which apropos
/usr/bin/apropos
% file /usr/bin/apropos
/usr/bin/apropos: Bourne shell script text executable
% head -3 /usr/bin/apropos
#!/bin/sh
#
# apropos -- search the whatis database for keywords.
```

A user would usually (perhaps not if one is a student; see below for more details) set the **path**-variable in the startup file **.tcshrc** which usually resides in the login directory. The period in the name makes the file invisible. Type **ls -a** to see the names of all the dot-files.

To see your path, type **echo $path**, or give the command **set**, which prints all the shell variables.

Shell-variables are not exported to sub-processes so the shell creates an environment variable, **PATH**, as well. **PATH** is exported to sub-processes and it contains a **:**-separated list of directories).

```
% set var = hello
% echo $var        like print
hello
% tcsh             start a sub-shell
% echo $var
var: Undefined variable.
% exit

% setenv var hello    an environment variable, no =
% tcsh                sub-shell
% echo $var
hello
```

To see all your environment variables, type **setenv**. Another useful environment variable is the manual search path, **MANPATH** and the **LD_LIBRARY_PATH** (much more details later on).

A note on the student environment. To make it easier for beginners (both teachers and students) Medic has set up a standard environment where you do not have to create your own startup files. One does not have to use it (I don't), but if you do this is how you can modify your environment. Edit (create, the first time) the file **.vcs4/tcshrc** on your login-level. Here is sample file:

```
setenv LD_LIBRARY_PATH .
alias m less
set prompt = '> '
set path = ( $path . )
```

It sets an environment variable. An alias is created so that one type **m** instead of **less** (**less** is a so called pager, very useful). I do not like standard prompt so I have set it to **>** . The last row appends **.** to the path. Be careful with the last one!

In order to test that the changes work one can logout and then login. This is a bit slow, and if one has made a mistake in tcshrc it may be impossible to login. One should then be able to do a failsafe login and correct the mistake.

Less time consuming, and safer, is to open a new terminal window (xterm or gnome). Each time a new terminal is opened a shell is created as a subprocess under the terminal and the shell reads the startup file. One can see if the changes work in the new window.

Note that changes in tcshrc do not affect the already existing shells (windows). If one does not use the standard environment it is easy to update the existing shells as well.

More on * and & in C

In **main** we have reserved storage for the variable **sum**. The variable has an address in memory. Let us add the following two statements after the declaration of **sum**:

```
double *p;        /* p is a pointer to double */
p = &sum;         /* p = address of sum       */
```

& gives the address of a variable, so the second line assigns the address of **sum** to the pointer variable **p**, **p** points to **sum**.

Given **p** we can get the value of **sum** by using the indirection (or dereferencing) operator, *****. Thus ***p** is the value of **sum**.

This explains the first line, the declaration of **p**. It says that ***p** is a double, so that **p** must be a pointer to double.

We can now understand how the parameters are passed in the call, **sum_array(a, &sum, n);**. The address of **sum**, **&sum**, is passed as a parameter to the function.
Inside the function it is possible to access (read and write) the memory where **sum** is residing, since the function has been given the address.

Inside the function **sum** equals the address (it is equivalent to the pointer **p** above). Since we have passed the address (and not the value) we must use ***** to access the value itself (and not the address).

The variable **n** is copied to the function. If we change **n** inside the function the original **n** (in **main**) will not be changed.

The compiler will pass the address of the first element, **&a[0]**, when we write the name of the array. So the function can access all the elements in the array.

If-statements and logical expressions

```
real      a, b, c, d
logical   q

if( a < b .and. c == d .or. .not. q ) then
 ...  zero or more statements
else
 ...  zero or more statements
end if

float a, b, c, d, q;

if( a < b && c == d || !q ) {
 ...  zero or more statements
} else {
 ...  zero or more statements
}
```

Operation	Fortran77	C	Fortran90
$<$	`.lt.`	`<`	`<`
$\leq$	`.le.`	`<=`	`<=`
$=$	`.eq.`	`==`	`==`
$\neq$	`.ne.`	`!=`	`/=`
$\geq$	`.ge.`	`>=`	`>=`
$>$	`.gt.`	`>`	`>`
and	`.and.`	`&&`	`.and.`
or	`.or.`	`||`	`.or.`
not	`.not.`	`!`	`.not.`
true	`.true.`	$\neq$ `0`	`.true.`
false	`.false.`	`0`	`.false.`

Note: **if (! q == 1.25)** $\Leftrightarrow$ **if ((!q) == 1.25)**, not **if(! (q == 1.25))**.
Look at the predence table at the end of this handout.

Include (header) files

The **cc**-command first runs the C preprocessor, **cpp**. **cpp** looks for lines starting with # followed by a directive (there are several). From the man-page for **cpp**:

```
#include "filename"
#include <filename>
```

Read in the contents of filename at this location. This data is processed by cpp as if it were part of the current file. When the **<filename>** notation is used, filename is only searched for in the standard "include" directories. Can tell **cpp** where to look for files by using the **-I**-option.

A typical header file contains named constants, macros (somewhat like functions) and function protypes, e.g:

```
#define M_PI   3.14159265358979323846  /* pi */
#define __ARGS(a)       a
extern int MPI_Send __ARGS((void *, int, MPI_Datatype, in
```

It is common to store several versions of a program in one file and to use **cpp** to extract a special version for one system.

In **_ompc_init** from Omni, a Japanese implementation of OpenMP:

```
...
#ifdef OMNI_OS_SOLARIS
    lnp = sysconf(_SC_NPROCESSORS_ONLN);
#else
#ifdef OMNI_OS_IRIX
    lnp = sysconf(_SC_NPROC_ONLN);
#else
#ifdef OMNI_OS_LINUX
       ...  deleted code
```

Under Linux we would compile by:

```
cc -DOMNI_OS_LINUX ...
```


Using the man-command

```
% man sin
SIN(3) Linux Programmer's Manual SIN(3)


NAME
   sin, sinf, sinl - sine function

SYNOPSIS
   #include <math.h>

   double sin(double x);

   float sinf(float x);

   long double sinl(long double x);

DESCRIPTION
   The sin() function returns the sine of x,
   where x is given in radians.

RETURN VALUE
   The sin() function returns a value between -1 and 1.

CONFORMING TO
   SVID 3, POSIX, BSD 4.3, ISO 9899. The float and
   the long double variants are C99 requirements.

SEE ALSO
   acos(3), asin(3), atan(3), atan2(3), cos(3), tan(3)
```

You will not find man-pages for everything.
Can try to make a keyword search: **man -k keyword**.

Some useful C-tools

indent and **cb** (Sun) are pretty printers for C.
indent file.c saves the old file in **file.c~**. If the C-program contains syntax errors **indent** can "destroy" it.
I use **indent -i2 -kr -nut file.c**.

lint is a C program checker (not Linux). From the manual:

lint detects features of C program files which are likely to be bugs, non-portable, or wasteful.

lint also checks type usage more strictly than the compiler. lint issues error and warning messages. Among the things it detects are:

- Unreachable statements
- Loops not entered at the top
- Automatic variables declared and not used
- Logical expressions whose value is constant.

lint checks for functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Using **lint** on our program gives:

```
% lint prog.c -lm

function returns value which is always ignored
    printf              scanf

declared global, could be static
    sum_array           lint.c(37)
    ddot                lint.c(58)
```

Add **static**, **static void sum_array** makes **sum_array** local to the file (**prog.c**) it is defined in.

lint is not available on Linux systems. Here are two alternatives. Ask the compiler to be more careful:

```
gcc -Wall  programs ...
```

(there are several other **W**-options). No complaints on my example. Much more details can be obtained from **splint** (**www.splint.org**).

You may want to start with **splint -weak**

```
% splint -weak ex.c
Splint 3.1.1 --- 15 Jun 2004

ex.c: (in function main)
ex.c:23:5: Assignment of int to double: a[k] = k + 1
  To allow all numeric types to match, use +relaxtypes.
ex.c:24:17: Function sin expects arg 1 to be double
           gets int: k + 1
Finished checking --- 2 code warnings
```

These are warnings and not errors, but they are still worth to check. By using a type cast we can make **splint** quiet.

```
a[k] = (double) k + 1;  // or (double) (k + 1)
b[k] = -sin((double) k + 1);
```

Since we have a prototype for **sin** (from **math.h**) there was an automatic cast to double in the first version of the code. Just typing **splint ex.c** will lead to more complaints, e.g. that the return status from **scanf** is ignored. They can be fixed by taking care of the status, or explicitly ignoring it:

```
(void) scanf("%d", &in);
```

Having made the functions local to the file (**static**) there remains a few comments, but they cannot be fixed unless one adds so-called annotations (extra information in the form of special comments) to the code. Another alternative is to switch of a groups of warnings. **splint -strict ex.c** gives even more warnings.

A common Fortran construction

Fortran77 does not have dynamic memory allocation (like Fortran90 and C). If you need an **m** by **n** matrix **A** you would usually reserve space for the largest matrix you may need (for a particular application). If you pass the matrix as an argument to a procedure the procedure must be told about the extent of the first dimension (the number of rows) in order to be able to compute the address of an element. If the maximum number of rows is **max_m** the address, **adr( )**, of **A(j, k)** is given by

```
adr(A(j, k)) = adr(A(1, 1)) + max_m*(k - 1) + j - 1
```

So, a matrix is stored by columns in Fortran. In C it is stored by rows (so the compiler must know the number of columns in the matrix). Since you can allocate the precise number of elements in C this is less of an issue.

A program may look like this:

```
  integer            :: m, n
  integer, parameter :: max_m = 1000, max_n = 50
  double precision, dimension(max_m, max_n) :: A

  call sub ( A, max_m, m, n )  ! m, n actual sizes
end
subroutine sub ( A, max_m, m, n )
  integer :: max_m, m, n
  double precision, dimension(max_m, *) ::  A
  ... ! can have 1 instead of *
```

Better (but not necessary) is:

```
  call sub ( A, max_m, max_n, m, n )
  ...
  subroutine sub ( A, max_m, max_n, m, n )
  integer :: max_m, m, n
  double precision, dimension(max_m, max_n) :: A
```

since index checks can be made by the compiler.

Part of the manual page for the Lapack routine **dgesv**:

```
NAME
 dgesv - compute the solution to a real system
         of linear equations A * X = B,

SYNOPSIS
 SUBROUTINE DGESV(N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)

 INTEGER N, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
 ...
ARGUMENTS
 N (input) The number of linear equations, i.e., the
           order of the matrix A. N >= 0.

 NRHS (input)
           The number of right hand sides, i.e.,  the
           number of columns of the matrix B. NRHS >= 0.

 A (input/output)
           On entry, the N-by-N coefficient matrix A.
           On exit, the factors L and U from the
           factorization A = P*L*U; the unit diagonal
           elements of L are not stored.

 LDA (input)
           The leading dimension of the array A.
           LDA >= max(1,N).
...
```

It is possible to construct a nicer interface in Fortran90 (C++). Essentially **subroutine gesv(A, B, ipiv, info)** where **gesv** is polymorphic, (for the four types **S**, **D C**, **Z**) and where the size information is included in the matrices.

Array operations for Fortran90

You must be able to understand this kind of code in one assignment, so here are some examples. One can do more.

```
program array_example
  implicit none

  ! works for other types as well
  integer                     :: k
  integer, dimension(-4:3)    :: a     ! Note -4
  integer, dimension(8)       :: b, c  ! Default 1:8
  integer, dimension(-2:3, 3) :: M


  a = 1 ! set all elements to 1
  b = (/ 1, 2, 3, 4, 5, 6, 7, 8 /) ! constant array
  b = 10 * b

  c(1:3) = b(6:8)
  print*, 'size(a), size(c)     = ', size(a),   size(c)
  print*, 'lbound(a), ubound(c) = ',lbound(a),ubound(a)
  print*, 'lbound(c), ubound(c) = ',lbound(c),ubound(c)

  c(4:8) = b(8:4:-1) ! almost like Matlab
  print*, 'c = ', c  ! can print a whole array

  print*, 'minval(c) = ', minval(c) ! a builtin func.
  a = a + b * c                     ! elementwise *
  print*, 'a = ', a
  print*, 'sum(a) = ', sum(a)       ! another builtin
```



```
  M = 0
  M(1, :) = b(1:3) ! Row with index one
  print*, 'M(1, :) = ', M(1, :)

  M(:, 1) = 20     ! The first column
  where ( M == 0 ) ! instead of two loops
    M = -1
  end where

  print*, 'lbound(M) = ', lbound(M) ! an array

  do k = lbound(M, 1), ubound(M, 1) ! print M
    print '(a, i2, a, i2, 2i5)', ' M(', k, ', :) = ', &
           M(k, :)
  end do
end

% ./a.out
 size(a), size(c)     =  8 8
 lbound(a), ubound(c) =  -4 3
 lbound(c), ubound(c) =  1 8
 c =  60 70 80 80 70 60 50 40
 minval(c) =  40
 a =  601 1401 2401 3201 3501 3601 3501 3201
 sum(a) =  21408
 M(1, :) =  10 20 30
 lbound(M) =  -2 1
 M(-2, :) = 20   -1   -1
 M(-1, :) = 20   -1   -1
 M( 0, :) = 20   -1   -1
 M( 1, :) = 20   20   30
 M( 2, :) = 20   -1   -1
 M( 3, :) = 20   -1   -1
```


Some dangerous things

```
  for(k = 0; k < n; k++)
    a[k] = b[k] + c[k];
    e[k] = f[k] * g[k];
```

is <u>not</u> the same as

```
  for(k = 0; k < n; k++) {
    a[k] = b[k] + c[k];
    e[k] = f[k] * g[k];
  }
```

Similarly for if-statements.

```
  if ( j = 0 ) printf("j is equal to zero\n");

  /* while a valid char... */
  while ((c = getchar()) != EOF )  { ...

  k == 3;
```

Actual and formal parameters lists: check position, number and type. Can use interface blocks ("prototypes").

```
program main
  double precision :: a, b

  a = 0.0
  call sub(a, 1.0, b)
  print*, a, b
end
subroutine sub(i, j)
  integer :: i, j

  i = i + 1
  j = 10.0
end
```



```
% a.out
Segmentation fault  % result depends on the compiler
```

Remove the line `j = 10.0` and run again:

```
% a.out
     2.1219957909653-314  0.  % depends on the compiler
```

C- and Fortran compilers do not usually check array bounds.

```
void sub(double a[]);

#include <stdio.h>
main()
{
        double          b[1], a[10];

        b[0] = 1;
        sub(a);
        printf("%f\n", b[0]);
}

void
sub(double a[])
{
        a[11] = 12345.0;
}
```

Running this program we get:

```
% a.out
12345.000000
```

Changing `a[10]` to `a[1000000]` gives Segmentation fault.

Some Fortran-compilers can check subscripts (provided you do not lie):

```
program main
  double precision, dimension(10) :: a

  call lie(a)
  print*, 'a(1) = ', a(1)

end program main
subroutine lie(a)
  double precision, dimension(10) :: a

  do j = 1, 100  !!! NOTE
    a(j) = j
  end do

end subroutine lie

% ifort -CB lie.f90
% ./a.out
forrtl: severe (408): fort: (2): Subscript #1 of the
array A has value 11 which is greater than the upper
 bound of 10


Change dimension(10)   to
       dimension(100)  in lie

% ifort -CB lie.f90
% a.out
 a(1) =     1.0000000000000
```


Precedence and associativity of C-operators

Operators have been grouped in order of decreasing precedence, where operators between horizontal lines have the same precedence.

Operator	Meaning	Associativity
`( )`	function call	→
`[ ]`	vector index	
`->`	structure pointer	
`.`	structure member	
`++`	postfix increment	
`-`	postfix decrement	
`!`	logical negation	←
`~`	bitwise negation	
`++`	prefix increment	
`--`	prefix decrement	
`+`	unary addition	
`-`	unary subtraction	
`*`	indirection	
`&`	address	
`(type)`	type cast	
`sizeof`	number of bytes	
`*`	multiplication	→
`/`	division	
`%`	modulus	
`+`	binary addition	→
`-`	binary subtraction	
«	left shift	→
»	right shift	
`<`	less than	→
`<=`	less or equal	
`>`	greater than	
`>=`	greater or equal	
`==`	equality	→
`!=`	inequality	

Operator	Meaning	Associativity
`&`	bitwise and	→
`^`	bitwise xor	→
`|`	bitwise or	→
`&&`	logical and	→
`||`	logical or	→
`?:`	conditional expression	←
`=`	assignment	←
`+=`	combined assignment and addition	
`-=`	combined assignment and subtraction	
`*=`	combined assignment and multiplication	
`/=`	combined assignment and division	
`%=`	combined assignment and modulus	
`&=`	combined assignment and bitwise and	
`^=`	combined assignment and bitwise xor	
`|=`	combined assignment and bitwise or	
«=	combined assignment and left shift	
»=	combined assignment and right shift	
`,`	comma	→

Here are a few comments, see a textbook or my links for a complete description.

- Left to right associativity (→) means that `a-b-c` is evaluated as `(a-b)-c` and not `a-(b-c)`. `a = b = c`, on the other hand, is evaluated as `a = (b = c)`. Note that the assignment `b = c` returns the value of `c`.

  `if ( a < b < c ) ...;` means `if ( (a < b) < c ) ...;` where `a < b` is 1 (true) if `a < b` and 0 (false) otherwise. This number is then compared to c. The statement does <u>not</u> determine "if b is between a and c".

- `a++;` is short for `a = a + 1;`, so is `++a;`. Both `a++` and `++a` can be used in expressions, e.g. `b = a++;`, `c = ++a;`. The value of `a++;` is `a`'s value before it has been incremented and the value of `++a;` is the new value.
- `a += 3;` is short for `a = a + 3;`.
- As in many languages, integer division is exact (through truncation), so `4 / 3` becomes 1. Similarly, `i = 1.25;`, will drop the decimals if `i` is an integer variable.
- `expr1 ? expr2 : expr3` equals `expr2` if `expr1` is true, and equals `expr3`, otherwise.
- `(type)` is used for type conversions, e.g. `(double) 3` becomes `3.0` and `(int) 3.25` is truncated to `3`.
- `sizeof(type_name)` or `sizeof expression` gives the size in bytes necessary to store the quantity. So, `sizeof(double)` is 8 on the Sun system and `sizeof (1 + 2)` is 4 (four bytes for an integer).
- When two or more expressions are separated by the comma operator, they evaluate from left to right. The result has the type and value of the rightmost expression. In the following example, the value 1 is assigned to `a`, and the value 2 is assigned to `b`. `a = b = 1, b += 2, b -= 1;`
- Do not write too tricky expressions. It is easy to make mistakes or to end up with undefined statements.
  `a[i++] = i;` and `i = ++i + 1;` are both undefined.
  See the standard, section 6.5, if you are interested in why.

Precedence of Fortran 90-operators

Operators between horizontal lines have the same precedence.

Operator	Meaning
unary user-defined operator	
`**`	power
`*` `/`	multiplication division
`+` `-`	unary addition unary subtraction
`+` `-`	binary addition binary subtraction
`//`	string concatenation

`==` `.EQ.`
`/=` `.NE.`
`<` `.LT.`
`<=` `.LE.`
`>` `.GT.`
`>=` `.GE.`
	equality
	inequality
	less than
	less or equal
	greater than
	greater or equal

`.NOT.`	logical negation
`.AND.`	logical and
`.OR.`	logical or
`.EQV.` `.NEQV.`	logical equivalence logical non-equivalence
binary user-defined operator	

Comments:
`==` is the Fortran90 form and **`.EQ.`** is the Fortran77 form, etc.
In Fortran90 lower case is permitted, .e.g **`.not.`** .

About the user defined operators. In Fortran90 it is possible to define ones own operators by overloading existing operators or by creating one with the name **`.name.`** where **`name`** consists of at most 31 letters.

C can be very hard to read. An example from the 3rd International Obfuscated C Code Contest (this program is by one of the winners, Lennart Augustsson).
See **`http://www.ioccc.org/`** for more examples.

```
typedef struct n{int a:3,
b:29;struct n*c;}t;t*
f();r(){}m(u)t*u;{t*w,*z;
z=u->c,q(z),u->b=z->b*10,
w=u->c=f(),w->a=1,w->c=z->
c;}t*k;g(u)t*u;{t*z,*v,*p,
*x;z=u->c,q(z),u->b=z->b,v
=z->c,z->a=2,x=z->c=f(),x
->a=3,x->b=2,p=x->c=f(),p
->c=f(),p->c->a=1,p->c->c=
v;}int i;h(u)t*u;{t*z,*v,*
w;int c,e;z=u->c,v=z->c,q(
v),c=u->b,e=v->b,u->b=z->b
,z->a=3,z->b=c+1,e+9>=c&&(
q(z),e=z->b,u->b+=e/c,w=f(
),w->b=e%c,w->c=z->c,u->c=
w);}int(*y[4])()={r,m,g,h};
char *sbrk();main(){t*e,*p,*o;
o=f(),o->c=o,o->b=1,e=f(),
e->a=2,p=e->c=f(),p->b=2,
p->c=o,q(e),e=e->c,(void)write
(1,"2.",2);for(;;e=e->c){q(e),
e->b=write(1,&e->b["0123456789"],
1);}}t*f(){return i||(i=1000,
k=(t*)sbrk(i*sizeof(t))),k+--i;
}q(p)t*p;{(*y[p->a])(p);}
```

```
% a.out
2.718281828459045235360287471352662497757247093699
5957496696762772407663035354759457138217852516642742
74663919320030599218174135966290435729003342952605
956307381323286279434907632338298807531952510190115^C
```


Using make

Make keeps track of modification dates and recompiles the routines that have changed.

Suppose we have the programs **`main.f90`** and **`sub.f90`** and that the executable should be called **`run`**. Here is a simple makefile (it should be called **`Makefile`** or **`makefile`**):

```
run: main.o sub.o
     g95 -o run main.o sub.o

main.o: main.f90
     g95 -c main.f90

sub.o: sub.f90
     g95 -c sub.f90
```

A typical line looks like:

```
target: files that the target depends on
^Ia rule telling make how to produce the target
```

Note the tab character. Make makes the first target in the makefile. **`-c`** means compile only (do not link) and **`-o`** gives the name of the executable.

To use the makefile just give the command **`make`**.

```
% make
g95 -c main.f90
g95 -c sub.f90
g95 -o run main.o sub.o
```

To run the program we would type **`run`** .

If we type **`make`** again nothing happens (no file has changed):

```
% make
'run' is up to date.
```

Now we edit **`sub.f90`** and type **`make`** again:

```
% make
g95 -c sub.f90
g95 -o run main.o sub.o
```

Note that only **`sub.f90`** is compiled. The last step is to link **`main.o`** and **`sub.o`** together (**`g95`** calls the linker, **`ld`**).

Writing makefiles this way is somewhat inconvenient if we have many files. **`make`** may have some builtin rules, specifying how to get from source files to object files, at least for C. The following makefile would then be sufficient:

```
run: main.o sub.o
     g95 -o run main.o sub.o
```

Fortran90 is unknown to some make-implementations and on the student system one gets:

```
% make
make: *** No rule to make target 'main.o',
          needed by 'run'.  Stop.
```

We can fix that by adding a special rule for how to produce an object file from a Fortran90 source file.

```
run: main.o sub.o
     g95 -o run main.o sub.o

.SUFFIXES: .f90
.f90.o:
        g95 -c $<
```