

## 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  $s = s + \text{term}$  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.

For more history and background see the Wikipedia article:  
[http://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](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\\_\(book\)](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.

### 1.1 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. 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 or tcsh), using the return-statement, Zero usually means OK. If we write echo \$status, in the shell, we can check the return value.

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 .

## 1.2 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:

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

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 lines 8, 9 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 some compilers allow for C++-comments as well (lines starting with `//`).

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.

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  $1 + 1/2 + 1/3 + \dots + 1/1000$ .

```
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, `ld` the linker, takes care about this part (more about `ld` later on). The math library resides in a file, `/usr/lib/libm.so`. 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.
```

### 1.3 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/cc4JXzK.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.

## 1.4 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 be 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.

## 1.5 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  $\sum_{k=1}^n 1/k^2$ .

```
1 #include <stdio.h>
2
3 void sums(double *, double *, int);
4
5 int main()
6 {
7     double sum1, sum2;
8
9     sums(&sum1, &sum2, 1000);
10    printf("The sums are: %e and %e\n", sum1, sum2);
11
12    return 0;
13 }
14
15 void sums(double *a_sum1, double *a_sum2, int n)
16 {
17     int k;
18
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 division */
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 it 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.

## 1.6 Arrays

In this program we create a one-dimensional array (vector) containing ten elements. We call the function `init` to initialize the elements to  $1, 2, \dots, 10$ . 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

```
double 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.

## 1.7 Two-dimensional arrays

Here is one small example where we multiply two  $4 \times 4$ -matrices together (order four, since we are going to work with such matrices in the OpenGL part).

```
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 after row, in memory. So the memory layout, of the matrix C, for example, would be:

```
address
base      C[0][0]
base + 1  C[0][1]
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.

## 1.8 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 copy of your original file. In my example the copy is stored in `my_program.c~`.

## 1.9 If-statements and logical expressions

Here are a few examples. Note single `&` and `|` are **bitwise** operations.

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

if ( a < b && c == d || !q ) {
    ... 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. What `if` does the `else` belong to, if you have nested if-statements? One the next page comes an example:

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a[] = { 0, 0, 1, 1 }, b[] = {
6          0, 1, 0, 1}, k;
7
8      for (k = 0; k < 4; k++) {
9          printf("----- k = %d\n", k);
10         if (a[k])
11             if (b[k])
12                 printf("1: then\n");
13         else                               /* incorrect indentation */
14             printf("1: else\n");
15
16         /* Correct interpretation */
17         if (a[k]) {
18             if (b[k])
19                 printf("2: then\n");
20             else
21                 printf("2: else\n");
22         }
23
24         if (a[k]) {
25             if (b[k])
26                 printf("3: then\n");
27         } else
28             printf("3: else\n");
29
30     }                                     /* end for */
31
32     return 0;
33 }
    
```

When we have nested if-statements, the `else` belongs to the innermost if, so lines 10-14 and lines 17-22 are equivalent. Note the **incorrect** indentation on line 14. If you want the `else` to belong the outer if, use braces as in lines 24-28.

```

% a.out
----- k = 0
3: else
----- k = 1
3: else
----- k = 2
1: else
2: else
----- k = 3
1: then
2: then
3: then
    
```

Lines 5, 6 show new syntax. `a` and `b` will be integer arrays with four elements. The next page contains a precedence table for the operators in C, but we have not looked at all of them in this introduction.



### 1.10 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 |
|---|---|---------------|
| ( )<br>[ ]<br>-><br>.<br>++<br>-                              | function call<br>vector index<br>structure pointer<br>structure member<br>postfix increment<br>postfix decrement  | →             |
| !<br>~<br>++<br>--<br>+<br>-<br>*<br>&<br>(type)<br>sizeof    | logical negation<br>bitwise negation<br>prefix increment<br>prefix decrement<br>unary addition<br>unary subtraction<br>indirection<br>address<br>type cast<br>number of bytes   | ←             |
| *<br>/<br>%   | multiplication<br>division<br>modulus   | →             |
| +<br>-  | binary addition<br>binary subtraction   | →             |
| <<br>>  | left shift<br>right shift   | →             |
| <<br><=<br>><br>>=  | less than<br>less or equal<br>greater than<br>greater or equal  | →             |
| ==<br>!=  | equality<br>inequality  | →             |
| &<br>^<br>  | bitwise and<br>bitwise xor<br>bitwise or  | →             |
| &&<br>  | logical and<br>logical or   | →             |
| ?:  | conditional expression  | ←             |
| =<br>+=<br>-=<br>*=<br>/=<br>%=<br>&=<br>^=<br> =<br><=<br>>= | assignment<br>combined assignment and addition<br>combined assignment and subtraction<br>combined assignment and multiplication<br>combined assignment and division<br>combined assignment and modulus<br>combined assignment and bitwise and<br>combined assignment and bitwise xor<br>combined assignment and bitwise or<br>combined assignment and left shift<br>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 ( $\rightarrow$ ) 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.