A short introduction to C, Fortran 90, FORTRAN 77, tcsh and bash

Thomas Ericsson
Computational Mathematics
Chalmers/GU

2013

Contents

1 A short introduction to C **4**

2 Hello World! **4**

3 Functions, a first example **6**

4 Separate compilation and `ld` **9**

5 More on prototypes and type conversion **9**

6 void functions, passing parameters **11**

7 Arrays **14**
7.1 Two-dimensional arrays 16

8 A matter of style **17**

9 If-statements and logical expressions **18**

10 Some useful C-tools **19**

11 A few words about the C99 standard **19**

12 More on `cpp` **21**

13 Using the `man`-command **22**

14 More on matrices **23**
14.1 Dynamic memory allocation 24
14.2 This will not work with Fortran 26
14.3 This will work with Fortran 27
14.4 C and large arrays 28

15 A note on structs and `const` **30**
15.1 Structures 30
15.2 `const` 31
15.3 `const` and structures 31

16 Assert, a debugging tool **32**

17 Precedence and associativity of C-operators **33**

18 A short introduction to Fortran **35**

19 A simple example **36**

20 Numerical constants **38**
20.1 Warning: do not mix single and double precision 38

21 The simple example in FORTRAN 77 **39**

22 How to compile **42**

23 If-statements and logical expressions **42**

24 A small Fortran 90-example **43**

25 A common Fortran construction **44**

26 Dynamic memory allocation in Fortran 90 **47**

27 Some dangerous things **47**

28 Precedence of Fortran 90-operators **50**

29 A few words about `tcsh` and `bash` **51**

29.1 The path 51
29.2 Now something about `bash` 53
29.3 A note on the student environment 53

1 A short introduction to C

C is a widely used programming language, especially in Unix applications. The language was developed in 1972 by Dennis Ritchie at Bell Labs for use with the Unix operating system. I learnt C reading the classic book "The C Programming Language" by Brian Kernighan and Dennis Ritchie. The book was published 1978. C is a fairly small language, the book is only 228 pages. I have several C++ books, all containing more than 1000 pages each. Since C was used to develop the Unix system, it has support for low level operations, such as finding out the address of a variable. It is also a very concise language, having abbreviations for common operations.
`k = k + 1` and `s = s + term` can be written `k++` and `s += term`, for example.

This is convenient if you are an experienced C-programmer, but it may cause problems for the novice.
Here is another C-feature. In C an assignment such as `k = 2 * j - m;` has a value, which is the value of `k`, the leftmost variable. Matlab follows C when it comes the logical values in if-statements, zero is false and non-zero is true. This means that the following C-statement is correct

```
if ( k = 2 * j - m ) {
  do something
}
```

It computes the value of `k` and checks if it is non-zero. If we had intended to do something when `k` equals `2 * j - m` we should have written

```
if ( k == 2 * j - m ) {
  do something
}
```

Another, more severe, problem is that there is no index control for array indices, like there is in Matlab. One tends to use pointers (addresses) frequently as well and there is little control of these. So, in short, one should be very careful when writing C-programs, or there is a large risk that one has to spend long hours debugging.

In 1989 C became an ANSI standard, often referred to C89, and the year after came the ISO-standard, C90 (although C89 and C90 describe the same language). In 1999 came a new standard, C99.

For more history and background see the Wikipedia article:
`http://en.wikipedia.org/wiki/C_(programming_language)` .
There is also a page about the book:
`http://en.wikipedia.org/wiki/The_C_Programming_Language_(book)` .

The following introduction 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. C has several forms of some constructs. Professional code has many extra details as well.

C can be very hard to read and there even was the "International Obfuscated C Code Contest".
See `http://en.wikipedia.org/wiki/IOCCC` for unreadable and amusing programs.

2 Hello World!

We start with the compulsory Hello World!-program. I wrote the program using an editor and saved it in the file `hello.c`. If you do not have a favourite editor like `vim`, `gvim`, `emacs` etc, I recommend using `nedit`, the Nirvana editor. It is quite capable and easy to use, although it does not have support for the utf-8 character set (used on the lab computers). This is mostly a problem if you write in Swedish (using å-ö). In the printout below, I listed the program in a terminal window using the `cat`-command (you do not have to do this every time, of course :-)

```
% cat hello.c
#include <stdio.h>
int main()
{
  printf("Hello World!\n");
  return 0;
}

% gcc hello.c

% a.out
Hello World!

% ./a.out
Hello World!
```

`%` is the prompt. I compiled the program using, `gcc`, the GNU C-compiler. The executable ("machine code") was stored in the file named `a.out` (you can store it in another file if you like). Finally I executed the program by typing the name of the executable. If you do not have `.` in your Unix-path you would type `./a.out` instead. The dot means the current working directory, so `./a.out` means the `a.out` in the directory where I am at the moment.

Let us look at the code. The first line, the one starting with a `#` is read by the C preprocessor, `cpp`. It will read the file, `/usr/include/stdio.h`, and place it in the program. This file, a so-called *include file* or *header file*, typically contains named constants, macros (somewhat like functions) and function prototypes. Named constants are used so we do not have to write numbers to choose a particular option, instead we can write a name.

The main program, must be called `main`, is an integer (`int`) function. It can take parameters, but we ignore them in this example (the `( )`) and it returns status information to the Unix-system (to the shell, `bash` ot `tcsh`), using the `return`-statement, zero usually means OK. We can print the status in the shell (`echo $status` in `tcsh`, `echo $?` in `bash`). One could also use the status in if-statements in the shell. If you like, you can return `EXIT_SUCCESS` (or `EXIT_FAILURE`) instead of zero and one, provided you include the header file `stdlib.h`.

The input parameters are used to pass arguments from the shell to the program. When giving the `ls` command with the long flag, `ls -l`, the ls-command (a compiled C-program) can access the flag `-l`.
`printf` is a print statement, and `\n` means newline. Semicolon, `;`, ends a statement, so it is not like in Matlab where an end of line suffices. If we forget the semicolon after the `printf` statements, we get a syntax error and the compiler complains:

```
% gcc hello.c
hello.c: In function ‘main’:
hello.c:5: error: syntax error before "return"
```

The braces, `{ }`, are used to delimit the body of the function.

To find out more about what flags (options) `gcc` can take, we type `man gcc` in a terminal window. The following command

```
% gcc -o hello -O hello.c
```

optimizes the code for speed (overkill for this tiny example) and places the executable in `hello` instead of in `a.out`. To execute the program we type `hello` or `./hello` .

3 Functions, a first example

Now to a more complicated example, where we use a very primitive method (the trapezoidal method) to approximate

$$\int_a^b e^{-x^2}\,dx, \quad a < b$$

The interval, (a, b), is divided into n intervals and on each interval the integral is approximated by the area of a trapezoid, and the formula is:

$$\int_a^b f(x)\,dx \approx h\left[\frac{f(a)}{2} + f(a+h) + f(a+2h) + \cdots + f(b-h) + \frac{f(b)}{2}\right], \quad \text{where} \quad h = \frac{b-a}{n}$$

There are much better methods and one could write a code that accepts more general integrands, but this is, after all, not a course in numerical analysis.

Since the program would become too messy if I added all the comments to the code, I have numbered the lines and added comments afterwards. **Note** that the line numbers are **not** part of the code.

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  double trapeze(double, double, int);
 5
 6  int main()
 7  {
 8    printf("The integral is approximately = %e\n", trapeze(0, 1, 100));
 9
10    return 0;
11  }
12
13  double trapeze(double a, double b, int n)
14  {
15    /* A primitive quadrature method for approximating
16       the integral of exp(-x^2) from a to b.
17       n is the number of sub intervals.
18     */
19
20    int k;
21    double x, h, sum = 0.0;
22
23    if (n <= 0) {
24      printf("*** n must be at least 1.\n");
25      return -1;
26    }
27
28    h = (b - a) / n;
29    x = a;
30    sum = 0.5 * exp(-x * x);
31    for (k = 1; k < n; k = k + 1) {
32      x = x + h;
33      sum = sum + exp(-x * x);
34    }
35    sum = sum + 0.5 * exp(-b * b);
36    sum = sum * h;
37
38    return sum;
39  }
```

The example code contains a main-program and a function. On line 2, we include `math.h` since the program uses the exponential function, `exp`, and we need the *prototype* for the function. A prototype gives the name of the function and the types of input and output parameters. Since `exp` takes a double precision argument and returns a double precision value the prototype is:

```
double exp(double);
```

`double` is the name of the double precision (8 bytes) floating point type. The reason we use prototypes is to supply the compiler with more information, so it can warn us if we call a function with the wrong number or types of the parameters. The compiler would also use the information to make type conversions of parameters (more below).

Our own function, `trapeze`, takes three input arguments, the interval endpoints a and b, and a number, n, of intervals, and returns the approximation of the integral. On line 4 I have supplied a prototype for the function. One can, but does not have to, supply the variable names as well.

On line 8 I print some text and call the function. `printf` is a function that can take a different number of arguments. In this case the first is a string, and the second the value returned from `trapeze`. `%e` is a format code, which tells `printf` that the integral value should be written using an engineering format (decimals and exponential part). To see the other format codes, we use the manual command in Unix.
Type `man -s3 printf` in a terminal window (note that `man printf` gives you another manual page).

Lines 13-39 show the `trapeze` function. Note that the first line looks like the prototype, but now *with* variable names. Comments are written between `/* */`, but many compilers allow for C++-comments as well (lines starting with `//`), this came with the C99-standard.
Lines 20, 21 are type declarations of so-called automatic variables. These variables are local to the function. Space is allocated when the function is entered and the memory is deallocated when we return from the function. The `sum`-variable is initialized as well, this could be done in the executable code instead (similar to line 29).
Lines 23-26 show an if-statement. The rules are roughly as in Matlab, although negation is written using `!` and not `~`.
The then-part is made up by two statements and they must be grouped together using braces. The braces are not necessary for one statement, but some programmers add them anyhow. The `trapeze` function should always return a value, even when `n` has an illegal value, so the program returns the impossible value, `-1` in that case. The statement, `return value;`, is similar to assigning `value` to the output parameter in Matlab, but `return` also means that we jump back to the main program.

In line 30 we call the exponential function. Note that `x^2` does not work in C (or rather, it means bitwise exclusive OR). Lines 31-34 form a loop, the two statements, 32-33, are grouped together using braces. If we forget the braces, only line 32 will be repeated in the loop, and line 33 will be executed **once** after the loop.

The general format of the for-statement is:

```
for(init; test; update)
  loop body
```

Written with a while loop we understand the meaning:

```
init:
while ( test ) {
  loop body
  update;
}
```

So `k = 1` corresponds to `init`, the `test` is `k < n` and `update` is `k = k + 1`. In words, set `k` to one, then the loop is entered. Repeat the loop body as long as `k < n`. At the end of each loop iteration, the loop variable, `k` is updated by one. To updatee by two, write `k+=2` and to decrease by two use `k-=2` etc.

C has many abbreviations, `k = k+1` can be written `k++` and `a = a + b` can be abbreviated as `a += b`. Using these shorter forms, the loop can be written:

```
for (k = 1; k < n; k++) {
  x += h;
  sum += exp(-x * x);
}
```

Sometimes one can see strange looking loops (at least to a C-novice). The following two loops both compute an approximation to $1 + 1/2 + 1/3 + \cdots + 1/1000$.

```
sum = 0;
k = 1;
for(; k <= 1000;) {
  sum += 1.0 / k;
  k++;
}

sum = 0;
k = 1;
for(;;) {
  sum += 1.0 / k;
  if ( k == 1000 )
    break;         // Jump out of the loop
  k++;
}
```

On line 38 the function returns the value to `main`.

Let us now compile and execute the code:

```
% gcc trap.c -lm
% a.out
The integral is approximately = 7.468180e-01
```

The exact value is approximately 0.74682413. `-lm` informs the compiler that we need to use a library, the mathematics library, since the code calls the exponential function. We say that we *link with the math library.* A special program, the linker, takes care about this part (more about this in the lectures). The math library resides in a file, `/usr/lib64/libm.so` (for our 64-bit system). The `m`-part of `libm` is what is used in `-lm`. Some compilers do not require that we write `-lm`, but they will link with library automatically. If we forget it on our system we get a link error:

```
% gcc trap.c
/tmp/ccgqMVKZ.o(.text+0xd2): In function ‘trapeze’:
: undefined reference to ‘exp’

  etc.
```

4 Separate compilation and `ld`

In the example I have stored both `main` and `trapeze` in the same file `trap.c`. This would be unrealistic in large applications, however, so it is possible to split the file into separate files. So, suppose that we have two files, `trap_main.c` containing lines 1, 4-11 (i.e. not line 2, since `main` does not use `exp`), and `trapeze.c` containing lines 2, 13-39. Here are two ways to compile the code.

```
% gcc trap_main.c trapeze.c -lm
% a.out
The integral is approximately = 7.468180e-01
```

If a large part of a program does not change, we can compile that part once and for all. In the first `gcc`-command I compile `trapeze.c`, using the `-c` flag (option). This tells the compiler to produce an object file, `trapeze.o`, but not to try to produce an executable. The object file is later used when compiling `trap_main.c`. We save time by not having to recompile `trapeze.c` (think of a file containing thousands of lines).

```
% gcc -c trapeze.c                an object file is produced
% ls -l trapeze.o
-rw-------  1 thomas _math 1232 Nov 18 15:49 trapeze.o

% gcc trap_main.c trapeze.o -lm   use it here
% a.out
The integral is approximately = 7.468180e-01
```

If we forget `trapeze.o` we get a *link error*.

```
% gcc trap_main.c
/tmp/ccgkJmlR.o(.text+0x3d): In function ‘main’: undefined reference to ‘trapeze’
collect2: ld returned 1 exit status
```

We will get the same effect if we make a spelling error when calling `trapeze`. Say we type `Trapeze` instead of `trapeze` in the `printf` statement in `main`. We get:

```
% gcc trap_main.c trapeze.o -lm
/tmp/cc4JCXzK.o(.text+0x29): In function ‘main’: undefined reference to ‘Trapeze’
collect2: ld returned 1 exit status
```

even though `trapeze.o` is included. The reason is that C is case sensitive, `trapeze` and `Trapeze` refer to different functions. `ld`, which is mentioned, is the so-called *linker*, which combines object files, libraries (e.g. the math library) to an executable. This is not the whole truth (there is a dynamic linker as well), but it is accurate enough for this course. So, the `gcc`-command does not only compile, but it runs `cpp` and `ld` as well.

5 More on prototypes and type conversion

It is easier to appreciate the prototypes when we use separate compilation (different files). Suppose we have written `trapeze(0, 100)` in `main`. The compiler complains:

```
% gcc trap_main.c trapeze.o -lm
trap_main.c: In function ‘main’:
trap_main.c:8: error: too few arguments to function ‘trapeze’
```

If we remove the prototype, the following happens:

```
% gcc trap_main.c trapeze.o -lm
% a.out
The integral is approximately = 7.234109e-320
```

So, no complaints and the wrong answer. This is different from Java, which would complain. A C-programmer must be more careful. Be **very careful** when you call a function. Check the number and types of parameters. I have been slightly careless when calling `trapeze`. `0` and `1` are **integer** constants, but since I have provided a prototype, the compiler will automatically convert the numbers to the corresponding double precision constants, `0.0` and `1.0`. To avoid the type conversion I could have written `trapeze(0.0, 1.0, 100)`. The reverse can happen, a double value can be truncated to an integer value (the decimals will deleted).
Study the following example (`%d` is a format for printing integers):

```
% cat trunc_ex.c
#include <stdio.h>

int trunc_ex(int, double);

int main()
{
  double result;

  result = trunc_ex(1.99, 23);
  printf("trunc_ex = %e\n", result);

  return 0;
}

int trunc_ex(int k, double d)
{
  printf("k = %d, d = %e\n", k, d);

  return 3.1415926535897932;
}

% gcc trunc_ex.c
% a.out
k = 1, d = 2.300000e+01
trunc_ex = 3.000000e+00
```

If we remove the prototype, the compiler will not make the conversions for us. Instead we end up with garbage:

```
% gcc trunc_ex.c
% a.out
k = 1030792151, d = 4.933640e-313
trunc_ex = 3.000000e+00
```

In `main`, `1.99` is stored as an 8 byte double precision number and 23 as a four byte integer. When `trunc_ex` is called it will pick up the first four bytes of the stored double, and interpret those bytes as an integer. To access `d` the function will take the four bytes from `23` and the next four bytes, whatever they contain, and make a double precision number of the eight bytes. Note that no conversion is made for either number, `trunc_ex` will just read the bits and make numbers from them. Finally, the reason we get the correct conversion of `3.1415926535897932` is that a function is of type `int`, by default.

Division with integers behaves in a special way (but the same rule applies to C++, Fortran, Java etc).
Integer division produces integer quotients, decimals are truncated. `5 / 2` will be `2`, `-2 / 5` becomes `0` etc.
`5.0 / 2` or `5 / 2.0` or `5.0 / 2.0` will all give you `2.5` since the integer will be converted to the "dominating type" double before the division. Note that `10.0 * (1 / 10)` is `0.0`, since `1 / 10` is computed first, giving `0`. The integer zero is then converted to `0.0` and the product is `0.0`.

6 void functions, passing parameters

The functions we have seen so far return values. There are functions that do not return values this way, a so-called *void function.* A void function corresponds to a Matlab function, looking something like
`function function_name(list of parameters)` (so no return variable).
The difference is that one can write a C-function so that it can change its input parameters (this is not possible in Matlab). This makes it necessary to discuss how parameters (arguments) are passed when a function is called. Let us look at `trapeze` again.

```
double trapeze(double a, double b, int n)
{ ...  }
```

The function works with **copies** of `a`, `b` and `n`, so if the function changes one of the variables, the original variables (or constants) in `main` will **not** change. This way of passing parameters is called *call-by-value.*

In order to be able to change a variable, we use *call-by-reference*, i.e. we will pass the memory-*address* of the variable rather than the variable's value. Since the function has access to the address, it can change the value of the variable. If `var` is the name of an integer or double variable, `&var` is its address, and `&` is called the *address operator.* We also say that `&var` is a *pointer* to `var`. If `adr` is an address to a location in memory, `*adr` is the corresponding value of what is stored there. Using `*` is called *dereferencing* or *indirection*, `*` is the *indirection* or *redirection operator.* An address to a variable is often called a reference (like in Java programming).

Time for an example. This piece of code computes approximations to $\sum_{k=1}^{n} 1/k$ and $\sum_{k=1}^{n} 1/k^2$.

```
#include <stdio.h>

void sums(double *, double *, int);

int main()
{
  double sum1, sum2;

  sums(&sum1, &sum2, 1000);
  printf("The sums are: %e and %e\n", sum1, sum2);

  return 0;
}

void sums(double *a_sum1, double *a_sum2, int n)
{
  int k;

19    *a_sum1 = 0.0;
20    *a_sum2 = 0.0;
21
22    for (k = 1; k <= n; k++) {
23      *a_sum1 += 1.0 / k;          /* 1.0 to avoid integer divsion */
24      *a_sum2 += 1.0 / (k * k);
25    }
26  }
```

```
% gcc sums.c
% a.out
The sums are: 7.485471e+00 and 1.643935e+00
```

Let us start with the `sums` function, lines 15-26. We have a `void` function which takes three parameters, the third is the number of terms. `double *a_sum1` should be read in the following way. `*a_sum1` is a `double`, and `*` is the indirection operator, so `a_sum1` must be an address to a `double`. I have tried to indicate this fact by naming the variable `a_sum1`, `a` for address. This is for pedagogical reasons, one would usually name the variable `sum1` and write `double *sum1`. We can now understand the prototype on line 3. The first (and second) argument is of type `double *`, a pointer to `double`.

On lines 19, 20 I set the values to zero. We should **not** try to set the addresses to zero. Note that we use the same syntax on lines 23 and 24. Note that we use `1.0 / k` rather than `1 / k` (in which case the sum would be one, since `1 / k = 0` when `k > 1`).

Let us now look at the main program. On line 7 we define `sum1` and `sum2` as ordinary `double` variables. On line 9 we call the function. Note that since we have a `void` function, it is illegal to try and write something like `variable = sums(...)`, since `sums` does not return a value in its name. Note that we pass the **addresses** of `sum1` and `sum2`, it would be **wrong** to write `sums(sum1, sum2, 1000);`.

If you think these things are hard to follow, you should know that you are not alone, most beginners to C find this a bit hard.

Let us declare two *pointer variables* by adding the following line to the code (after line 7):

```
double *p1, *p2;
```

So, `p1` can point at a `double` variable, it can contain the address of a double precision variable. We can set `p1` to point at `sum1` and `p2` to point at `sum2`, like in the piece of code:

```
p1 = &sum1;
p2 = &sum2;
sums(p1, p2, 1000);
printf("The sums are: %e and %e\n", *p1, *p2);
```

but even

```
printf("The sums are: %e and %e\n", sum1, sum2);
```

How, you may ask, can we print `sum1` and `sum2`, even though these variables have not been passed as arguments to `sums`? The explanation, is that we passed the pointers, and `sums` can access the memory where `sum1` and `sum2` are stored, through the pointers.

Note that the following programming will end in tears (the remaining code remains unchanged):

```
int main()
{
  double *p1, *p2;

  sums(p1, p2, 1000);
  printf("The sums are: %e and %e\n", *p1, *p2);

  return 0;
}
```

When we try to run it we get the feared error message:

```
% gcc sums3.c
% a.out
Segmentation fault
```

A *Segmentation fault* (or abbreviated *segfault*) can be a nasty error, at least if we have a large complicated program, since the bug can be very hard to find. It is caused by the program trying to access a memory location which it is not allowed to access, or it may try to write to a read-only part of the memory. Another message of the same type is *Bus error*, where the program may try to access a non-existent address, for example. In the sums-example it is very easy to find the bug. We have allocated memory for the pointer variables, but have not allocated memory for the summation variables. So `p1` and `p2` do not point to any variables, the pointers have not been assigned any values, they point to random addresses in memory. The program crashes in `sums` when `*a_sum1 = 0.0;` is executed.

Here comes another example where we must use addresses. We must use call-by-reference when reading data, here are a few lines of code:

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    int i;
 6    double d;
 7
 8    printf("type a value for i: ");
 9    scanf("%d", &i);
10
11    printf("type a value for d: ");
12    scanf("%le", &d);
13
14    printf("i = %d, d = %e\n", i, d);
15
16    return 0;
17  }
```

```
% a.out
type a value for i: -123
type a value for d: -1.23e-45
i = -123, d = -1.230000e-45
```

On order for `scanf` to be able to return a value we must supply a pointer to the variable. On lines 8 and 11 we do not supply a newline, that is why we can type the input on the same line as the prompt text. Note on line 12 that is says `le` (the letter ℓ) for long. If we omit the letter, `scanf` will try to read a single precision number instead of a double. This will lead to a conversion error:

```
type a value for i: 12
type a value for d: -1.23e3
i = 12, d = 3.713054e-307
```

Suppose we have a `non-void` function. In that case it is bad programming practice to return values in the input parameters as well (even though it is possible). We say that the function has side-effects.

7 Arrays

In this program we create a one-dimensional array (vector) containing ten elements. We call the function `init` to initialize the elements to $1, 2, \ldots, 10$. Finally we compute the sum of the element using the function `array_sum`.

```
 1  #include <stdio.h>
 2
 3  void init(double [], int);
 4  double array_sum(double [], int);
 5
 6  int main()
 7  {
 8    double vec[10];
 9
10    init(vec, 10);
11    printf("The sum is: %e\n", array_sum(vec, 10));
12
13    return 0;
14  }
15
16  void init(double v[10], int n)
17  {
18    int k;
19
20    for(k = 0; k < 10; k++)
21      v[k] = k + 1;
22  }
23
24  double array_sum(double v[10], int n)
25  {
26    int k;
27    double sum;
28
29    sum = 0.0;
30    for(k = 0; k < 10; k++)
31      sum += v[k];
32
33    return sum;
34  }
```

```
% a.out
The sum is: 5.500000e+01
```

On line 8 we reserve storage for an array having ten double elements. Indices start at zero and end at nine, unlike Matlab. Note that we use `[ ]` for the index. So, the loop variables in the loops, e.g. on line 20, go from

zero to nine. It would be inefficient to copy the array when the functions are called. Instead call-by-reference is used. So, if the function changes an element in the array, it changes the original. We do this in the `init`-routine. Note that we should not use the address or indirection operators for the array.
Compare the prototypes, lines 3, 4, with the function declarations, lines 16, 24. It is allowed to leave out the dimension of the array. So line 16 can be written

```
void init(double v[], int n)
```

and analogously for line 24. The reason is that the compiler does not need to know the number of elements in the array, the find the address of a specific element. Note also that an array in C is not some kind of object, like in Java. A function does not know the number of elements in the array unless we pass that information in an extra argument (the variable `n` in the example). In fact, when we call the function, only the address of `v[0]` is sent to the function. We could actually call `init` this way:

```
init(&vec[0], 10);
```

There is a close relationship between pointers and arrays but I leave that out in this introduction.

One should know that there is no index control in C. Changing the loop in `init` to

```
for(k = -3; k < 11; k++)
  v[k] = k + 1;
```

causes no complaints, but nasty things may happen as in the following example.

```
 1  void func(double a[]);
 2
 3  #include <stdio.h>
 4  main()
 5  {
 6    double b, a[10];
 7
 8    b = 1;
 9    func(a);
10
11    printf("%f\n", b);
12
13    return 0;
14  }
15
16  void func(double a[])
17  {
18    a[11] = 12345.0;
19  }
```

```
% gcc nasty.c
% a.out
12345.000000
```

On line 8 we set `b` to one, and then, on line 9, we call `func` with the array, `a`. When we print `b` on line 10, **the value has changed**, even though `b` is not an argument to the function. This is very nasty, and can be very hard to find in a large program. What is going on? The elements of a one-dimensional array is stored consecutively, with no gaps, in memory. One can find out the addresses of the elements in the array and of the variable `b`, and it turns out that `b` is stored in a position that would correspond to `a[11]`, provided `a` had twelve elements. Changing `a[11]` to `a[1000000]`, for example, gives Segmentation fault.

7.1 Two-dimensional arrays

You can find more about arrays at the end of this tutorial. Here is one small example where we multiply two 4×4-matrices together.

```
 1 #include <stdio.h>
 2
 3 void mat_mul(double[4][4], double[4][4], double[4][4]);
 4 void mat_print(double[4][4]);
 5
 6 main()
 7 {
 8   int row, col;
 9   double A[4][4], B[4][4], C[4][4];
10
11   for (row = 0; row < 4; row++)
12     for (col = 0; col < 4; col++) {
13       A[row][col] = row + col;
14       B[row][col] = row - col;
15     }
16
17   mat_mul(A, B, C);
18   mat_print(C);
19
20   return 0;
21 }
22
23 void mat_mul(double A[4][4], double B[4][4], double C[4][4])
24 {
25   int row, col, k;
26   double sum;
27
28   for (row = 0; row < 4; row++)
29     for (col = 0; col < 4; col++) {
30       sum = 0.0;
31       for (k = 0; k < 4; k++)
32         sum += A[row][k] * B[k][col];
33       C[row][col] = sum;
34     }
35 }
36
37 void mat_print(double C[4][4])
38 {
39   int row, col;
40
41   for (row = 0; row < 4; row++) {
42     for (col = 0; col < 4; col++)
43       printf("%8.2f ", C[row][col]);
44     printf("\n");
45   }
46 }
```

One could write a more general code, but this is all we need. Line 37 can be written:

```
void mat_print(double C[][4])
```

but not

```
void mat_print(double C[][])
```

for example. The reason is that C stores matrices row-major order (row after row), in memory. So the memory layout, of the matrix `C`, for example, would be:

```
address
base         C[0][0]  (see the note about byte addressable
base + 1     C[0][1]   memory below)
base + 2     C[0][2]
base + 3     C[0][3]
base + 4     C[1][0]
base + 5     C[1][1]
base + 6     C[1][2]
base + 7     C[1][3]
base + 8     C[2][0]
        etc.
```

The compiler knows the baseaddress, `base = &C[0][0]`, and to compute `&C[row][col]` it needs to know the number of elements in a row, `row_len`, say (four in the example).

```
  &C[row][col] = base + row_len * row + col
```

If one should be picky, the memory on one of our machines is byte addressable, and since a double precision variable is stored using eight bytes, the correct formula is:

```
  &C[row][col] = base + 8 * (row_len * row + col)
```

So this is the reason why `void mat_print(double C[][4])` is sufficient, but `void mat_print(double C[4][])` or `void mat_print(double C[][])` are not.

8 A matter of style

The placement of braces on other details of programming style, has been the focus of many heated and lengthy debates. In all my examples I have placed the braces using a special style, e.g:

```
    for (k = 1; k < n; k++) {
        x += dx;
        sum += exp(-x * x);
    }
```

This style is known as the "Kernighan & Ritchie coding style" and comes from the classic book I mentioned on page one. One can write this piece of code in other ways, e.g.

```
  for (k = 1; k < n; k++)
    {
      x += dx;
      sum += exp (-x * x);
    }
```

which is the GNU-style, used to write GNU software. I will not start a debate about it in this introduction; find your own style and stick to it. One style I do not recommend is:

```
  for(k=1;k<n;k=k++){x+=+dx;sum+=exp(-x*x);}
```

`indent` is a very useful command for pretty printing, formatting, C-programs. There are many options, I use the following:

```
indent -kr -i2 -nut my_program.c
```

`-kr` is the Kernighan & Ritchie style, `-i2` means two spaces for indentation in loops and if-statements etc, `-nut` means that spaces and not tabs are used for indentation.

```
indent -gnu -i2 -nut my_program.c
```

gives you the GNU style instead.

The choice of style affects other parts of the program as well, e.g. the position of braces in if-statements, and the layout of comments and declarations.
To read about the different styles, type `man indent`, and read under `COMMON STYLES`. If you use `indent` on a program with syntax errors, `indent` may produce an incorrectly indented program (if a brace is missing, for example). For that reason, `indent`, makes a backup copy of your original file. In my example the copy is stored in `my_program.c~`.

9 If-statements and logical expressions

Here are a few examples. Logcal *and* is written `&&`, *or* is `||` and *negation* `!` . Note single `&` and `|` are **bitwise** operations.

```
double a, b, c, d, q;

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

The relational operators, <, <=, ==, >=,> are written the same way as in Matlab, with the exception of “not equal” which is written `!=`.

Note: `if ( ! q == 1.25 )` $\Leftrightarrow$ `if ( (!q) == 1.25 )`, not `if( ! ( q == 1.25) )`.

Now a word about the so-called dangling else. When we have nested if-statements, the `else` belongs to the innermost if-statements, so with correct indentation this is how it works:

```
if ( condition )
  if ( other condition ) {
    statements
  } else {
    statements
  }
```

If you want the `else` to belong the outer if, use braces:

```
if ( condition ) {
  if ( other condition ) {
    statements
  }
} else {
  statements
}
```

10 Some useful C-tools

Consider the following lines (part of `warn.c`):

```
  if ( variable = 24 )
    printf("var equals 24\n");
```

This is probably not what we meant (an assignment), we probably meant "`if ( variable == 24 )`". The compiler warns us, provided we switch on the `-Wall` flag, thus:

```
% gcc warn.c     No warning

% gcc -Wall warn.c
warn.c: In function 'main':
warn.c:8: warning: suggest parentheses around assignment used as truth value
```

`gcc` actually warns us against something slightly different. Assignments in if-statements are typically used in the following situation

```
  if ( (variable = func()) == test_value )
```

where the parentheses are necessary, since `==` has higher priority than `=`.

Another useful tool is `splint`, "secure programming lint" which checks C-programs for security vulnerabilities and coding mistakes. `splint` analyzes the code without executing it, so runtime errors are not caught. `splint` on the example above gives:

```
% splint -weak warn.c
Splint 3.1.1 --- 19 Jul 2006

warn.c: (in function main)
warn.c:8:8: Test expression for if is assignment expression: variable = 24
  The condition test is an assignment expression. Probably, you mean to use ==
  instead of =. If an assignment is intended, add an extra parentheses nesting
  (e.g., if ((a = b)) ...) to suppress this message. (Use -predassign to
  inhibit warning)

Finished checking --- 1 code warning
```

`splint` without `-weak` gives an additional warning:

```
warn.c:8:8: Test expression for if not boolean, type int: variable = 24
  Test expression type is not boolean or int. (Use -predboolint to inhibit
  warning)
```

If you want a very strict check try `splint -strict`.

11 A few words about the C99 standard

Note that it is not supported by all compilers.
C99 extends the previous C-version, C89, and adds support for (among other things):

- a boolean data type, complex numbers
- intermingled declarations and code

- //-comments
- inline functions
- variable-length arrays
- restrict qualifier to allow more aggressive code optimization (more later on)

Here a few lines showing how to use the boolean data type:

```
#include <stdbool.h>
 ...
  bool b;

  b = a > b;
  b = true;
  b = false;
 ...
```

Here complex numbers:

```
#include <complex.h>
 ...
  double complex z, w, wz;

  z = 1 + 2 * I;
  w = 3 + 4 * I;
  wz = 3 * w * z;

  printf("%e %e\n", creal(wz), cimag(wz));
 ...
```

Intermingled declarations and code:

```
#include <stdio.h>

int main()
{
  int k = 22;

  for(int k = 0; k <= 2; k++)  // C++ declaration style
    printf("%d\n", k);

  printf("%d\n", k);

  return 0;
}

% gcc -std=c99 c99_mixed.c   NOTE
% a.out
0
1
2
22
```

Inline functions. From the C-standard:

```
  inline double func(double x)
    ...
```

> Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined.

Variable-length arrays:

```
double func1(int, int);
double func2(int m, int n, double A[m][n]);
double func3(double A[m][n], int m, int n);  // WRONG

int main()
{
  int  m = 50, n = 100;
  double vec[n], A[m][n];  // m, n OK here
  ...
}

double func1(int m, int n)
{
  double A[m][n], tmp[n];  // allocated when entering func1

  func2(m, n, A);
  ...
}

double func2(int m, int n, double A[m][n])  // OK, m, n first then A
{
  ...
}

double func3(double A[m][n], int m, int n) // WRONG, A first, m, n last
{
  ...
}
```

12 More on cpp

The gcc-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. It is possible to 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 prototypes, e.g:

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

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 ...
```

13 Using the `man`-command

One way of finding out what header-files are necessary, is to use the `man`ual-command, e.g:

```
% 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. One can try to make a keyword search: `man -k keyword`.

14 More on matrices

In Fortran (dense) matrices are stored in the same way in (almost) all programs. This is because the matrix is a builtin type in Fortran and the language has a lot of support for matrix computations. This is not the case in C, and so there are several possible data structures for storing matrices. It is important to pick the proper data structure if the matrix should be passed as an argument to a Fortran routine or used together with a performance library. Another issue is how we would like to access the elements in the matrix. Is it important to be able to write `A[row][col]` or will `*(A + row * n + col)` do?

Here comes a short description of some alternative data structures. Suppose we would like to store the matrix:

$$\mathbf{A} = \left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right]$$

The most obvious way is illustrated by the following short program.

```
#include <stdio.h>
int main()
{
  double A[2][3], elem = 0;
  int    row, col;

  for(row = 0; row < 2; row++)
    for(col = 0; col < 3; col++)
      A[row][col] = ++elem;

  return 0;
}
```

This way to create matrices is rather limited. We would like to have a more dynamic choice of dimensions. The first step would be something like:

```
#include <stdio.h>
int main()
{
  const int m = 2, n = 3;
  double A[m][n];
    ...
```

Some compilers accept such constructions, but not all (see page 21). The following is allowed, but a bit clumsy:

```
#include <stdio.h>
#define _M 2
#define _N 3
int main()
{
  const int m = _M, n = _N;
  double A[_M][_N];
    ...
```

Such a matrix can be passed as a parameter to a Fortran program.

14.1 Dynamic memory allocation

Some assignments in the course require that tests should be performed for a sequence of matrices of increasing sizes. It is inconvenient having to edit the program, changing the dimensions, recompiling etc. This leads us to dynamic memory allocation. So first a few words about that.

The C-library routines `malloc` and `free` are used to allocate memory and to return it. `stdlib.h` contains the prototypes. In C++ we have `new` and `delete`. Java has garbage collection, so only `new` is necessary. Fortran90 has `allocate` and `deallocate`.
We will concentrate on C from now on. `ptr = malloc(size)` returns a pointer, `ptr`, to a block of data at least `size` bytes suitably aligned for any use. If there is not enough available memory `ptr` will be a null pointer.
`free(ptr)` will return the memory to the application, though not to the system. Memory is returned to the system only upon termination of the application. If `ptr` is a null pointer, no action occurs. It is illegal to free the same memory more than once, to try to use freed memory and to free using a pointer not obtained from `malloc`.

Here is a typical piece of code where we allocate 100 double precision numbers. Note the use of `sizeof` and the check on the pointer value. We then store some values in the memory. The first loops uses pointer arithmetic and the second uses vector notation. Note that `vec` is a pointer and not a vector but it is allowed to mix the notation.
There are differences between vectors and pointers though. If we have the declaration:

```
double *vec, vector[100];
```

`vec` can point to something else but `vector` cannot. We need space for the pointer variable, `vec`, but `vector` itself takes no space,

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  double *vec;       // vec is a pointer to double
  int n = 100, k;

  if( (vec = malloc(n * sizeof(double))) == NULL ) { // sizeof(double) = 8
    printf("malloc of vec failed.\n");
    exit(EXIT_FAILURE);  // EXIT_FAILURE a named constant defined in stdlib.h
  }

  for(k = 0; k < n; k++)
   *(vec + k) = k;        // pointer notation

  for(k = 0; k < n; k++)
   vec[k] = k;            // vector notation

  free(vec);              // release the memory

  return 0;
}
```

What I would like to do is to allocate memory for an $m \times n$-matrix `A`, using `malloc`, and then pass `A` as an argument to a function, receiving `A` as an $m \times n$-matrix so that I can use matrix-indexing `A[row][col]` inside the function. This can be done with some trickery (and with some compilers), but I do not know how to do it in a completely legal way (following the C-standard) unless I use variable-length arrays defined in C99, see page 21.

The following pages show some common ways of storing matrices in C. Some ways can work together with Fortran and some cannot. We know that a matrix is stored by rows in C. So if `A` is the address of the `[0][0]`-element, `A[row][col]` has address `A + n * row + col` where `n` is the number of elements in a row. We can use vector indexing instead of using pointer arithmetic. Here is an example (to make the code shorter I will not check that `malloc` succeeded, a bad programming practice). I have added a function to show how the parameter could be passed.

```
#include <stdio.h>
#include <stdlib.h>

double sum_elements(double *A, int m, int n);

int main()
{
  double *A;
  int m = 2, n = 3, k;

  A = malloc(m * n * sizeof(double)); // Allocate memory for the m x n-matrix

  for(k = 0; k < m * n; k++)
    A[k] = k + 1;                     // This is ONE way to access the elements

  printf("result = %e\n", sum_elements(A, m, n));

  free(A);
  return 0;
}

double sum_elements(double *A, int m, int n)
{
  double sum = 0;
  int row, col;

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      sum += A[n * row + col];        // This simulates A[row][col]-access.
                                      // We could use pointer notation.
  return sum;
}
```

One advantage of this approach is that it easy to pass the array as an argument to a Fortran routine (and it is easy to store the matrix by columns instead). Note that Fortran uses column-major order and not row-major order as C.

14.2 This will not work with Fortran

Here come two other ways to store a matrix. The first method does **not work** together with Fortran though, but the other does. Both methods support `A[row][col]`-indexing.
Here comes the first example:

```
#include <stdio.h>
#include <stdlib.h>

double sum_elements(double **A, int m, int n);

int main()
{
  double **A, elem = 0;                   // Note **
  int m = 2, n = 3, row, col;

  A = malloc(m * sizeof(double *));       // Allocate space for row pointers.
                                          // Note double * . sizeof(double *) = 8.

  for(row = 0; row < m; row++)
    A[row] = malloc(n * sizeof(double));  // Allocate space for elements in a row.
                                          // Note double.

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      A[row][col] = ++elem;               // Note A[row][col]

  printf("result = %e\n", sum_elements(A, m, n));

  for(row = 0; row < m; row++)            // free
    free(A[row]);

  free(A);                                // free again,
                                          // Note the order of the calls to free.

  return 0;
}

double sum_elements(double **A, int m, int n)
{
  double sum = 0;
  int row, col;

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      sum += A[row][col];

  return sum;
}
```

The memory layout *may* look something like this after we have initialised the matrix. The arrows show how the addresses point.

```
        variable     content    address

   +---->    A[0]      135200     134168  --+
   |         A[1]      135232     134176  --|--+
   |                                        |  |
   |      A[0][0]           1     135200 <--+  |  start of first row
   |      A[0][1]           2     135208       |
   |      A[0][2]           3     135216       |
   |                              135224       |  Note, a gap
   |      A[1][0]           4     135232 <-----+  start of second row
   |      A[1][1]           5     135240
   |      A[1][2]           6     135248
   |
   +----        A      134168     429080
```

`A` points to `A[0]` which in turn points to `A[0][0]`, the first element in the first row. `A[1]` points to the beginning of the second row, i.e. `A[1][0]`. The first `malloc` allocates space for `A[0]` and `A[1]` (`m` row pointers). Then comes a loop with `m` calls to `malloc` where each one allocates memory for storing the `n` elements in row.

We note that `sizeof(double *)` is eight since `A[0]` and `A[1]` are eight bytes apart (134176-134168=8). The double precision numbers are eight bytes apart, except between `A[0][2]` and `A[1][0]` where the gap happens to be 16 bytes. This is the reason this data structure cannot be used when calling Fortran routines, the elements are **not** contiguous in memory.
One advantage with this data structure is that all the rows need not have the same length.
Note also that this storage requires more memory than the usual matrix data structure (we need extra space for the row pointers). That is true with the next method as well, but it has the advantage of giving contiguous elements, making it possible to pass the array to a Fortran routine.

14.3 This will work with Fortran

```
 ...
  double **A;

  A = malloc(m * sizeof(double *));       // Allocate space for row pointers.
                                          // Note double * .

  A[0] = malloc(m * n * sizeof(double));  // Allocate space for the elements in the matrix.
                                          // Note that we get contiguous elements.
  for(row = 1; row < m; row++)
    A[row] = A[0] + row * n;              // Give the row pointers their values, i.e.
                                          // find out where each row starts.
                                          // There are n elements in each row.

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      A[row][col] = ++elem;
 ...
```

The memory layout may look something like this after we have initialised the matrix. The arrows show how the addresses point.

```
         variable     content    address

    +---->    A[0]      135768     134744  --+
    |         A[1]      135792     134752  --|--+
    |                                        |  |
    |      A[0][0]           1     135768 <--+  |  start of first row
    |      A[0][1]           2     135776       |
    |      A[0][2]           3     135784       |
    |      A[1][0]           4     135792 <-----+  start of second row
    |      A[1][1]           5     135800
    |      A[1][2]           6     135808
    |
    +----        A      134744     429080
```

To pass the array to Fortran we use the parameter `&A[0][0]`, `A[0]` or `*A`.

For more details about this and other topics, see the C-FAQ:
`http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c.html`

14.4 C and large arrays

Some of the assignments require that you use large arrays. This may be a problem in C. Consider the following program:

```
#include <stdio.h>

main()
{
  int k, n = 2000000;
  double large_array[n];

  for(k = 0; k < n; k++)
    large_array[k] = 1;

  printf("Last %f\n", large_array[n - 1]);

  return 0;
}
```

When we try to run it we get:

```
% gcc stack_problems_1.c
% a.out
Segmentation fault
```

The reason is that `large_array` is allocated on the stack, which has a limited size. We can find out the size by using the command limit. Thus:

```
% limit  (works provided you use tcsh, type  ulimit -a  if you are using bash)
cputime      unlimited
filesize     unlimited
datasize     unlimited
```

```
stacksize    10240 kbytes
coredumpsize 0 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      500
```

So, the stack is limited to 10240 kbyte, but we need 2000000 * 8 / 1024 kbyte, i.e. 15625 kbyte (the stack is used for some other purposes as well so it must be a bit larger). So, let us increase the stack size and try again:

```
% limit stacksize 15700   (in bash ulimit -s 15700)
% a.out
Last 1.000000
```

Another way is to store the array in a segment in `a.out`. If we make `large_array` static, i.e. we have the type declaration: `static double large_array[2000000];` our program will work with the default stack size. The array is now stored in the bss-segment.

```
% gcc stack_problems_2.c
% limit stacksize
stacksize 8192 kbytes
% a.out
Last 1.000000

% size a.out
text    data     bss              dec            hex filename
 925     252 16000032         16001209         f428b9 a.out
```

One drawback with static variables is that they exist for the lifetime of the program (even if we do not use the array). So, yet another way (common) is to use dynamic memory allocation (i.e. we use `malloc`/`free`) placing the array on the heap:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int k, n = 2000000;
  double *large_array;

  if ( (large_array = malloc(n * sizeof(double))) == NULL) {
    printf("Could not malloc large_array.\n");
    exit(EXIT_FAILURE);
  }

  for(k = 0; k < n; k++)
    large_array[k] = 1;

  printf("Last %f\n", large_array[n - 1]);

  free(large_array);

  return 0;
}
```

15 A note on structs and const

In one lecture and the corresponding lab we will use structs (records in Pascal, post in Swedish). Think of a struct as a box, a container, in which we can store variables. We will also write the function:

```
  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

where `mxArray` is a struct. This section will also explain what `const`, in `const mxArray *prhs[]` means.

`const` is used to protect variables against accidental change. It also serves as documentation for anyone reading the code (this variable is not supposed to be changed). We start with some simpler examples, protecting scalar variables.

15.1 Structures

Here is a toy-example of a struct, `my_mxArray`, where we store a pointer to double (will point to allocated doubles) and an integer storing the number of allocated.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  double *vec;
  int n;
} my_mxArray;

int main()
{
  // mx is a pointer to my_mxArray and pmx is a vector of pointers to my_mxArray
  my_mxArray mx, *pmx[2];

  // Store values in mx
  mx.n = 20;
  mx.vec = malloc(mx.n * sizeof(double));
  for(int k = 0; k < mx.n; k++)
    *(mx.vec + k) = k;

  printf("%f, %f\n", *(mx.vec), *(mx.vec + mx.n - 1));

  // Allocate space for the my_mxArray:s
  pmx[0] = malloc(sizeof(my_mxArray));
  pmx[1] = malloc(sizeof(my_mxArray));

  // and store values in the two structs
  (*pmx[0]).n   = 30;  // *pmx[0].n is wrong
  pmx[1] -> n   = 40;  // a correct alternative (see the precedence table)
  pmx[0] -> vec = malloc(pmx[0] -> n * sizeof(double));
  pmx[1] -> vec = malloc(pmx[1] -> n * sizeof(double));

  for(int k = 0; k < pmx[0] -> n; k++)
    *(pmx[0] -> vec + k) = k;

  printf("%f, %f\n", *(pmx[0] -> vec),   *(pmx[0] -> vec + pmx[0] -> n - 1));
  free(mx.vec);
  free(pmx[0]);
  free(pmx[1]);

  return 0;
}
```

15.2 const

```
const_ex1(const double a, const double *b, double * const c, const double * const d)
{
 a++;     // error: protects a
 b++;     // OK:    can change the pointer itself
 (*b)++;  // error: but not what it points to
 c++;     // error: protects the pointer
 (*c)++;  // OK:    but not what it points to
 d++;     // error: protects the pointer
 (*d)++;  // error: and what it points to
}

% gcc -c const_ex1.c
const_ex1.c: In function 'const_ex1':
const_ex1.c:4: error: increment of read-only location
const_ex1.c:6: error: increment of read-only location
const_ex1.c:7: error: increment of read-only location
const_ex1.c:9: error: increment of read-only location
const_ex1.c:10: error: increment of read-only location
```

`const double a` and `double const a` are equivalent, so those forms have not been included above.

15.3 const and structures

Now for a mix of const and structures. "illegal" means that the compiler will complain, OK means that it will not complain (though it may not be useful programming). To simplify the example we redfined struct.

```
typedef struct {
  double v[10];
  int n;
} my_mxArray;

void ex1(const my_mxArray * s[])    // or void ex1(my_mxArray const * s[])
{
  (*s[0]).v[5]  = 7;   // illegal
  s[0] -> v[5]  = 7;   // illegal
  s[1] = s[0];         // OK
}

void ex2(my_mxArray * const s[])
{
  s[0] -> v[5] = 7;    // OK
  s[1] = s[0];         // illegal
}
```

```
void ex3(const my_mxArray * const s[])
{
  s[0] -> v[5] = 7;     // illegal
  s[1] = s[0];          // illegal
}

void ex4(const my_mxArray * const s[])
{
  double *pv;

  pv = (double *) s[0] -> v;
  pv[5] = 7;            // OK, the compiler cannot protect us against everything
}
```

In the last example we “fool” the compiler, changing the protected v-array using the pointer pv.

16 Assert, a debugging tool

assert can be used to generate error messages, both as a debugging tool and as a mechanism to generate warnings. Here is a short example:

```
#include <assert.h>
#include <stdio.h>

int main()
{
  double x;

  x = 10;
  assert(x <= 5);

  return 0;
}

./a.out
a.out: assert.c:9: main: Assertion ‘x <= 5’ failed.
Abort
```

17 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	
`&`	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 not 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 our 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, it is hard to read and one may 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.

18 A short introduction to Fortran

The next few pages contain the rudiments of Fortran 90 and a glance at FORTRAN 77. This text 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.

FORTRAN was designed by a group, lead by John W. Backus at IBM. The language was proposed as alternative to coding in assembly language and the first compiler appeared in 1957. The language evolved through the years but you will still find huge amounts of code written in FORTRAN 66 (came in 1966) and FORTRAN 77. So you must be able to read such code but you probably do not have to write any (if you are lucky :-). You will use some FORTRAN 77-code in one of the labs.
Fortran 90 (no capitalization) is essentially a new language. It has array operations (somewhat like MATLAB), pointers, recursion, prototypes, modules, overloading of operators (and more). FORTRAN 77 has none of these. Fortran 90 is much nicer than FORTRAN 77, which is quite primitive. Fortran 95 was a minor revision of Fortran 90, but Fortran 2003 is a major update containing support for object-oriented programming. The most recent version is Fortran 2008, which is a minor upgrade of the 2003-version. This text does not contain any code-examples in Fortran 2003 or Fortran 2008 and not many compilers have full support for these latest versions.
Fortran is used primarily for scientific computing, and in this author's view the language is much better suited for such tasks than C. The main reason is that Fortran has better support for vectors and matrices.

We start with a Fortran 90-example, but first some basic rules. Fortran 90 (and later) supports a free format, you can write the statements wherever you like on the line (not so in FORTRAN 77 and earlier). Case or blanks (space) are not significant (unless they are in strings or serve as a separator between keywords). The examples contain extra blanks to increase readability. Comments are written using `!`. A statement ends when the line ends. Long statements can be continued by adding `&` at the end of the line that should be continued (like adding `...` in MATLAB).

There should be one main program, `program program_name` where you can choose `program_name` (the same rules as for variable names in MATLAB). You end the main program with `end` or `end program program_name` . There are (essentially) two types of procedures, functions and subroutines (correspond to void functions in C). Fortran has an implicit type rule. Unless you specifically give a type of a variable, all variables are single precision real unless the variable name starts with one of `i, j, k, l, m, n` in which case the variable is an integer. Using the implicit rule is asking for problems, like in the following example: (using `%` as the prompt in the command window):

```
% cat spell.f90              list the program
program spell
  sum = 0.0             ! set summation variable to zero
  do k = 1, 1000        ! when k = 1, 2, ..., 1000
    sum = sum + 1.0 / k  ! update sum
  end do
  print*, 'the sum is ', smu  ! print the value
end program spell


% gfortran spell.f90         compile
% a.out                      execute
 the sum is   8.6880505E-44  garbage
```

Note that the compiler, `gfortran`, does not warn us (compilers usually do not). If we want a warning we add the statement `implicit none`, after the first line, making it necessary to supply type declarations, so

```
program spell
```

```
  implicit none    ! no implicit type rule
  real    :: sum   ! a single precision variable
  integer :: k     ! an integer variable
...

% gfortran spell.f90
 In file spell.f90:10
  print*, 'the sum is ', smu  ! print the value
                           1
Error: Symbol 'smu' at (1) has no IMPLICIT type
```

Correcting the error and running again we get the printout

```
 the sum is    7.485478
```

In Fortran call by reference is used for both arrays and scalars (it is possible to say much more about this topic, but this is sufficient for the HPC-course). This means that a procedure can change the value of a scalar argument, which is not the case in C unless you pass the address of the variable to the function.

19 A simple example

The code contains one main-program one function and a subroutine. The function computes the inner product of two vectors and the subroutine sums the elements in an array and returns the sum in a parameter. The line-numbers have been added to make it easier to comment the code, they are not part of the program.

```
 1  program main
 2    implicit none                ! Highly recommended!
 3    integer            :: k, n, in
 4    double precision   :: s
 5    double precision   :: ddot    ! the type of the ddot-function
 6
 7  ! Array indices start at one by default.
 8    double precision, dimension(100) :: a, b
 9
10    n = 100
11    print*, "Type a value for in:"
12    read*,  in
13    print*, "This is how you write: in = ", in
14
15    do k = 1, n              ! do when k = 1, 2, ..., n
16      a(k) = k               ! use do k = 1, n, step for a non-unit step
17      b(k) = -sin(dble(k))   ! using sin
18    end do
19
20    print*, "The inner product is ", ddot(a, b, n)
21
22    call sum_array(a, s, n)     ! NOTE, call
23    print*, "The sum of the array is ", s
24
25  end program main
```

Additional comments:
line 5, tells the calling program the type of the function. It is possible, and sometimes necessary, to provide more elaborate prototypes, but this is not covered in this course. Single precision is written `real` in Fortran.

line 8: You can write the index-range `dimension(begin:end)`, e,g, `dimension(-20:35)` or `dimension(0:20)` if you are a C-programmer (in which case 20 is the last and legal index).
line 11-13: simple I/O-statements with a standard layout (there are more fancy versions).
line 15, 18: the standard loop. More generally, `do variable = start, end, step`, where `step = 1` is the default.
line 16: note `( )` for the index.
line 17: the type cast (type conversion) from integer `k` to double precision `dble(k)` is necessary, or the compiler will complain. No automatic conversion like in C.
line 20: calling the function `ddot`.
line 22: calling the subroutine, note that the result is returned in the scalar `s`.

```
 1 function ddot(x, y, n) result(s)
 2   implicit none
 3   integer :: n
 4   double precision, dimension(n) :: x, y
 5   double precision :: s   ! The type of the function
 6
 7   integer          :: k
 8
 9   s = 0.0
10   do k = 1, n
11     s = s + x(k) * y(k)
12   end do
13
14 end function ddot
```

line 1, `result` defines a result-variable. You give the function a value by assigning a value to the result-variable, much like in MATLAB.
line 5, you give the function its type using a type declaration for the result-variable.
line 7, `k` is local to the function (as usual).
line 9, `0.0` is actually a single precision zero, which is converted to a double precision zero (the type of `s`). More about constants later on.

```
 1 subroutine sum_array(a, s, n)
 2   implicit none
 3   integer           :: n
 4   double precision  :: s
 5   double precision, dimension(n) :: a
 6
 7   integer :: k
 8
 9   s = 0.0
10   do k = 1, n
11     s = s + a(k)
12   end do
13
14 end subroutine sum_array
```

line 1, 9, 11: since call by reference is used, the subroutine changes the original value of `s`.

Some additional comments.
Since Fortran 90 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 functions.

20 Numerical constants

1 is an integer constant. Fortran (like C) performs integer division using truncation. `4/3` becomes `1` and `-4/3` becomes `-1`.
`1.0` is a real constant (single precision) and `1.0d0` is a double precision constant in Fortran77. The `d0` stands for zero exponent and `d` for double precision, so $1.0 \cdot 10^0$.
`-1.23e-14` is the single precision value $-1.23 \cdot 10^{-14}$ and `-1.23d-14` the double precision value.
Fortran has built-in support for complex arithmetic. See a Fortran-book or tutorial for details.
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 of kind (type) prec
  y = 1.24e-4_prec !
...
```

20.1 Warning: do not mix single and double precision

You can mix single and double precision, but the outcome may be unexpected. Here are a few examples.

```
program warning
  implicit none
  real             :: sp  ! a 32-bit floating point variable
  double precision :: dp  ! a 64-bit floating point variable

  sp = sqrt(2.0d0) ! you lose digits
  dp = sqrt(2.0)   ! you lose digits

  ! the abs-values are the errors
  print*, sp, dp, abs(sp - sqrt(2.0d0)), abs(dp - sqrt(2.0d0))

  print*, sqrt(2.0d0) * dp    ! dp is already ruined

```

```
14  ! (2.0 / 3.0) is computed in single, you lose digits
15    print*, (2.0 / 3.0) * 3.0d0, 2.0 / 3.0 * 3.0d0
16
17  ! 3.0 is first converted to 3.0d0, then 2.0d0 / 3.0d0 is computed
18    print*, 2.0d0 / 3.0 * 3.0d0
19
20  ! integer division
21    print*, 4 / 3 * 3.0d0
22
23  end program warning
```

```
Here is the output:
  1.414214  1.41421353816986  2.420323430563087E-008  2.420323430563087E-008
  1.99999996577146
  2.00000005960464        2.00000005960464
  2.00000000000000
  3.00000000000000
```

SP = single precision and DP = double precision in these comments.
line 3, 4, `sp` is 32-bit SP-variable and `dp` 64-bit DP-variable.
line 6, `sqrt(2.0d0)` is computed in DP but then assigned to single.
line 7, `sqrt(2.0)` is computed in SP but then assigned to a DP-variable.
line 10, writing the values and errors (regarding `sqrt(2.0d0)` as exact).
line 12, using the accurate `sqrt(2.0d0)` but the SP-value stored in `dp`. The product is *not* `2.0`.
line 15, due to the ( ), `(2.0 / 3.0)` is computed in single, this SP-quotient is then converted to DP (but *not* re-computed) and the multiplied by the DP `3.0d0`. The same happens with second expression, since it is computed left to right (`/` and `*` have the same priority).
line 18, the “weaker” (smaller) `3.0` is first converted to the “stronger” `3.0d0`, giving `2.0d0 / 3.0d0`, giving an accurate answer.
line 21, `4 / 3` becomes `1` which is promoted to `1.0d0` which is multiplied by `3.0d0`, so the answer is *not* `4.0`.

If you want to be on the safe side never mix precisions.

21 The simple example in Fortran 77

Here comes the Fortran 77-version, but first some comments. Fortran 90 is almost a new language, but in my simple example the differences are not that striking.

- Fortran 77 has a column oriented layout dating back to the 80 column punched card. The first five columns are used for labels (positive integers), column six is for continuation statements, columns 7-72 for the statement and columns 73-80 are for comments (used to number the cards in case you dropped them :-)
  If you are young and have not heard about punched cards, have a look here:
  `http://en.wikipedia.org/wiki/Punch_cards` .

- There are no result-statements in functions.

- The type declarations are written in a different way:

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

  instead of

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

although FORTRAN 77-declarations are allowed in Fortran 90 as well. A FORTRAN 77-program is (essentially) also a Fortran 90-program, so it is possible to mix the two styles. There are a few FORTRAN 77-constructions which have been deprecated in Fortran 90, and some Fortran 90-compilers may complain if you compile old code.

The example program, coded in FORTRAN 77, 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 are proper ways of writing FORTRAN 77-loops using labels (the numbers). `continue` is an empty statement often used to mark the end of a loop.

```
      do 10 k = 1, n
        s = s + x(k) * y(k)
10    continue
c
c     or shorter
c
      do 20 k = 1, n
20      s = s + x(k) * y(k)
```

Here comes the code:

```
      program main
*
*     Comments:           c, C or * in column one
*                         text in columns > 72
*     !                   Fortran 90-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  to know where we are
      integer           k, n, in
      double precision  a(100), b(100), sum
      double precision  ddot

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

      do k = 1, n
        a(k) = k
        b(k) = -sin(dble(k))
      end do

      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
```

```
c double precision is the type of the function
      double precision function ddot(x, y, n)
      implicit none
      integer           n
      double precision  x(n), y(n)

      integer           k
      double precision  sum

      sum = 0.0  ! could use ddot instead of sum
      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
      do k = 1, n
        sum = sum + a(k)
      end do

      end
```

Suppose you, by mistake, move the `double precision`-statement in `sum_array` one step to the left. The letter `d`, in the sixth column, tells the compiler that ther previous line (`integer n`) continues, essentially giving (spaces are not significant in variables names in FORTRAN 77), the line:

```
      integer ndoubleprecisiona(n), sum
```

resulting in complaints, from the compiler, about `n` and `a` lacking a type declaration. Removing the `implicit none` silences the complaints, the compiler will assume that `a(k)` is a call of the function `a`.

Removing one more space will give another compiler error, a non-digint in the label field.
Writing a long line, like:

```
c                                                                        comments
c23456789012345678901234567890123456789012345678901234567890123456789012
      s = s + long and complicated expression                     + 2.1d72
```

will add `2.1d7` and not the intended `2.1d72`.

Here is a final curiosity, which according to legend caused the crash of Mariner 1 spacecraft, but legend was wrong and Fortran was not to blame in this case, see: `http://en.wikipedia.org/wiki/Mariner_1`
and `http://catless.ncl.ac.uk/Risks/9.54.html#subj1.1`
But FORTRAN 66 *could* have caused a problem, as the Risks-article says. Consider the following **legal** piece of code (legal in FORTRAN 66 and 77, that is, not legal in Fortran 90):

```
      do 5 k = 1. 125
        s = s + vec(k)
5     continue
```

This is not a loop but an assignment, the variable `do5k` is assigned the value `1.125`. The dot should have been a comma. There was no implicit-statement in FORTRAN 66 (it came in FORTRAN 77), but a good compiler of its day could list variables occurring once.
Be very careful when programming in FORTRAN 77.

22 How to compile

The Fortran compilers available on the student system are: `g77` (FORTRAN 77) and `gfortran` (both Fortran 90 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 the course homepage for details.

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

```
%                       the prompt in the shell
% gfortran prog.f90     if everything in one prog.f90, prog.f would be Fortran77
```

Produces the executable file a.out

```
% a.out            execute
% ./a.out          if you don't have . in your path
```

Suppose we have three files `main.f90`, `dot.f90` and `sum.f90`.

```
% gfortran main.f90 dot.f90 sum.f90
```

Can compile the files one by one. `-c` means “compile only”, do not link.

```
% gfortran -c main.f90          produces the object file main.o
% gfortran -c dot.f90           produces the object file dot.o
% gfortran -c sum.f90           produces the object file sum.o
% gfortran main.o dot.o sum.o   link the object files

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

The last version is useful if you are working with and re-compiling a few routines and linking with existing object files, containing a large part of the code.
One can give many options (or flags) to the compiler. Note that the names are not standardized between compilers.

```
% gfortran -O3 prog.f90  optimize the code
```

23 If-statements and logical expressions

```
      double precision :: a, b, c, d
      logical          :: q          ! Fortran has a logical type

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

You cannot use `0` and `1` to denote false and true, as you can in C. Instead you have the logical constants, `.false.` and `.true.` (yes, the dots should be there).

Operation	FORTRAN 77	Fortran 90
$<$	`.lt.`	`<`
$\leq$	`.le.`	`<=`
$=$	`.eq.`	`==`
$\neq$	`.ne.`	`/=`
$\geq$	`.ge.`	`>=`
$>$	`.gt.`	`>`
and	`.and.`	`.and.`
or	`.or.`	`.or.`
not	`.not.`	`.not.`

24 A small Fortran 90-example

Here is a tiny example showing some of the array capabilities of Fortran 90.

```
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                       ! like in Matlab

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

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

  print*, 'minval(c) = ', minval(c) ! a built-in functions
  a = a + b * c                     ! elementwise *
  print*, 'a = ', a
  print*, 'sum(a) = ', sum(a)       ! another built-in
  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
 ! a = text format, i2 integer with width of 2
  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(a) =  -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
```

25 A common Fortran construction

Even if we program in Fortran 90 we typically use huge amounts of FORTRAN 77-code. This makes it necessary to understand something about the layout of matrices in memory and how the compiler computes the address of a specific element, the address computation. If you get this wrong disaster will follow.

FORTRAN 77 does not have dynamic memory allocation (like Fortran 90 and C). Say you want to solve a sequence of linear least squares problems of different sizes. In FORTRAN 77 you would typically reserve space for the largest matrix you need, even though the actual problem might be smaller. Say that the largest problem has `max_m` rows and `max_n` columns and that actual (current) problem has `m` rows and `n` columns (`m` $\le$ `max_m` and `n` $\le$ `max_n`).
Say you pass the matrix as an argument to a procedure. The compiler (when compiling the procedure) must be told about the extent of the first dimension (the number of rows), of the matrix, in order to produce the address computation code.
The reason for rows rather columns, is that Fortran stores matrices in column-major order (column after column). If `adr( )` is the address of `A(j, k)` then

```
      adr(A(j, k)) = 8 * (adr(A(1, 1)) + max_m * (k - 1) + j - 1)
```

eight, since we assume that memory is byte-addressable and that `A` is a double precision matrix (eight bytes per floating point number). In C a matrix is stored in row-major order (so the compiler must know the number of columns in the matrix), but since you can allocate the precise number of elements in C this is less of an issue.
In the following program we reserve space for a 5×4-matrix, line 4, and all the elements are set to 0 (so we can spot incorrect elements later on). `parameter` on line 3, defines *named constants*, so `max_m` and `max_n` are not variables but names for the numbers 5 and 4 respectively. We are only using a 3×3-sub matrix of `A`, and on lines 10-12 this matrix is set to `[1, 2, 3; 4, 5, 6; 7, 8, 9]` (using MATLAB-syntax). On line 15 `print_A` is called, the subroutine prints the 3×3-sub matrix. On line 19 `print_A` is called again, but this time using `m` instead of the correct `max_m`.

The crucial line in `print_A` is line 26, where it says `max_m` in `dimension(max_m, max_n)`. `max_n` is of no importance (at least not for the address computation). Line 30 is a more elaborate print-statement using a format.

```
 1  program wrong_max_m
 2    implicit none
 3    integer, parameter :: max_m = 5, max_n = 4
 4    integer, dimension(max_m, max_n) :: A = 0
 5    integer :: m, n
 6
 7    m = 3 ! using part of the max_m times max_n-matrix
 8    n = 3
 9
10    A(1, 1:n) = (/ 1, 2, 3 /)
11    A(2, 1:n) = (/ 4, 5, 6 /)
12    A(3, 1:n) = (/ 7, 8, 9 /)
13
14    print*, 'Calling print_A with the correct max_m'
15    call print_A(A, max_m, max_n, m, n)
16
17    print*, '------------------------------------'
18    print*, 'Calling print_A with an incorrect max_m'
19    call print_A(A, m, max_n, m, n)  ! using m instead of max_m
20
21  end program wrong_max_m
22
23  subroutine print_A(A, max_m, max_n, m, n)
24    implicit none
25    integer :: max_m, max_n, m, n
26    integer, dimension(max_m, max_n) :: A
27    integer :: row, col
28
29    do row = 1, m
30      write(*, '(a3, i2, a, 5i5)') 'row ', row, ':', A(row, 1:n)
31    end do
32
33  end subroutine print_A
```

Here is the run:

```
 Calling print_A with the correct max_m
row 1:    1    2    3
row 2:    4    5    6
row 3:    7    8    9
 ------------------------------------
 Calling print_A with an incorrect max_m
row 1:    1    0    5
row 2:    4    0    8
row 3:    7    2    0
```

To see how the second result is produced we look at the memory layout (horizontally, to save space). `|` denotes a column-break.

```
In the main-program
```

```
after line 4
 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0
after line 12
 1 4 7 0 0 | 2 5 8 0 0 | 3 6 9 0 0 | 0 0 0 0 0

in print_A with the incorrect max_m = 3
 1 4 7 | 0 0 2 | 5 8 0 | 0 3 6 | 9 0 0 | 0 0 0 | 0 0
```

If we had not set all the elements in A to zero, the zeros in the second printout could have been random numbers (what happened to be stored in those memory locations). Removing the initialization on line 4 and running again, one may get things like (where I had to change the format to make room for the large negative number):

```
 Calling print_A with an incorrect max_m
row 1:            1           62            5
row 2:            4            1            8
row 3:            7            2   -1264574544
```

It is not necessary for the compiler to know max_n when compiling print_A (unless A is a three dimensional array, of course), so it is legal to write

```
  double precision, dimension(max_m, *) ::  A
or
  double precision, dimension(max_m, 1) ::  A
```

just to tell the compiler that A is two-dimensional (the * marks an index position). Better is:

```
  double precision, dimension(max_m, max_n) :: A
```

since index checks can be performed by some compilers.

A more common name for max_m is LDA, Leading Dimension A. This can be seen in the manual page for the Fortran 77 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,*)     <---------- NOTE LDA
 ...
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.     <---------- NOTE LDA
```

```
          LDA >= max(1,N).
...
```

There is are nicer interface in Fortran90 (C++). Essentially, `subroutine gesv( A, B, ipiv, info )` where `gesv` is polymorphic, (for the four types `S`, `D`, `C`, `Z` for single, double, complex and double complex) and where the size information is included in the matrices. Most people seem to use the FORTRAN 77-interface, however, and it is easier to use from C.

26 Dynamic memory allocation in Fortran 90

Here are two ways, first automatic arrays:

```
...
call dynamic(k + r * s, 100) ! for example
...
subroutine dynamic(m, n)
integer                           :: m, n
double precision, dimension(m, n) :: A
double precision, dimension(-2:n) :: vec ! first index -2, just to show you can

compute ...

end subroutine dynamic
```

The second method is similar to C's `malloc/free`.

```
subroutine dynamic(m, n)
integer                                        :: m, n
double precision, allocatable, dimension(:, :) :: A
double precision, allocatable, dimension(:)    :: vec
integer :: status

allocate(vec(-2:n))  ! first index -2, just to show you can
allocate(A(m, n), stat = status) ! if you are careful
if ( status /= 0 ) then
  ! some problem, this is very primitive error handling
  print*, 'Cannot allocate A'
  stop
end if

compute ...

deallocate(vec)
deallocate(A)

end subroutine dynamic
```

27 Some dangerous things

When debugging code it is very important to check actual and formal parameter lists. Actual parameters are the ones supplied when calling the routine, formal parameters are the ones inside the routine.

Check position, number and type. It is possible to use so-called interface blocks ("prototypes") for increased security.

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

  a = 0.0
  call sub(a, 1.0, b) ! three parameters
  print*, a, b
end
subroutine sub(i, j)  ! two parameters and different types in the call
  integer :: i, j

  i = i + 1
  j = 10.0            ! trying the change the value of the constant 1.0
end

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

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

```
% a.out      the result depends on the compiler
       4.940656458412465E-324  1.330526861551857E-312
```

`b` is undefined and the contents of `a` is treated as an integer inside the subroutine. Since Fortran uses call by reference, the interpretation of the data, corresponding to the formal parameter `a`, inside the subroutine is given by the type declaration, `integer :: i`.
C- and Fortran compilers do not usually check array bounds. Here is an example in C.

```
#include <stdio.h>
void sub(double a[]);

int main()
{
        double          b[10], a[10];

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

        return 0;
}

void sub(double a[])
{
        a[10] = 12345.0;
}
```

Running this program we get:

```
% a.out
12345.000000
```

So `b[0]` has changed even though it is not a parameter to the function, `sub`. The reason is that `a[10] = 12345.0;` is illegal, nine is the largest index and `a[10]` happens to have the same address as `b[0]`. Changing `a[10]` to `a[1000000]`, in the function, 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

% gfortran -fbounds-check lie.f90
% a.out
Fortran runtime error: Array reference out of bounds for array 'a', upper bound of dimension 1
exceeded (in file 'lie.f90', at line 12)
```

If we change `dimension(10)` to `dimension(100)`, in the subroutine, so lying, the compiler will not detect the index error.

28 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.

29 A few words about `tcsh` and `bash`

The shell is a command-line interpreter, usually running as a sub-process to a command window. When you, for example, give the command `cd a_path`, the shell will change your current directory, and when you type `ls`, the shell will create a new process, starting the compiled C-program `/bin/ls`. There are a number of shells, two wellknown are the Bourne shell, `/bin/sh`, written by Stephen Bourne at Bell Labs and `csh`, `/bin/csh`, written by Bill Joy while at the University of California, Berkeley. See `http://en.wikipedia.org/wiki/Unix_shell` for more historical notes.
On the math-system `/bin/sh` is a symbolic link to `/bin/bash` the Bourne-Again shell and `/bin/csh` is a symbolic link to `/bin/tcsh`, the TENEX C shell.
You can change shell if you like, I am using `tcsh`, and I will start with some aspects of `tcsh`.

29.1 The 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 my `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`, a few words about `bash` come at the end of this section. 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 64-bit LSB executable, AMD x86-64, version 1 (SYSV), for GNU/Linux 2.6.9,
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, 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).

29.2 Now something about `bash`

Most of the above details about `tcsh` work in `bash` as well. Here are some differences. The shell startup file is called `.bashrc`. The path-variable is named `PATH`. You can set (a short path) the following way (you do not use `set` as in `tcsh`):

```
% PATH=/bin:/usr/bin
```

To export a variable to a sub-process, use the `export`-command, like in this example:

```
bash-3.2$ A_VARIABLE=123         bash-3.2$ is the prompt
bash-3.2$ echo $A_VARIABLE
123
bash-3.2$ bash                   start a sub-shell
bash-3.2$ echo $A_VARIABLE
                                 not defined

bash-3.2$ export A_VARIABLE=123  use export
bash-3.2$ bash                   start a sub-shell
bash-3.2$ echo $A_VARIABLE
123                              defined
```

`set` prints all the variables, but there is no `setenv`-command, use `export` instead.

For **much** more on `tcsh` and `bash` try

```
  man tcsh
  man bash
```

or

```
  info tcsh
  info bash
```

for a more structured layout.

29.3 A note on the student environment

To make it easier for beginners (both teachers and students) Chalmers/GU has a standard environment where you do not have to create your own startup files. One does not have to use it (I do not). The following page describes how to modify the standard environment:
`http://www.chalmers.se/its/EN/computer-workplace/linux/various-linux-questions`