excerpt from the book

Practical Guide To Computer Simulations

World Scientific 2009, ISBN 978-981-283-415-7

see http://www.worldscibooks.com/physics/6988.html

with permission by World Scientific Publishing Co. Pte. Ltd.

Alexander K. Hartmann
Institute of Physics
University of Oldenburg
Germany

October 22, 2009

Chapter 1

Programming in C

Performing computer simulation in a sophisticated way always includes writing computer programs. Even if one builds upon an existing package, one usually has to perform minor or major modifications or write new subroutines for data analysis.

Several programming languages are commonly used in science. The historically first widespread high-level procedural programming language for this purpose is *Fortran*, which was developed in the 1950s. Thirty years ago, Fortran 77 was a standard in numerical computations, which has led to the development of many Fortran-based libraries like the NAG (*Numerical Algorithms Group* [Philipps (1987)] library. Since Fortran used to be a very inflexible programming language, other high-level structural programming languages like Pascal (and its extensions Modula-2 and Delphi) and C (and its object-oriented extension C++) were developed from the 1970s on. These languages allow for complex data types, dynamic memory allocation and modularization, which makes the treatment of large-scale simulations much easier. During the 1980s, the C programming language became a standard in particular for operating system programming. Hence, standard operating systems like UNIX (Linux) and Windows are written in C. This resulted in C being used in many other fields as well, in particular for scientific purposes. As a result, all standard numerical libraries are now available for C/C++ and nowadays C is the dominating language for writing simulation programs in science.

Note that Fortran caught up via the Fortran 90 and Fortran 95 standards, capable of basically everything C can do. Thus, Fortran is still heavily used, in particular because it allows for easier parallelization of codes related to linear and differential equations. Nevertheless, in this book we have to select one language to concentrate on. We select C (and C++ in Chap. 3) because it offers, on the one hand, everything you can expect from a high-level programming language. On the other hand, C is relatively close to machine-level programming, which leads to a very high performance, but also makes insecure programming possible. Also, more recent developments like Java have a similar syntax to C and C++. C is also widespread outside the scientific community in business-

related areas, a field where many people reading this book will work in after their studies. Therefore, a scientist benefits much from knowing C very well. In general, all high-level programming languages are equally powerful. Therefore, good proficiency in C allows to learn another programming language very quickly.

There are many additional programming languages used for specific purposes such as writing scripts (traditional shell script languages, Perl) or languages used within certain statistical and/or mathematical packages like *Mathematica*, *Maple*, *Matlab*, or *R*. If you want to learn these, you have to consult special literature.

Here, we start by explaining how to write simple C programs and what the most fundamental program ingredients are. This also includes advanced techniques like structures and self-defined data types. Note that we use a standardized form called *ANSI C*. Structuring programs is easier when using subroutines and functions, which are covered in the second section. Next, details for input-/output techniques are presented. In the fourth section of this chapter, more details concerning pointers and memory allocation are explained. In the last section, some examples for macro programming are given. C++ extensions related to object-oriented programming are covered in Chap. 3.

Before starting, you should be advised that learning to program is a matter of practice. Consequently, it will not be sufficient to just read the following sections. Instead, you should sit in front of a computer and try, use, and modify all given examples extensively!

As always in this book, most concepts are introduced via examples. Most standard statements and control structures are covered, but not explained in the most-general way. Nevertheless, it should be sufficient in 95% of all cases. For an exhaustive and complete descrition of C, you should consult specialized literature [Kernighan and Ritchie (1988)].

1.1 Basic C programs

It is assumed that you are familiar with your operating system, in particular with concepts like commands, files and directories, and that your are able to use a text editor. All examples given here are based on UNIX shell commands, but should be simple to convert to your favorite operating system. As always in this book, the basic functionalities are explained, such that fundamental applications are covered. For a complete reference to C programming, you should consult specialised books on this topic [Kernighan and Ritchie (1988)]. Note that the standard (so-called ANSI) C contains a complete set of commands such that programs for all tasks can be implemented. Nevertheless, for many standard applications, e.g. computing complex functions, solving equations, calculating integrals, or performing algorithms on complex data structures like graphs, there are extensions of the standard C, collected in so-called *libraries*. More about libraries and how to use them can be found in Chap. 6.

GET SOURCE CODE
DIR: `c-programming` FILE(S): `first.c`

In this chapter, we start the introduction to C by presenting a very first C program, which you should enter in your favorite text editor. There you should save the file with the file name, say, `first.c`. The program looks as follows:

```
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("My first program\n");
6   return(0);
7 }
```

The meaning of the different lines will be explained in a minute. First, you should *compile* the program. This means that in the current form the program cannot be executed by the computer. The compiler converts it into so-called *machine code*, which can be executed. Usually, the machine code resulting from your program will be put together with other already existing machine codes, e.g. for some standard tasks like input/output subroutines. This is the so-called *linking* state of the compilation performed by the *linker*

Under UNIX/Linux, you can compile the program in a shell, provided the current directory of the shell is in the directory where the program is stored, via the command

```
cc -o first -Wall first.c
```

The main command is `cc`, which is an abbrevation of *C compiler*. In the above example, two *options* are used for the command `cc`:

`-o` (for ouput) gives the name the executable program will get, here `first`.

`-Wall` The `W` stands for *warning* and `all` means the compiler will report (almost) *all* strange things it observes, even if it is not an error in a strict sense. Since C is very sloppy about many rules, it lets you get away with a lot, which most of the times is not what you want to do.

The classical example is the comparison operator `==`, i.e. you have to write `a==b`, if you want to test whether the values stored in the variables `a` and `b` are equal. This is different from `a=b`, which assigns `a` the value of `b` (see below). Nevertheless, for conditional statements like `if(a==b)`, sometimes the programmer mistakingly writes `if(a=b)`, which is nevertheless a meaningful statement under some circumstances. Therefore, the compiler does not report this as an error, but with `-Wall`, the compiler warns you that you have used an assignment where a conditional statement is expected.

Hence, you should use the highest warning level `-Wall` always, and ignore the warnings only, if you know exactly what you are doing. Note that

there are several other warning options in C which are not turned on even when using `-Wall`, despite its name. E.g. the option `-Wextra` (previously called just `-W`) turns on additional warnings. A complete list of all warning options can be found in the compiler documentation.

The last argument of the `cc` comand is the file you actually want to compile. Note that, in principle, a program may consist of several files, which can all be passed via arguments to the compiler. By default, the C compiler will generate an executable. Consequently, the compiling process will consist of the actual compiling of the input source files, which generates a (hidden) *assembler code* for each input file. The assembler codes are translated to executable codes. Finally the different executable codes are *linked* into one program. If the option `-c` is given, the linking stage will be omitted. In this case several exetutable `.o` files are created. The C compiler, when called from the command line, offers a huge number of options, which are all described in the manual pages. Under UNIX/Linux, you can access the manual via the `man` command, i.e. by entering "`man cc`".

Now we come back to the structure of the program `first.c`. The first line *includes* a file. This means that the file `stdio.h` contains some definitions which the compiler should read before proceeding with the compilition process. In this case, `stdio.h` contains declarations of some built in standard functions for input/output i/o. The compiler needs these declarations, because it has to know, for example, how many parameters are passed to the function. The files containing declarations are usually called *header files*, hence the suffix is `.h`. In this example, the compiler needs the defintion of the subroutine `printf()`, which means *print formatted* (see below). Note that the acutal *definitions* of the subroutines, i.e. the code for the subroutine, is not contained in the header files, but in *libraries* (see page 16) and Sec. 6.1.

The main program of every C program is called `main()`, which comprises the remaining file of `first.c`. In fact, all codes which do something in C programs are functions, i.e. they take some arguments and return something. Hence, `main()` is also a function. Here, it takes zero arguments, indicated by the fact that nothing is written in the brackets `()` after the name (`main`) of the function in line three. Below, you will learn how to pass to `main()` some arguments via the command line. By default, the function `main()` returns an *integer* value, which is indicated by the `int` in front of the function name.

The definition of a function, i.e. the program code which is executed when the function is called, is contained inside the pair {...} of brackets (lines 4–7). Note that the opening and closing brackets are written below each other for readability, but this is not required by the compiler. You could have written everything in one line, or insert many empty lines, etc. Also, the position inside a line does not matter at all, e.g. your program could look like

```
#include <stdio.h>
int
main
(){
  printf(
"My first program\n");            return(0);
                  }
```

Certainly, the first version is much more readable than the second, so you should follow some formatting rules. This is discussed in Sec. 2.2. Note that concerning the `#include` *directive*, which is only a kind of command passed to the compiler but not a true C command, the syntax is more strict. Basically, each `#include` directive must be written in exactly one line (unless one explicitly uses several lines via the \ symbol at the end of a line). Also, only one directive is allowed per line. More information on the *compiler directives* and the related *macros* is given in Sec. 1.6.

The code describing the functionality of `main()` comprises the two lines 5–6 here. First, the `printf()` subroutine prints the string given as argument to the screen, i.e. here the string "`My first program`". The `\n` symbol is a formatting symbol. It indicates that after printing the string a *new line* is inserted. Consequently, any subsequent output, if also performed using `printf()` occurring during the execution of a program, will be printed starting at the next line. One can print many different things using `printf()`, more details are explained in Sec. 1.3. Note that formally the subroutine `printf()` is a *function*, since in C *all* subroutines are functions. Functions can have side effects, like printing out information or changing content of variables. Also, all functions calculate or *return* a value, which is often less important than the side effects, like in this case. The function call, together with the terminating semicolon, makes a *statement*. Each statement must close with a semicolon, even if it is the last statement of a function or the last statement of a block (see page 19) of statements.

In line 6, `return(0)` states the value the function returns. Here it is just 0, which means for `main()` by convention that everything was OK. In general, one can return arbitrary complex objects. Simple mathematical functions will return a real number, e.g. the square root of the argument, but also more complex objects like strings, arrays, lists or graphs are possible. More on functions in C is given in Sec. 1.2.

To actually execute the command, one types into the shell the name of the program, i.e.

```
first
```

Note that this assumes that the current directory is part of the search path contained in the environment variable `PATH`. If this is not the case, one has to type in "`./first`". As a result of the program execution, the message "`My first program`" will appear on the screen (inside the shell, just in the line after where you have called the program), the program will terminate and the shell will wait for your next command.

1.1.1 Basic data types

To perform non-trivial computations, a program must be able to store data. This can be done most conveniently using *variables*. In C, variables have to be *defined*, i.e. the *type* has to be given (see below on the syntax of a definition). A definition will also cause the compiler to reserve memory space for the variable, where its content can be stored. The type of a variable determines what kind of data can be stored in a variable and how many bytes of memory are needed to store the content of a variable. Here, we cover only the most important data types. The most fundamental type is

- `int`

  This type is used for integer numbers. Usually, four bytes are used to store integer numbers. Hence, they can be in the range $[-2^{31}, 2^{31}-1]$. Alternatively you can use `unsigned int`, which allows for numbers in the range $[0, 2^{32}-1]$

To define some variables of type `int`, one writes `int` followed by a comma-separated list of variable names and ended by a semicolon, like in

```
int num_rows, num_columns, num_entries;
```

which defines the three variables `num_rows`, `num_columns` and `num_entries`. Valid names of variables start with a letter {a...z,A,...,z} and may contain letters, digits $\{0,\ldots,9\}$, and the underscore '_' character.[1] It is not allowed to used reserved key words. Forbidden are particularly statement key words like `for` or `while` as well as predefined and self-defined data types.

Initially, the values of variables are undefined (some compilers automatically assign zero, but you cannot rely on this). To actually assign a number, the C statement should contain the variable name, followed by an equal character '`=`' followed e.g. by a numeric constant, and terminated by a semicolon like in

```
num_rows = 10;
num_colums = 30;
```

Note that the spaces around the '`=`' are not necessary, but allow for better readability. To the right of the '`=`' symbol, arbitrary arithmetic expressions are allowed, which may contain variables as well. If a variable appears in an expression, during the execution of the program the variable will be replaced by its current value. A valid arithmetic expression is

```
num_entries = num_rows * num_colums;
```

More on arithmetic expressions, including the introduction of the most common operators, will be presented in Sec. 1.1.2. Note that one can write numeric constants also in octal (number with leading `0`) or in hexadecimal (number with leading `0x`), e.g. the above statements could read as

[1] One can start variables also with the underscore '_' character, but this is also done by library subroutines, hence you should avoid an explicit use of the underscore.

```
num_rows = 012;
num_colums = 0x1e;
```

The value of one or several variables can be printed using the `printf` command used above. For this purpose, the *format string* passed to `printf` must contain *conversion specifications* for the variables, where the actual value will be inserted, when the string is printed. The variables (or any other arithmetic expression) have to be passed as additional arguments after the string, separated by commas. The most common conversion specification to print integers is "`%d`". The values of the variable can be printed in a C program for example with

```
printf("colums = %d, rows = %d, total size = %d\n",
       num_rows, num_colums, num_entries);
```

The '`\n`' at the end of the format string indicates again a *new line*. Using the above statements, the C program will print in this case

```
colums = 10, rows = 30, total size = 300
```

One can also assign variables right at the place where they are defined. This is called *initialization* and can be done e.g. using

```
int num_rows = 30;
```

This is even mandatory when defining a variable as being a *constant*. This means its content is not allowed to change. Syntactically, this is indicated by the qualifier `const` written in front of the type name, e.g.

```
const int num_rows = 30;
```

Hence, if you try to assign a value to `num_rows` somewhere else in the program, the compiler will report an error.

Other fundamental data types, which have similar fundamental properties and usage like `int`, include

- `short int`

  If you are sure that you do not need large numbers for specific variables, you can economize memory by using this data type. It consumes only two bytes, hence numbers in the range $[-2^{15}, 2^{15} - 1]$ are feasible, while for `unsigned short int`, numbers in the range $[0, 2^{16} - 1]$ can be used. Variables of these types are handled exactly like `int` variables.

- `char`

  This data type is used to store characters using the ASCII coding systems in one byte. Characters which are assigned have to be given in single quotes. When printing characters using `printf()`, the conversion specification `%c` can be used. The following three lines define a variable of type `char`, assign a character and print it.

```
char first_char;
first_char = 'a';
printf("The first character is %c.\n", first_char);
```

Note that you can also use `char` variables to store integers in the range $[-128, 127]$. Therefore, when you print the variable `first_char` using the `%d` conversion specification, you will get the ASCII code number of the contained character, e.g. the number 97 for the letter '`a`' and the number 10 for '`\n`'. The code numbers might be machine and/or operating system dependent.

Variables of the type `unsigned char` can store numbers in the range $[0, 255]$.

- `double` and `float`

  These are used to store floating-point numbers. Usually `float` needs four bytes, allowing for numbers up/down to $\pm 3.4 \times 10^{38}$, while `double` uses eight bytes allowing for much larger numbers ($\pm 9 \times 10^{307}$).

  When assigning `float` or `double` numbers, one can e.g. use the fixed point notation like in `-124.38` or a "scientific" notation in the form `-1.2438e2` which means -1.2438×10^2 (equivalently you could write `-12.438e1` or `-1243.8e-1`). Note that the numeric constant `4` is of type `int`, while the constant `4.0` is of type `double`. Nevertheless, you can assign `4` to a floating-point variable, because types are converted implicitly if necessary, see Sec. 1.1.2.

  When printing `float` or `double` numbers usually the conversion specifications `%f` for fixed point notation or `%e` for scientific notation are used.

  Note that standard C does not provide support for complex numbers. This is no problem, because there are many freely available *libraries* which include all necessary operations for complex computations, see Chap. 6.

- `long int` and `long double`

  On some computer systems, more memory is provided for these data types. In these cases, it makes sense to use these data types if one requires a larger range of allowed values or a higher precision. The use of variables of these data types is the same as for `int` or `double`, respectively.

- addresses, also called *pointers*

  Generic addresses in the main memory can be stored by variables defined for example in the following way

  ```
  void *address1, *address2;
  ```

  Note that each variable must be preceded by a '`*`' character. The type `void` stands for "anything" in this case, although sometimes it stands for

"nothing" (see Sec. 1.2). If one wants to specify that a pointer stores (or "points to") the address of a variable of a particular type, then one has to write the name of the data type instead, e.g.

```
int *address3, *address4;
```

One important difference from a `void` pointer is that `address3+1` refers to the memory address of `address3` plus the number of bytes needed to store an `int` value, while `address1+1` is the memory address of `address1` plus one byte.

The actual address of a variable is obtained using the `&` operator (see also next section), e.g. for an integer variable `sum` via

```
address1 = &sum;
```

GET SOURCE CODE
DIR: `c-programming` FILE(S): `address.c`

The `printf()` conversion specification for pointers is '`%p`', the result is shown in hexadecimal. E.g.

```
printf("address = %p\n", address1);
```

could result in

```
address = 0xbffa0604
```

The value stored at the memory location where `address1` points to is obtained by writing `*address1`. Consequently, the lines

```
int sum;
int *address1;

address1 = &sum;
sum = 12;
printf("%d \n", *address1);
```

will result in printing `12` when running.

A slightly advanced use of pointers can be found in exercise (1).

GET SOURCE CODE
DIR: `c-programming` FILE(S): `sizeof.c`

Note that you can determine the actual number of bytes used by any data type, also the self-defined ones (see Sec. 1.1.4), via the `size_of()` function within a C program. The name of the data type has to be passed to the function as argument. For example, the following small program `sizeof.c` will print the number of bytes used for each type:

```
#include <stdio.h>

int main()
{
  printf("int uses %d bytes\n", sizeof(int));
  printf("short int uses %d bytes\n", sizeof(short int));
  printf("char uses %d bytes\n", sizeof(char));
  printf("long int uses %d bytes\n", sizeof(long int));
  printf("double uses %d bytes\n", sizeof(double));
  printf("float uses %d bytes\n", sizeof(float));
  printf("long double uses %d bytes\n", sizeof(long double));
  printf("pointers use %d bytes\n", sizeof(void *));
  return(0);
}
```

Note that the format string of the `printf` statement, i.e. the first argument, contains a *conversion character* `%d`, which means that when printing the format string, the number passed as second argument will be printed as integer number in place of the conversion character. More details on the `printf` command and its format strings are explained in Sec. 1.3. Thus, the program may produce the following output:

```
int uses 4 bytes
short int uses 2 bytes
char uses 1 bytes
long int uses 4 bytes
double uses 8 bytes
float uses 4 bytes
long double uses 12 bytes
pointers use 4 bytes
```

Hence, the usage of `long int` does not increase the range compared to `int`, but so does `long double`. The `sizeof()` command will be in particular important when using dynamic memory management using pointers, see Sec. 1.4.

The precision of all standard C data types is fixed. Nevertheless, there are freely available libraries which allow for arbitrary precision including corresponding operators, see Chap. 6. The operators for the standard arithmetic data types are discussed in the next section.

1.1.2 Artithmetic expressions

Arithmetic expressions in C basically follow the same rules of traditional arithmetics. The standard operations addition (operator `+`), subtraction (`-`), multiplications (operator `*`), and division (`/`), as well as the usual precedence rules are defined for all numeric data types. Furthermore, it is possible to introduce explicit precedence by using brackets. The following expressions give some examples:

```
  total_charge = -elementary_charge*(num_electrons-num_protons);
  weighted_avg_score = (1.0*scoreA+2.0*(scoreB1+scoreB2))/num_tests;
```

When executing the program, the values which result in evaluating the expressions to the right of the '=' character are assigned to the variables shown to the left of the '=' character. Note that an assignment is an expression itself, whose result is the value which is assigned. Therefore, the result of the assignment `x = 1.0` is 1.0 and can be assigned itself like in

```
y = x = 1.0;
```

Nevertheless, such constructions are not used often and should usually be avoided for clarity.

Note that for integer variables and constants, the division operation results just in the quotient, being an integer again, e.g. `5/3` results in a value `1`. This is even the case if `5/3` is assigned to a variable of type `double`! The reminder of an integer division can be obtained using the modulus operator (`%`), e.g. `5%3` results in `2`. Note that the modulus operator is not allowed for one or two `double` or `float` operands. Nevertheless, in general, if a binary operator combines objects of different types, the value having lower resolution will be converted internally to the higher resolution. Consequently, the type of an expression will be always of the highest resolution. On the other hand, when assigning the result of an expression to a variable of lower resolution (or when passing parameters to functions as in Sec. 1.2), the result will also be automatically converted. Therefore, when assigning a floating-point result to an integer variable, the value will be truncated of any fractional part. One can also perform explicit type conversions by using a *cast*. This means one writes the desired type in `()` brackets left of a constant, variable or expression in brackets, e.g. in

```
void *addressA;
int *addressB;

addressB = (int *) addressA+1;
```

`addressB` will point 4 bytes ahead of `addressA`, while without the cast it would point only one byte behind `addressA`.

As pointed out in the last section, `*address` refers to the content of the variable where `address` points to. Hence, one can change the content of this variable also by using `*address` on the left side of an assignment, like in

```
int sum;
int *address1;

sum = 12;
address1 = &sum;
*address1 = -1;
printf("%d \n", sum);
```

which will result in printing the value `-1` when the program is executed. The value of `address1` will not be changed, but will still be the address of variable `sum`.

It occurs very often that a variable is modified relative to its current value. Hence, the variable occurs on the left side of an assignment as well as on the right side within an arithmetic expression. For the simplest cases, where the variable is incremented, decremented, multiplied with, or divided by some value, a special short syntax exists (as well as for the modulus operator and for the bitwise operations presented below). For example,

```
counter = counter + 3;
```

can be replaced by

```
counter += 3;
```

Consequently, the general form is

⟨*variable*⟩ ⟨*operator*⟩= ⟨*expression*⟩

where ⟨*operator*⟩ can be any of

```
+  -  *  /  %   &  |  ^  <<  >>
```

The last five operators are for bitwise operations which are discussed below. On the left side, ⟨*variable*⟩ stands for an arbitrary variable to which the value is assigned, an element of an array or of a structure, see Sec. 1.1.4. The ⟨*expression*⟩ on the right side can be any valid arithmetic expression, e.g. it may contain several operators, nested formulas, and calls to functions. Standard mathematical functions are introduced at the end of this section.

GET SOURCE CODE
DIR: `c-programming` FILE(S): `increment.c`

There are special unary operators `++` and `--` which increase and decrease, respectively, a variable by one, e.g. `counter++`. It can be applied to any variable of numeric or pointer type. In addition to the change of the variable, this represents an expression which evaluates to the value of the variable *before* the operator is applied or *after* the operator is applied, depending on whether the operator is written *behind* (postincrement) or *in front* (preincrement) of the variable. For example, compiling and running

```
int counter1, counter2, counter3, counter4;

counter1 = 3;
counter2 = counter1++;
counter3 = ++counter1;
counter4 = --counter1;
printf("%d %d %d %d\n", counter1, counter2, counter3, counter4);
```

will result in

```
4 3 5 4
```

Since the computer uses a binary representation of everything, in particular any natural number n can be interpreted as sequence $a_k a_{k-1} \ldots a_1 a_0$ of 0s and 1s when writing the number in binary representation $n = \sum_i 2^i a_i$. For instance, the number 201 is written as binary sequence (highest order bits at the left) `11001001`. For these sequences, there are a couple of operators which act directly on the bits. We start with operators involving two arguments.

`&` Calculates a bitwise AND of the two operands. For each pair (a, b) of bits, the `&` operation is defined as shown in the table on the right: The result is only 1 if a AND b are 1. Hence, for the numbers 201 (binary `11001001`) and 158 (binary `10011110`) one will obtain from `201 & 158` the result 136 (binary `10001000`).

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

`|` Calculates a bitwise OR of the two operands, defined as shown in the table on the right: The result is 1 if a OR b are 1. Therefore, for the numbers 201 and 158 one will obtain the result 223 (binary `11011111`).

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

`^` Calculates a bitwise XOR of the two operands, defined as shown in the table on the right: The result is 1 if a OR b is 1 but not both. Hence, for the numbers 201 and 158 one will obtain the result 87 (binary `01010111`).

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	0

`<<` The expression `seq << n` evaluates to a shift of the sequence of bits stored in the variable `seq` (or obtained from an expression in general) by `n` positions to the left, which is equivalent to a multiplication with 2^n. Thus, if `seq` contains 51 (binary `00110011`) `seq << 2` will result in 204 (binary `11001100`). Note that if the number overflows, i.e. gets too large relative to the type of the variable or expression, the higher order bits will be erased. Consequently, if `seq` is of type `unsigned char`, `seq << 4` will result in 48 (binary `00110000`).

`>>` Likewise, the expression `seq >> n` evaluates to a shift of the sequence of bits stored in the variable `seq` by `n` positions to the right, which is equivalent to an integer division by 2^n.

Furthermore, `~` is a unary inversion operator, which just replaces all `1`s by `0`s and all `0`s by `1`s. Therefore, the result depends on the standard number of bits, because all leading `0`s will be set to one. For example, variable `seq` containing the value 12 (binary `1100`, when omitting leading `0`s as usual), `~seq` will result in (4 bytes = 32 bits for `int`) 4294967283 (binary `11111111111111111111111111110011`).

For floating point arithmetic, there are many predefined functions like trigonometric functions, exponentials/logarithms, absolute value, Bessel functions, Gamma function, error function, or maximum function. The declarations of these functions are contained in the header file `math.h`. Thus, this file must be included, when using these functions. When compiling without `-Wall`, note that the compiler will not complain, if `math.h` is not included and this may yield incorrect results. Furthermore, `math.h` contains definitions of useful constants like the value of π or the Euler number e. The following simple program `mathtest.c` illustrates how to use mathematical functions.

GET SOURCE CODE
DIR: `c-programming` FILE(S): `mathtest.c`

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6    double z= 0.25*M_PI;
7    printf("%f %f %f\n", sin(z), cos(z), tan(z));
8    printf("%f %f \n", asin(sqrt(2.0)/2.0)/M_PI, acos(0.0)/M_PI);
9    printf("%f %f\n", pow(M_E, 1.5), exp(1.5));
10   printf("%f %f %f\n", fabs(-3.4), floor(-3.4), floor(3.4));
11   printf("%f %f\n", fmax(1.3, 2.67), fmin(1.3, 2.67));
12   printf("%e %e %e %e\n", erf(0.0), erf(1.0), erf(2.0), erf(5.0));
13   printf("%f %f %f %f\n", tgamma(4.0), tgamma(4.5),
14                           tgamma(5.0), exp(gamma(5.0)) );
15   printf("%f %f %f\n", j0(2.0), j1(2.0), jn(1.0,2.0));
16   return(0);
17 }
```

When compiling this program, one needs the additional option `-lm`, which means that the library for mathematical functions will be linked. In this library, precompiled code for all of these functions is contained and those which are needed will be included in the final program.[2]. You can compile the program using

```
cc -o mathtest -Wall mathtest.c -lm
```

In line 6 of the program, the predefined constant for π (3.14159265358979323846) is used. Other important constants are the Euler number (`M_E` = 2.7182818284590452354) and $\sqrt{2}$ (`M_SQRT2` = 1.41421356237309504880). Lines 7–8 illustrate the usage of trigonometric functions and their inverse functions. Note that the arguments have to be in radians, i.e. units of 2π. Line 9 shows how to use the general power a^x `pow()` and the exponential e^x `exp()` functions. In line 10, the function `fabs()` for calculating the absolute value of a number, and `floor()` for rounding downwards are shown. Line 11 introduces the usage of the functions `fmax()`

[2]Note that *dynamical* linking is usually used, i.e. the corresponding code is only linked when running the program. This keeps the executables short. Nevertheless, explicit linking at compile time can be forced using the option `-static`

and `fmin()` for calculating the maximum and the minimum of two numbers, respectively. In line 12, four values of the error function $\mathrm{erf}(x) = 2/\sqrt{\pi}\int_0^x e^{-y^2}\,dy$ are obtained. In line 13, sample usages of the gamma function are shown. Note that `tgamma()` is the gamma function $\Gamma(x)$ while `gamma()` represents $\ln\Gamma(x)$. Remember that for integer values n one has $\Gamma(n) = (n-1)! = \prod_{k=1}^{n-1} k$. Finally, in line 15, the Bessel function of the first kind is shown. `j(`n, x`)` calculates the Bessel function of order n, while $j0$ and $j1$ are shortcuts for `jn(0, . )` and `jn(1, . )`, respectively. Hence, when running the program, you will get

```
0.707107 0.707107 1.000000
0.250000 0.500000
4.481689 4.481689 0.000000 1.000000
3.400000 -4.000000 3.000000
2.670000 1.300000
0.000000e+00 8.427008e-01 9.953223e-01 1.000000e-00
6.000000 11.631728 24.000000 24.000000
0.223891 0.576725 0.576725
```

These functions are documented on the corresponding man pages, e.g. `man sin`. More complicated functions, like Airy functions, elliptic functions, or spherical harmonics can be found in special libraries, as explained in Chap. 6.

Finally, there is a special operator `?`, which is related to the `if` statement and is explained in the next section.

1.1.3 Control Statements

Using only the statements presented so far, a program would just consist of a linear sequence of operations, which does not allow for complex problems to be solved. In this section, statements are introduced which enable the execution flow of a program to be controlled.

The simplest statement is the `if` statement, which allows to select between two statements depending on the evaluation of a conditional expression. A simple example is

```
if(x <= 0)
  step_function = 0.0;
else
  step_function = 1.0;
```

where the variable `step_function` will be assigned the value 0.0 if `x` is smaller than or equal to zero, and `step_function` will get 1.0 if `x` is larger than zero.

The general form of an `if` statement is as follows:

`if(` ⟨*condition*⟩ `)`
  ⟨*statement 1*⟩
`else`
  ⟨*statement 2*⟩

During the execution, first the ⟨*condition*⟩ is checked. If the condition is *true*, then ⟨*statement 1*⟩ is executed. If the condition is *false*, then ⟨*statement 2*⟩ is executed. These statements can be arbitrary, e.g. assignments as in the above example, calls to functions, another nested `if` statement, etc. Also, the type of the statements can be different, e.g. ⟨*statement 1*⟩ may be a call to a function and ⟨*statement 2*⟩ may be an assignment. Note that the `else` part can be absent. In this case, if the condition is *false*, the execution continues with the next statement after the `if` statement.

Basic conditional expressions can be formed by comparing (arithmetic) expressions using relational operators. In C we have

```
==  !=  <  <=  >  >=
```

Consequently, an expression of the form ⟨*expr1*⟩ ⟨*OP*⟩ ⟨*expr2*⟩, will be evaluated for ⟨*OP*⟩ being `==` to *true* if both expressions are equal, while for the operator `!=` represents the test for non-equality. For the case ⟨*OP*⟩ being `<=`, *true* results if ⟨*expr1*⟩ is smaller than or equal to ⟨*expr2*⟩, etc. Note that the relational operators have lower precedence than the standard arithmetic operators, meaning they will be evaluated last. This means that `counter < limit + 1` is equivalent to `counter < (limit + 1)`.

C uses the integer 0 to represent a logical *false* and *all* other values to represent *true*. Therefore, for the comparison `counter==n_max` one could write instead `counter-n_max`. This results in exactly the same behavior, since this expression becomes zero if `counter` is equal to `n_max`. In fact, C does not formally distinguish between arbitrary expressions and conditional expressions. Thus, wherever a condition is expected, any expression can appear in a C program. Nevertheless, to make a program more readable, which usually leads to less programming bugs, one should use conditional operators, if conditions are possible and meaningful.

A common mistake, as mentioned on page 5, is to use `=` instead of `==` when intending a comparison. Nevertheless, a statement like `if(a = b+1)` is valid C syntax, because an assignment is also an expression itself, where the result is the value which is assigned. Hence, in this case the result of the expression will be `b+1` which in C represents *false* if `b` equals -1 and *true* for all other values of `b`. Usually, an assignment inside an `if` statement is not intended by the program author. Hence, you should use the compiler option `-Wall`, which makes the compiler print out a warning, when it encounters such a case.

Compound conditional expressions can be formed by grouping them using braces and combining them using the logical OR operator `||`, the logical AND operator `&&`, and the logical NOT operator `!`, e.g.

```
(counter>=10) && (counter<=50) && !(counter % 2==0)
```

which evaluates to *true* if the value of `counter` is inside the interval $[10, 50]$ and not an even number, i.e. odd.[3] The NOT operator has higher precedence

[3] Instead of `!(counter % 2==0)` one could also write `(counter % 2!=0)` or `(counter % 2==1)` or right away `(counter % 2)`.

than the AND/OR operators. Note that a sequence of `||` or `&&` operators is evaluated from left to right. This evaluation stops when the result of the full expression is determined, e.g. after the first occurrence of *true* for a sequence of || operators. This can be dangerous when using, inside a compound conditional expression, operators which change variables, like the post-increment operator `++`, as used in `(num_particles > 100)&&(num_run++ > 100)`. In this case, it depends on the result of the first condition whether the post-increment operator in the second condition is performed or not. Consequently, in most cases such constructions should be avoided for clarity, even if they work as intended.

Recall the first example of this section, where a value is assigned to a single variable, which is different within the different branches of the statement. For this special usage of the `if` statement, there exists the *conditional operator* `?` as shortcut. For the above example the resulting *conditional expression* statement reads

```
step_function = (x<=0) ? 0.0 : 1.0;
```

The general format of the conditional expression is

⟨*condition*⟩ `?` ⟨*expression 1*⟩ : ⟨*expression 2*⟩

Hence, the `?` operator is ternary. The result of the conditional expression is ⟨*expression 1*⟩ if the ⟨*condition*⟩ evaluates to true, else the result is ⟨*expression 2*⟩. The conditional expression can be used where a standard expression can be used.

According to the general form of the `if` statement shown above, it might appear that only one statement can be put into each of the two branches. This restriction can be overcome by grouping any number of statements into a *block*, which is created by embracing the statements by `{ ... }` braces, e.g.

```
1  if(step % delta_measurement == 0)
2  {
3    num_measurements++;
4    sum += energy;
5    sum_squared += energy*energy;
6    sum_cubed += energy*energy*energy;
7    sum_quad += energy*energy*energy*energy;
8  }
```

This piece of code is from a simulation program, where the (somehow calculated) energy is recorded, if the number of steps is a multiple of `delta_measurement`. To obtain estimations for the average energy and a confidence interval ("error bar") as well as higher moments, one calculates running sums of the energy (line 4), the energy squared (line 5), the third (line 6) and the fourth power (line 7). The average energy will be `sum_energy/num_measurements`. For details on the calculation of simple statistical properties, see Sec. 7.3.

Note that it is also possible to define variables inside a block. For C, the variable definition must appear at the beginning of the block, as it must appear

at the beginning of `main()`, and in fact at the beginning of any function.[4] Using a block variable `squared_energy`, the above example would read

```
1   if(step % delta_measurement == 0)
2   {
3     double squared_energy = energy*energy;
4
5     num_measurements++;
6     sum += energy;
7     sum_squared += squared_energy;
8     sum_cubed += squared_energy*energy;
9     sum_quad += squared_energy*squared_energy;
10  }
```

In this way, the number of multiplications is reduced from five to three. Note that such block variables are only visible and accessible within the block where they are defined. The scope of variables will be explained in more detail in Sec. 1.2.

Note that in case of nesting `if...else` statements, sometimes one needs blocks to avoid ambiguities, e.g.

```
if(x==1)
  if(y==1)
    printf("case 11\n");
  else
    printf("case 1X\n");
```

is equivalent to

[4] This is different from C++, the object-oriented extension of C, where variables can be defined everywhere. Nevertheless, the clarity of a program is increased, if all definitions of a block are collected in one place.

```
if(x==1)
{
  if(y==1)
    printf("case 11\n");
  else
    printf("case 1X\n");
}
```

which is different from

```
if(x==1)
{
  if(y==1)
    printf("case 11\n");
}
else
  printf("case 1X\n");
```

Therefore, one needs the brackets, if the `else` part should explicitly belong to the first `if` statement. In any case, if several `if` statements are nested, brackets always help to clarify their meaning.

Next, we consider the `for` loop, which enables one statement or a block of statements to be executed several times. The following piece of code calculates the sum of the squares of numbers from 1 to `n_max`, with `n_max=100` here:

```
1  int counter, sum, n_max;
2
3  sum = 0; n_max = 100;
4  for(counter = 1; counter <= n_max; counter++)
5    sum += counter*counter;
```

The `for` command (line 4) controls how often the addition in line 4 is executed, with `counter` getting the values $1, 2, 3, \ldots, 99, 100$, one after the other. Consequently, `sum` will contain the value 338350, when the loop is finished. The general form of a `for` loop is as follows:

`for(` ⟨*init expression*⟩; ⟨*condition*⟩; ⟨*increment expression*⟩ `)`
⟨*body statement*⟩

During the execution, first the ⟨*init expression*⟩ is evaluated. Next, the ⟨*condition*⟩ is checked. If the condition is *true*, then the ⟨*body statement*⟩ is executed, next the ⟨*increment expression*⟩ evaluated. This completes the execution of one iteration of the loop. Then, once again the condition is tested to check whether it is still *true*. If it is, the loop is continued in the same way. This is continued until the ⟨*condition*⟩ becomes *false*. Then the execution of the `for` loop is finished. Note that it is not guaranteed that a loop finishes at some time, e.g. the following loop will run forever:

```
for(counter = 1; counter > 0; counter++)
  sum += counter*counter;
```

The ⟨*init expression*⟩, the ⟨*condition*⟩, and the ⟨*increment expression*⟩ are usually arithmetic expressions or assignments. Hence, instead of `counter++` one could write `counter = counter +1`; Due to its flexibility, C allows for even more general forms, e.g. one could use `printf("%d\n", counter++)` as ⟨*increment expression*⟩, which would result in printing the value of `counter` after each iteration of the loop. As you see, C allows for very compact code. Nevertheless, this often makes the program listing somehow hard to read, hence error-prone. Thus, it is preferable to write programs which might be a bit longer but whose meaning is obvious. Therefore, putting the `printf` statement also in the body of the loop in this case would lead to simpler-to-understand code. In this case one also has to use `{...}` braces to group a list of statements into one block.

In principle, one can use quite general constructions to perform a loop, so one could use also variables of type `double` as iteration counters like in

```
double integral, delta, x;

integral = 0.0; delta = 0.01;
for(x=0.0; x <=M_PI; x += delta)
  integral += delta*sin(x);
```

There are another two types of loops, the `while` loop and the `do ... while` loop. The former one has the following general form:

`while(` ⟨*condition*⟩ `)`
⟨*statement*⟩

The above ⟨*statement*⟩ is executed as long as the condition is true. Consequently, if one likes to perform a loop similar to a `for` loop, one has to put the initialization before the while loop and one has to include the increment in the ⟨*statement*⟩, usually a block of several statements, like in the following example:

```
1  int counter, sum, n_max;
2
3  sum = 0; n_max = 100;
4  counter = 1;
5  while(counter <= n_max)
6  {
7    sum += counter*counter;
8    counter++;
9  }
```

Very similar is the `do ... while` loop, which has the following general form

`do`
⟨*statement*⟩
`while(` ⟨*condition*⟩ `)`

The difference from the while loop is that the ⟨*statement*⟩ (again it may be a `{...}` block of statements) is executed at least once. Only after each iteration of the loop, the ⟨*condition*⟩ is tested. The iteration of the loop continues as long as the ⟨*condition*⟩ is *true*.

There are three different statements, which allow to exit at any time the execution of a block belonging to a loop. First, the `break` statement will immediately stop the execution of the loop without completing the block. Hence, the execution of the program continues with the first statement after the loop statement. Second, the `continue` statement will also terminate the current iteration of the loop, but the full loop is continued as normal, i.e., with the ⟨*increment statement*⟩ in case of a `loop` or with the next evaluation of the ⟨*condition*⟩ in case of a `while` or a `do...while` loop. Thus, if the ⟨*condition*⟩ allows, the loop will be continued with the next iteration. If several loops are nested, the `break` and `continue` statements act only on the innermost loop. Third, if a `return` statement is encountered, not only the current loop will be terminated, but the whole function where the loop is contained in will be exited.

Note that there is a fourth possibility. The C language contains a `goto` statement which allows to jump everywhere in the program. Possible jump targets can be identified by *labels*. Since the use of `goto` statements makes a program hard to understand ("spaghetti code"), one should not use it. One can always get along well without `goto` and no details are presented here.

Finally, the `switch` statement should be mentioned. It can be used when the program should branch into one of a set of several statements or blocks, and where the branch selected depends only on the value of one given expression. As an example, one could have a simulation with several different atom types, and depending on the atom type different energy functions are used. In principle, one could use a set of nested `if...else` statements, but in some cases the code is clearer using a `switch`. Since the `switch` statement is not absolutely necessary, we do not go into details here and refer the reader to the literature.

1.1.4 Complex data types

So far, only single variables have been introduced to store information. Here, complex data structures will be explained which enable the programmer to implement vectors, matrices, higher order tensors, strings, and data structures where elements of possibly different data types are grouped together to form one joint data type.

To define e.g. a vector `intensity`, which may contain the results of 100 measurements during a simulation, one can write

```
double intensity[100];
```

In C, such a vector is called an *array*. By this definition, the variable `intensity` is defined, i.e., also enough memory will be *allocated* during execution of the program to hold the 100 *elements* of the array. These 100 elements are stored consecutively in the memory, hence a chunk of `100*sizeof(double)`

bytes will be used to store the array. Note that the memory, which is allocated, may contain *any* content, i.e. it must be regarded as undefined. Although some compilers initialize all allocated memory with 0s, your program should perform always an explicit initialization. The allocated memory is available until the execution of the *block*, where the array is defined, is finished. This holds for all variables defined inside a block, including variables holding just a single element. More details on this so-called *scope* of variables can be found in Sec. 1.2.

One can access the `i`'th element of the array by writing `intensity[i]`, i.e. the *index* (also called subscript) is written in `[]` brackets after the name of the vector. In C, array indices start with 0, hence here the array elements `intensity[0]`, `intensity[1]`, ..., `intensity[99]` are available. In the following example the average of the values in the array is calculated . Note that some part of the code is not shown, as indicated by a *comment* in the program (line 4). In C, comments are indicated by embracing the comment text in '`/*`' and '`*/`'.

```
1  double avg_intensity;
2  int i;
3
4  /* ... "some" more code to generate data ... */
5
6  avg_intensity = 0.0;
7  for(i=0; i<100; i++)
8    avg_intensity += intensity[i];
9  avg_intensity /= 100;
```

The index can be any expression which results in an integer, so one could also access every third measurement only, if needed, starting at index 1, by writing `intensity[3*i+1]`.

Technically, a one-dimensional array is equal to a pointer which points to the beginning of the memory chunk: `intensity` contains the address of the beginning of the chunk, while `intensity[0]` contains the content stored in the first element of chunk, which is equivalent to `*intensity`. The second element can be accessed via `intensity[1]` or via `*(intensity+1)`. More details about this equivalence will be given in Sec. 1.4.

Note that during the execution of the program it is *not* checked whether one accesses (reads or writes) elements *outside* the reserved range. Consequently, one could easily write `intensity[100] = 0.9;`, which is in fact a mistake that will *not* be detected by the compiler. Often, this will not do any harm during the execution of the program, because it means that a memory location will be accessed which is located just ahead the chunk that has been reserved for the `intensity` array. The basic reason is that memory is not allocated consecutively. Thus, very often the part just ahead this chunk will not be used to store other variables and consequently the program will not be affected. But *if* some other data is stored just ahead the chunk, e.g. when almost all the main memory is used, some other data will be overwritten by assigning a value to

`intensity[100]`. In this case, the program *might* crash, or it might just give strange results. Therefore, different runs of the same program with the same input parameters might sometimes crash, sometimes give wrong results, and sometimes even do fine. Since such kind of bugs are obviously hard to detect, there are special tools to find them, so-called *memory checkers*. For details on these *very* useful tools, see Sec. 5.3.

The general form of an array definition is as follows:

⟨*type*⟩ ⟨*variable name*⟩ [⟨*size*⟩];

The ⟨*type*⟩ can be any predefined as well as self-defined data types, see below. For example, an array with 10 elements of the type `short int` is defined as follows:

```
short int flags[10];
```

It is also possible to mix the definition of single variables and arrays (and other variables based on the same basic type) in one line, e.g.

```
double avg_intensity, variance, intensity[100];
```

For old-fashioned standard C (before the standard ISO C99), the size of an array has to be known by compile time. Therefore, for compilers supporting only this type of C, it is not possible to write `int vector[size]` where `size` is a variable. In this case, if one wants to use arrays, where the size is not known at compile time, i.e. *variable-sized arrays*, one can always use the techniques of dynamic memory management as described in Sec. 1.4. This is the safest approach and also the most flexible. On the other hand, up-to-date compilers, like the *gnu C compiler* `gcc` [Loukides and Oram (1996)], support the definition of variable-sized arrays. Nevertheless, in case you use them, you might face a problem in rare cases, when you want to port your program to another system with old compilers. Furthermore, it is not possible to resize the array after it has been created, in contrast to arrays created using dynamic memory management.

As mentioned above, you cannot rely on arrays being initialized automatically upon creation. On the other hand, it is possible, like for single-valued variables, to combine the definition with an initialization, e.g.

```
int atom_weight[6] = {1,6,7,8,15,16};
```

Note that the length of the arrays can be larger than the number of elements given for initialization. If no array size is given, the size of the array will be automatically equal to the number of elements provided, like in

```
int atom_weight[] = {1,6,7,8,15,16};
```

where 6 elements will be allocated for `atom_weight`.

A very special type of array is one which has the basic type `char`. It is used to store *strings* and can be defined e.g. via.

```
char atom_name[100];
```

One can assign values like for other types of arrays. Note that the last element of a string should contain a terminating 0. Hence, one could write:

```
atom_name[0] = 'c';
atom_name[1] = 'a';
atom_name[2] = 'r';
atom_name[3] = 'b';
atom_name[4] = 'o';
atom_name[5] = 'n';
atom_name[6] = 0;
```

This is quite space consuming. A simpler form exists, in case one performs an assignment together with the definition, i.e. an initialization:

```
char atom_name[100] = {"carbon"};
```

Consequently, it is not necessary to assign one element after the other. To indicate string constants, double quotes are used, in contrast to single quotes for single characters. The terminating 0 is not given explicitly.

To print a string using `printf()`, the conversion specification `%s` can be used. In this case, one only needs to pass the name of the array, not all variables one after the other. The printing system will automatically print the string character after character until the terminating 0 is reached, hence

```
printf("%s\n", atom_name);
```

will result in

```
carbon
```

being printed on the screen.

There are many auxiliary functions which are very useful for string processing. The declaration of these functions is contained in `string.h`, which must be included at the beginning of your program, if you use any of these functions. Here, only few examples are given: `strcpy` ("string copy") will copy the content of the string passed as second parameter, including the terminating 0, to the string passed as first parameter, like in

```
char atom_name1[100] = {"carbon"}, atom_name2[100];

strcpy(atom_name2, atom_name1);
```

Therefore, the content of `atom_name2` will be also `"carbon"`. Note that there is no check for the length of strings. Hence, if the second string until the terminating 0 is longer than the length of chunk reserved for the first string, one will write beyond the boundaries of the arrays. This may result in unpredicted behavior, as discussed above, and should be avoided (see Sec. 5.3). If you are

not sure about the lengths of the strings, you could use `strncpy`, which has an additional third argument that states the maximum number of characters to be copied.

Note also that the above assignment *cannot* be performed by writing `atom_name2 = atom_name1`, which might appear natural. In fact, writing this in a program will result in a compiling error.[5]

Strings can be compared using `strcmp()`. A call `strcmp(s1,s2)`, will return 0 if the strings are equal, a negative value (usually -1) if `s1` is lexicographically smaller than `s2` and a positive value if `s1` is lexicographically larger than `s2`.

Another useful function is `sprintf()`, which allows a programmer to assemble strings easily. It works similar to `printf()`. As a difference, the result is not printed to the screen, but to the string which is passed as additional first argument. This is in particular useful when assembling file names, usually to write out simulation results, where the file names somehow contain some program parameters to distinguish different runs. This could look like

```
sprintf(file_name1, "sim_N%d_T%3.2f_id%d.out",
        num_particles, temperature, run_id);
```

Note that `%3.2f` means that a floating point number with at least 3 digits, 2 among them after the comma, is printed. More details on the use of `printf()` and related commands, including these conversion specifications, can be found in Sec. 1.3. Hence, if `num_particles` = 10, `temperature` = 1.3 and `run_id` = 15, the resulting file name will be

```
sim_N10_T1.30_id15.out
```

This file name can be used to create a file where one can write some data to, see again Sec. 1.3 on file creation and input/output.

Also, strings can be converted back to numbers. The "inverse" function to `sprintf()` is `sscanf()` and will be described in Sec. 1.3. For the special case a string `s` contains an integer number, i.e. the digits of the corresponding number, one can obtain the number as `int` value via the `atoi()` function:

```
number = atoi(s);
```

For more details on string functions, please refer to the documentation, e.g. `man string` will give an overview over many useful string functions.

Matrices or tensors can be implemented using multi-dimensional arrays. E.g a matrix with elements of type `double` can be defined via

```
double mat1[10][20];
```

[5] This is different when explicitly using pointers instead of arrays, i.e. `char *atom_name1, *atom_name2`. In this case one must explicitly allocate memory, see Sec. 1.4. Then one *could* in principle perform the given assignment, but no string will be copied, just both pointers will point to the same string. Afterward, a change of the content of `atom_name1` would also change the content of `atom_name2` since both are the same. Also, if some memory had been allocated for `atom_name2`, then it will be "lost" after this assignment, in case no other pointer refers to it. This would be a real bug.

The `i,j`'th element of the matrix `mat1` is accessed using `mat1[i][j]`, again arbitrary `int` expressions can be used instead of `i` and `j`. Note that again no automatic check for array boundaries is performed. Assuming that `vec1` and `vec2` are `double` arrays of lengths 20 and 10, respectively, the following piece of code multiplies the matrix `mat1` with the vector `vec1` and stores the result in the vector `vec2`, which is finally printed:

```
1  for(i=0; i<10; i++)
2  {
3    vec2[i] = 0.0;
4    for(j=0; j<20; j++)
5      vec2[i] += mat1[i][j]*vec1[j];
6  }
7  for(i=0; i<10; i++)
8    printf("%f\n", vec2[i]);
```

Technically, the compiler treats multi-dimensional arrays like a big one-dimensional array. Let's see how this works for the matrix: First, a big chunk of memory is allocated (again often uninitialized). Next, the array is stored in this chunck consecutively row by row.

Consequently, in this case, the matrix will be stored in a chunk of 10×20 `double` elements each. Therefore, `mat1` will contain the address where chunk starts, which is the same as `mat1[0]`. `mat[1]` contains the address where the part for the second row starts, while `mat[0][0]` is the content of the first element of the first row, and so on. The size of the allocated memory can be evaluated afterward at any position in the program using again the `sizeof()` function. For example, `sizeof(mat1)` will result in the total number of elements $20 \times 10\times$`sizeof(double)`, i.e. in 1,600 if `double` uses 8 bytes. The amount of memory reserved for one row can be obtained via `sizeof(mat1[0])`.

Even higher-dimensional arrays can be defined like

```
double tensor1[10][20][10];
```

and accessed e.g. via `tensor1[1][2][3] = 5.0`. Further details are not necessary here. In exercise (2), permutation matrices are considered.

GET SOURCE CODE
DIR: `c-programming` FILE(S): `person.c`

For arrays and matrices, several elements of the same type are put together. In many other cases, one would like to treat a set of elements of different types as single units. Consider as toy example a simulation of a social system of people. Here, one wants to store, for example, the age, height, sex, marital status and city of residence for each person. In C, one can group these element together using a *structure*. The elements are then called *members* of the structure. An example declaration of a structure for persons, including extensive comments, reads as follows:

```
struct person
{
  int          age; /* age of person (years)                   */
  double    height; /* height of person in meters              */
  short int    sex; /* 0=male, 1=female                        */
  short int status; /* 0=single,1=married,2=divorced,3=widowed */
  int         city; /* cities are coded as integers 1,...      */
};
```

It is possible for the members to have also more complex data types like arrays, other structures, or self-defined data types (see below).

To define variables, e.g. `p1, p2`, which are of type `struct person`, one writes

```
struct person p1, p2;
```

Alternatively, one could also list the variables to be defined directly after the closing bracket '`}`' of the structure declaration itself (and just before the final semicolon, which closes the declaration).

The members of a structure variable can be accessed by writing the variable name, a dot '`.`' and the name of the member. For instance, one could assign values to all members of `p1` using

```
p1.age = 43;
p1.height = 1.73;
p1.sex = 0;
p1.status = 0;
p1.city = 23;
```

It is also possible to assign complete structures within one statement like in

```
p2 = p1;
```

Thus, `p2.age` will have value 43, `p2.height` will be `1.73`, etc.

Now, `struct person` can be used as any existing type name. In this way, it is possible to define an array which contains all persons of the simulation:

```
int num_persons = 100;
struct person pers[num_persons];
```

The access to the different members of the elements of the `pers` array is obtained by combining the access methods for arrays with the access method for structures. For example, to initialize the marital status of all persons to 0 one could use:

```
for(p=0; p<num_persons; p++)
  pers[p].status = 0;
```

Please note that `pers.status[p]=0` is *not* correct in *this* case. It would be correct, if `pers` contained as member an array `status`.

Sometimes, it is useful to declare new names for data types, in particular if they are of complicated form like pointers to functions (see Sec. 1.4). The `typedef` construct serves this prupose. The format is the same as that of a single variable definition, but preceded by the keyword `typedef`, i.e.:

`typedef` ⟨*type*⟩ ⟨*type name*⟩;

Consequently, the name given is now for a "new" type, not for a variable. E.g., to define the new type name `person_t` for the `person` structure, one could write

```
typedef struct person person_t;

person_t p1, p2;
```

It is also possible to combine the structure declaration and the definition of the new name. In this case, one does not need to specify a name after the key word `struct`:

```
typedef struct
{
  int          age; /* age of person (years)                   */
  double    height; /* height of person in meters              */
  short int    sex; /* 0=male, 1=female                        */
  short int status; /* 0=single,1=married,2=divorced,3=widowed */
  int         city; /* cities are coded as integers 1,...      */
} person_t;
```

The author recommends to use the suffix `_t` (or similar rules) for the names of self-defined types to make it obvious everywhere that a new type name is used.

It is even possible just to define via `typedef` new names for existing data types like `double`. This could be useful e.g. in a situation where one knows in advance that one needs a sophisticated data structure later on for some variables, like special types for extremely high-precision numbers. In this case, one might want to start the implementation for development purposes with standard `double` and only switch to the high-precision data type later on. Hence, one could use `typedef double precision_t` so that one would only have to change this `typedef` and not all occurrences of definitions of variables of the high-precision type later on.

C offers other data types, e.g. *enumerations* (where each variable can have only specific values), *bit fields* (for storing data bit wise) and *unions* (for allowing variables to be at different types), which are not covered here. These data types are useful for some applications but not absolutely necessary. Please refer to the literature for details.

Based on these complex data types, even more advanced data structures can be developed. They are often very useful for organizing simulation programs

in a much better way, speeding up simulations considerably. Advanced data structures like lists, trees and graphs are discussed in Secs. 4.6 to 4.8.

1.2 Functions

GET SOURCE CODE
DIR: `c-programming` FILE(S): `step_fct.c`

Usually, there are many complex tasks, which have to be performed at different places in a simulation, e.g. evaluating mathematical functions or calculating statistical properties of a given set of numbers. One could explicitly include code for the calculation each time, for example for the step function as shown on page 17. But this is a waste of programming effort and makes the code not well structured. It is better to move this task to a *subroutine*, which in C is always a *function*. A function for calculating the step function could look like this:

```
1  /************* step_function() ******************/
2  /** Mathematical step function:                 **/
3  /** Returns 0 if argument is less or equal to 0 **/
4  /**         and 1 else                          **/
5  /**                                             **/
6  /** Parameters: (*) = return parameter          **/
7  /**         x: mathematical argument            **/
8  /*                                              **/
9  /** Returns:                                    **/
10 /**       step function value                   **/
11 /************************************************/
12 double step_function(double x)
13 {
14   if(x <= 0.0)
15     return(0.0);
16   else
17     return(1.0);
18 }
```

As you see, the code of each function should be accompanied by a *function comment* (lines 1–11) which explains what the function does, what arguments are to be passed and what is returned. Although this is not necessary for the program to compile correctly, you should get used to writing function comments, and other comments, right from the beginning. *Whenever* I code a function, I start with the function comment! Medium or large simulation projects, which have more than few pages of source code, in particular if their lifetime is longer than some weeks, cannot be handled efficiently without a good practice of commenting. More about this and other aspects of *software engineering* can be found in Secs. 2.1 and 2.2. Note that all example source codes, which are available online for this book, carry extensive comments. Therefore, all functions will exhibit such a function comment. Nevertheless, to save space, the functions presented in the book will usually *not* show their respective comments.

In line 12, first the *return type* is declared, i.e. it will return a `double` value here. Next, the function name `step_function` is given. Next, enclosed by `(...)` brackets, the arguments to the function are listed. Here, one variable of type `double` is the argument, which can be addressed by the name `x` inside the function. Note that several arguments are possible. In this case one has to separate them by commas (see below). The main part of a function (here lines 13–18) is enclosed by `{...}` brackets. The main part consists of definitions and statements, as in the `main()` part of a full program. In fact, as you may have realized by now, the `main()` part of the program follows the same syntax rules like any other function, with the only difference that it always has to have the special name `main()`. Finally, the value returned by the function is determined via the `return()` statement. There can be several `return()` statements inside the main part of the function, but only one will be executed each time the function is called.

A function is called in other parts of the code by writing the function name and, in brackets, the list of arguments, quite the same as for predefined functions like `sin()` or `printf()`. The value returned by a function can be used as any result of an expression, e.g., printed or assigned to a value of suitable type. The main program for our example, which may contain e.g. a loop, printing out a few function values in the interval $[-1, 1]$, could look like

```
1 int main()
2 {
3   double x;
4
5   for(x=-1.0; x<=1.0; x+=0.1)     /* print step function in [-1,1] */
6     printf("Theta(%f)=%f\n", x, step_function(x));
7
8   return(0);
9 }
```

When running the program, the output looks like (only a part is shown here, as indicated by the dots)

```
            .
            .
Theta(-0.100000)=0.000000
Theta(-0.000000)=0.000000
Theta(0.100000)=1.000000
Theta(0.200000)=1.000000
            .
            .
```

To be able to use the function `step_function()` inside `main()`, the compiler has to "be aware" of it when it compiles the code for `main()`. There are four ways of achieving this:

1. The full function definition of `step_function()` appears *before* `main()` in the source file. Then, when the compiler reaches the source code of

`main()`, it will have processed already the part where `step_function()` is defined, hence the function is known to the compiler.

In this way, `step_function()` is a *global* object, hence it can also be called from other functions, not only `main()`.

2. The full function definition comes *after* `main()` in the source file. To make the compiler aware of the existence of `step_function()` when it processes `main()`, one has to perform a declaration in advance, which is called *function prototype*. This means that in the source code before the definition of `main()`, the following line has to appear:

```
double step_function(double);
```

Note the closing semicolon at the end of the line. The function prototype carries all information which has to be known outside the function: the function name, the types and order of the arguments, and the type of the return argument.

All other functions which appear in the source code behind the function prototype of `step_function()` can use calls to `step_function()` in their code. Hence, if *all* function prototypes are listed at the beginning, each function is allowed to call each other function.

3. Very often, the source code is distributed over many source files. This usually happens if your simulation program is large and can be decomposed into several independent modules. There might be, for example, one module for setting up the simulation, another for auxiliary functions, some for the main simulation functions, and so forth. In this case, the second possibility has to be used, too. This means that for each source file where some function `func()` is to be used, a corresponding function prototype must be known, when it is compiled. Usually, the function prototype is not written in the file explicitly, but it is contained in a *header* file, just as the function prototypes for standard functions like `sin()` and `printf()`. This header file must be included using the `#include` directive, see Sec. 1.6. Note that for including your own header files, you have to use the form `#include "header.h"` instead of `#include <header.h>`, which is only used for predefined (system) header files.

In this case, where the function prototype is contained in a header file, the function prototype must be preceded by the key word `external`, which indicates that the function definition occurs outside the present file, e.g.

```
external double step_function(double);
```

Now, the function `step_function()` can always be used after the corresponding header file has been made known by using `#include`. Usually, one puts all `#include` directives at the beginning of a source file, such that all definitions are known everywhere inside the source file.

4. All first three possibilities allow `step_function()` to be used arbitrarily in other functions, provided that the functions are informed about `step_function()` via the definition itself or via a prototype. The reason is that `step_function()` is a *global* object.

   Nevertheless, it is possible to put the definition of `step_function()` just inside `main()`, like e.g. in:

```
1   int main()
2   {
3     double x;
4     double step_function(double x)
5     {
6       if(x <= 0.0)
7         return(0.0);
8       else
9         return(1.0);
10    }
11
12    for(x=-1.0; x<=1.0; x+=0.1) /* print step function in [-1,1] */
13      printf("Theta(%f)=%f\n", x, step_function(x));
14    return(0);
15  }
```

   In this case, `step_function()` can be only used *inside* `main()`. Now, `step_function()` is a *local* object. If you have another function in the source code which calls `step_function()`, the compiler will complain!

Thus, cases 1-3 provide *global* definitions of `step_function()`, while case 4 is a *local* definition. This distinction can also be made for other definitions like for variable definitions or type declarations. Nevertheless, type declarations are usually global. Function and variable definitions can be used quite often in the same context. Technically, in these two cases a name is connected to an address, either the address where the variable content is stored, or where the machine code for a function can be found.

The general form (according to ANSI standard) of a function definition is as follows:

⟨*return type*⟩ ⟨*function name*⟩ `(` ⟨*arguments*⟩ `)`
{
  ⟨*main part of function*⟩
}

For the return type and the comma-separated list of arguments, variables of arbitrary type can be used. Later we will see examples where even functions are passed as arguments to functions, see e.g. Sec. 6.1. Note that the list of arguments can be empty, i.e. no arguments are passed. The ⟨*main*

part of function⟩ can be any type of valid code, including definitions of variables, types and functions, and all kinds of statements. The variables defined here are local, as mentioned above. For the next example, we consider a function which takes an array of `double` numbers, and should return the minimum and maximum numbers. The function should work for arrays of arbitrary number of elements, hence one has to pass the array, in fact a pointer to the beginning of the array, and the number of elements. Since each function can return only one object, one can declare a structure with two elements, which should take the result, as in the following example:

GET SOURCE CODE
DIR: `c-programming` FILE(S): `min_max.c`

```
1  typedef struct
2  {
3    double min;
4    double max;
5  } minmax_t;
6
7  minmax_t minmax(double x[], int num)
8  {
9    int i;
10   minmax_t mm;
11
12   mm.min = x[0];
13   mm.max = x[0];
14
15   for(i=1; i<num; i++)
16   {
17     if(x[i] < mm.min)
18       mm.min = x[i];
19     if(x[i] > mm.max)
20       mm.max = x[i];
21   }
22
23   return(mm);
24 }
```

The first argument of the function (line 7) shows that `x` is an array, but without specifying the size. Hence, the `[]` brackets are empty. Note that one does not need to pass the number of elements fitting into the array; it can be obtained using the `sizeof()` statement, see page 28. Nevertheless, it is more obvious what is going on when the number is passed explicitly. The function is also more flexible in this way, because one might not use all entries of a large array.

Note that `minmax()` contains, in lines 9 and 10, definitions of local variables. The actual determination of the minimum and maximum elements is done in lines 12–21. The result is returned in line 23.

There is also an "old" (non ANSI standard) form of a function definition, where inside the `(...)` brackets only the parameters are stated, but no types are given. In this old form, the types are listed separately behind the `(...)` brackets and before the `{...}` block, where the function's main code is contained, e.g.

```
minmax_t minmax(x, num)
double x[];
int num;
{
  ...
}
```

Sometimes you do not want a function to explicitly return something. In this case, you can use the special type `void`. No return statement is necessary now. One can still use the `return` statement to leave a function, but it must be used without a return value. Without an explicit `return` statement, the function is just executed till the end of the block is reached.

Again, considering the `minmax()` example, instead of returning a structure of type `minmax_t` for returning two numbers, one can use a different approach: One can pass pointers (see page 10), which point to the variables where the return values should be stored. Using pointers enables the content of these variables to be changed. For the given example, this could look like:

```
1  void minmax2(double x[], int num, double *p_min, double *p_max)
2  {
3    int i;
4    double min_value, max_value;
5
6    min_value = x[0];
7    max_value = x[0];
8
9    for(i=1; i<num; i++)
10   {
11     if(x[i] < min_value)
12       min_value = x[i];
13     if(x[i] > max_value)
14       max_value = x[i];
15   }
16   *p_min = min_value;
17   *p_max = max_value;
18 }
```

When calling `minmax2()`, one must pass the addresses where the results should be stored, e.g. like in

```
minmax2(x, num, &x_min, &x_max);
```

Using pointers in this case is necessary because arguments are always passed *by value*. This means that the content of the argument variable is copied to a

local variable of the function, i.e., which is accessible only inside the function, similar to the variables defined in the main part of the function. Consequently, if the value of an argument variable is changed inside the function, which is possible, it will not affect the content of the variable outside of the function. Consider, e.g., the following variant of `minmax()`:

```
1  minmax_t minmax(double x[], int num)
2  {
3    minmax_t mm;
4
5    mm.min = x[num-1];
6    mm.max = x[num-1];
7
8    while(num >=0)
9    {
10     if(x[num] < mm.min)
11       mm.min = x[num];
12     if(x[num] > mm.max)
13       mm.max = x[num];
14     num--;
15   }
16
17   return(mm);
18 }
```

Here, the argument `num` is also used as counter during the main loop (lines 8–15), hence it is changed (line 14) during the execution. Nevertheless, when the execution returns outside `minmax()`, the value which has been passed will still remain at its original value. The only way to change variables outside a function in C is to use pointers, as we have described above. Note again that passing an array is basically like passing a pointer. Thus, all changes to elements of an array will persist after the execution of the function has been terminated.[6] Nevertheless, one can quantify an argument, which is a pointer or an array, as being constant by writing `const` in front of the definition of the argument, e.g.

```
minmax_t minmax(const double x[], int num)
```

This means that no changes to the array `x[]` are allowed. This is sometimes useful to prevent programming errors, because if you know in advance that the content should not be altered and, when using `const`, accidentally include changes to this variable in the code, the compiler will complain.

As we have seen in the previous examples, it is possible to define variables inside a function, like `i`, `min_value` and `max_value` in `minmax2()`. Such variables are called *local* variables. The argument variables, which hold copies of

[6] Note that for C++ there is also the possibility of passing variables *by reference*, which means that all changes to the argument will be effective also after the function has been finished. Technically, this is achieved by passing in fact pointers, but this is hidden to the programmer, who can use the argument variables like any local variables.

the actual arguments passed to the function, are also local. These variables are accessible only *inside* the function where they are defined. This defines the *scope* of a variable. Local variables exist only during the time a function is executed: The memory for the variable will be reserved when the function is called and the variable will be "deleted", hence the memory freed again, when the call finishes. Thus, each time a function is called again, all local variables are created from scratch. As we have seen previously, one can have other local objects like functions. Structures and types can also be defined just locally. Note that the scope of a variable or any other object can also be restricted to any block enclosed in `{...}` brackets. Variables defined inside a block are accessible only within the block and they exist only during the execution of the block. Each time the block is executed again such a variable is defined from scratch. Thus, the variable has no memory of past executions of the block.

There is one exception, the so-called *static* local variables. They are identified by the key word `static` in front of the type. These variables are created upon the first call to the function, and they exist till the end of the execution of the program. Such variables are useful when giving functions some internal memory, e.g. to store a status variable which should be remembered each time the function is called. For static variables one gives usually an initialization together with the definition. This initialization is only performed during the first call to the function! For example, the following function counts how often it is called and returns this number each time it returns from execution:

GET SOURCE CODE
DIR: `c-programming` FILE(S): `staticvars.c`

```
1 int do_something(int n)
2 {
3   static int num_calls = 0;
4
5   printf("I got:%d\n", n);
6
7   return(++num_calls);
8 }
```

Another example for an application of static variables can be found in Sec. 7.2.1, where pseudo random number generators are treated.

On the other hand, there are *global* variables and other global objects, which are defined outside any function or block, hence on the level where `main()` is defined. These variables and other objects are accessible from everywhere![7] This makes it very convenient and is even necessary for the definitions of self-defined data types, which cannot be passed as arguments to functions. This may also appear convenient for variables, e.g. for a large-scale simulation consisting of many particles. Here, one could store their data in global variables, hence one does not have to pass these variables to all functions where they are used.

[7] As an exception, a global variable which is declared `static` is only accessible inside the same source file. This may be useful when you want to use a name for a global variable, which is already or may be used as global variable elsewhere.

Nevertheless, as discussed in Sec. 2.2, the use of global variables is very error-prone and also very inflexible. *Do not use global variables, whenever you can avoid it !!!*[8] Consider e.g. a function which calculates the average velocity among a set of moving particles. If you use global variables, you have to write a function for the velocity calculation explicitly stating the name of the array variable which is used to store the data, such as `particle[]`. Hence, if in your program you simulate also a second independent set of particles, say stored in `particleB[]`, you have to provide another function for the velocity calculation of `particleB[]`. This is inefficient, so it is better to have one single function, where the actual particles to be treated are passed as arguments. Extrapolating these considerations leads to the conclusion that all data should be passed via arguments to functions. Consequently, global variables are not necessary, even dangerous, and should be avoided.

Note that if you define a function `sub()` inside another function `func()`, all other variables and other objects defined in `func()` will be accessible from `sub()`, and hence are global relative to `sub()`. Furthermore, a local definition overrides any definitions which are global at the given point, as in the following example:

GET SOURCE CODE
DIR: `c-programming` FILE(S): `hide.c`

```
1  int number1, number2;    /* these are global variables */
2
3  void do_something()
4  {
5    int number1;         /* overides global definition */
6
7    number1 = 33;
8    printf("subr: n1=%d, n2=%d\n", number1, number2);
9    number2 = 44;
10   printf("subr: n1=%d, n2=%d\n", number1, number2);
11 }

12 int main()
13 {
14   number1 = 11;
15   number2 = 22;
16   printf("main: n1=%d, n2=%d\n", number1, number2);
17   do_something();
18   printf("main: n1=%d, n2=%d\n", number1, number2);
19
20   return(0);
21 }
```

Inside `do_something()` only the global variable `number2` is accessible, while `number1` is purely local. Consequently, the assignment of `number1` will not affect the global variable, but it does so for `number2`. Therefore, when running the program one gets

[8] This is basically always possible, except for some debugging purposes.

```
main: n1=11, n2=22
subr: n1=33, n2=22
subr: n1=33, n2=44
main: n1=11, n2=44
```

Nevertheless, one can always design simulation programs such that one does not need nested levels of variable scopes. Thus, for clarity they should be avoided unless absolutely necessary. There is a special compiler warning flag `-Wshadow`, which makes the compiler report when some local definition overrides a global definition. This flag is usually not included in the `-Wall` set of warnings.

More advanced programming techniques which are based on functions, like *recursion*, *divide-and-conquer*, and *backtracking* are discussed in Secs. 4.2 to 4.5.

1.3 Input/output

Since you want to know the results of your simulations, your program needs some output. The most basic command for this is `printf()`, which prints to the standard output, usually the screen, and which was already preliminarily discussed above. Usually, you write out some raw or intermediate results to files. For this purpose, `fprintf()` can be used. Below, we will also introduce the relevant commands for creating, opening, writing, closing, and deleting files. Typically, your intermediate results have to be read in again to be processed further. Hence, ways to read in data are discussed next. Also, simulation programs need command line arguments, which are discussed afterwards. Finally, an easy method to compress data files during the simulation is presented.

The `printf()` ("print formatted") command consists of a format string and some (possibly zero) expressions to be evaluated and then printed:

`printf(` ⟨*format strings*⟩, ⟨*argument 1*⟩, ⟨*argument 2*⟩, ...`)`

All arguments except the format string are optional. The format string may contain characters which are just printed as they are given. Furthermore, the format string determines how many of the optional arguments have to be given and it states how the values of these different arguments are to be printed. This is achieved using so-called *conversion specifications*, which are always preceded by the '`%`' character. Each conversion specification requires one argument, respectively. E.g., a single integer is printed using `%d`, as shown in the following example:

```
int num_particles = 100;

printf("number of particles: %d\n",  num_particles)
```

which will result in

```
number of particles: 100
```

`\n`	a new line
`\t`	a horizontal tabulator
`\b`	a backspace
`\\`	a backslash '\'
`\"`	a double quote "

This format string also contains a character `\n`, which is an *escape sequence* and means that the printing after this output has been printed will continue in the next line. The most important escape sequences are

There are different types o conversion specifications. The most important ones, in the context of simulations, are

`%d`, `%i`	for integer numbers
`%x`, `%X`	hexadecimal representation of integers
`%f`	for floating point representation
`%e`, `%E`	scientific ("exponential") notation
`%s`	strings
`%p`	pointers (addresses).

Note that one writes '`%%`' for printing a '`%`'. A complete list of conversion specifications can be found in the documentation of `printf()`, e.g. by typing `man 3 printf` under a UNIX system. Some of these conversion specifications appear in the following primitive example (note that the format string can be split into several pieces, like here):

GET SOURCE CODE
DIR: `c-programming` FILE(S): `printing.c`

```
int counter = 777;
double energy1 = 35678.99;
void *pointer = &energy1;
char name[100] = {"network"};

printf("After %d (hex:%x) iterations an energy of %f \n"
       "(%e, stored at %p) was obtained for %s\n",
       counter, counter, energy1, energy1, pointer, name);
```

When executing these lines of code, the following output will appear on the screen:

```
After 777 (hex:309) iterations an energy of 35678.990000
 (3.567899e+04, stored at 0xbfaee7d0) was obtained for network
```

Although `printf()` can handle an arbitrary number of arguments, the number of arguments given must always match the number of conversion specifications in the format string. Otherwise the compiler will complain.

When printing just using the raw conversion specifications, the format of e.g. a floating-point number will always be the same. The predefined format can be modified by optional specifications. Details can be found again in the

documentations (man pages). Here, we give only the most important examples. First, right after the '`%`' there can be a *flag*. Important flags are `0`, to fill numbers with leading zeros, and `-` for left adjustment. Next, the (minimum) field width can be stated, which is just an integer number. Next, one can optionally have a precision specifier, which is a dot '`.`' followed by another number (or `.*`, which means that the precision is given as next argument by an expression of type `int`). For floating point/ exponential numbers, this is the number of digits after the radix, while for integer numbers or strings it is the maximum field width. Finally, there can be a *length modification*. The most important one is `l` which stands for *long*, i.e. `%ld` will expect a `long int` as corresponding argument. The following simple statement gives some examples for the modifications of the output format:

```
printf("%06d, %4.3f, %-20s, %lf\n",
       45, 3.14159265358979, "Hello", 36.5);
```

which will result in the following output

```
000045, 3.142, Hello               , 36.500000
```

Instead of printing the output to the screen (called standard output `stdout`), one can also print to files, see below, or to strings. The latter is done via the `sprintf()` function, which has the following format

`sprintf(` ⟨*target string*⟩, ⟨*format strings*⟩, ⟨*argument 1*⟩, ...`)`

Consequently, as example one can use this to concatenate strings

```
sprintf(name, "%s %s", first_name, family_name);
```

In the context of computer simulations, this function is quite useful to assemble parameter-sensitive file names, as shown in the example on page 27.

Printing to a file is more involved. First one must *open* the file. This is done using `fopen()`, which has the following format:

`fopen(` ⟨*file name string*⟩, ⟨*access mode string*⟩`)`

GET SOURCE CODE
DIR: `c-programming` FILE(S): `file_o.c`

Thus, one must provide two argument strings. The first one contains the name of the file. This string may contain the path name relative to the working directory from which the program is started.[9] The second string states the access mode, which can be `"w"` for *writing*, which means that a file will be created from scratch. Hence, if the file has existed previously, it will be deleted. Other important access modes are `"a"` for *appending* at the end of a file and `"r"` for *reading* a file. When the file is opened successfully, the function will return a *file pointer*, which points to an internal

[9] This may be different for computer systems, where the jobs are submitted via queuing systems. There, sometimes all paths have to be specified relative to the home directory, or relative to some special scratch directories.

structure where all information is stored, which is needed by the operating system to access the file. Technically, the file is treated as a so-called *stream*, which makes an output to different media possible in a unified way. To open a file, e.g. `"funcs.dat"`, for writing one can use

```
FILE *file_p;

file_p = fopen("funcs.dat", "w");
```

To actually write to the file, one can conveniently use the `fprintf()` function, which works exactly like the `printf()` function, except that the output is directed to a file. The format is similar to `printf()` except that the (additional) first argument must be a file pointer. Therefore, the general format is:

`fprintf(` ⟨*file pointer*⟩, ⟨*format strings*⟩, ⟨*argument 1*⟩, ...`)`

Assume that we want to write a four-column table containing some values of the functions $\sin(x), \cos(x), \exp(x)$ in the interval $[0, 2\pi]$. We write in the first line of the file a description of the following columns via

```
fprintf(file_p, "#    x    cos(x)   sin(x)   exp(x)\n");
```

Note that the first character '`#`' is recognized by most data analysis and plotting tools like `gnuplot` (see Secs. 7.4 and 7.6.2) as comment line. Hence, when you read in the file to postprocess the data, the first line will be ignored. Nevertheless, for your bookkeeping, you should always use such comment lines in your simulation output files. Having more information in the output files available will help you a lot in organizing and analyzing your results.

To write the actual data, one could use

```
for(x=0; x<=2*M_PI; x+=0.1)
  fprintf(file_p, "%f %f %f %f\n", x, sin(x), cos(x), exp(x));
```

There are some predefined file pointers like `stdout` (standard output) which writes to the screen. Hence, `fprintf(stdout, ...)` is equivalent to `printf(...)`. Also, there exists `stderr`, which is the standard output for error messages. This is also usually directed to the screen in interactive mode, but in case your programs runs in a special environment, like when using a queuing system, `stderr` is usually different from `stdout`.

Note that there are several other C functions for writing to files. Examples are `putc()`, which writes single characters, and `puts()`, which writes strings. Since they provide no functionality beyond `fprintf()`, they are not discussed here.

Finally, a file has to be *closed* when the access is terminated. For this purpose, the function `fclose()` has to be used, which expects as argument a file pointer of the file to be closed, e.g.:

```
fclose(file_p);
```

Once a file is closed, it can be accessed by other means, e.g. inspected via an editor. Note that while a simulation is running, data which is written via `fprintf()` will not be immediately forwarded to the file. Usually internal buffers are used, and the data is output to the file blockwise, each time the buffer is full. Nevertheless, emptying the buffer can be triggered within a program via the function `fflush()`, which also takes a file pointer as argument.

So far, we have discussed ways to output data from a program. On the other hand, your simulation programs usually need some input as well. The most direct way is to use interactive input, e.g. to type in some parameters values on request. This can be accomplished in C using the `getchar()` function, which just reads one character from the keyboard. Longer inputs can be read in using multiple calls to `getchar()`. Nevertheless, simulation programs are very often processed by batch queues on large-scale computing facilities. These batch jobs start at some time which is unknown in advance, hence you cannot sit in front of the screen, wait till your programs have started, and then supply the necessary input. Nevertheless, some programs, in particular for small systems and/or pedagogical purpose, may run interactively, quite often using a graphical user interface (GUI). Such interfaces are beyond the scope of this book; also the way the user interface is programmed under C depends quite often on the programming environment. Thus, the readers who are interested in simulations having a comfortable GUI should consider for example the JAVA programming language [JAVA]. Anyway, in the context of (large-scale) computer simulations, input to programs is either done via command-line arguments, which are treated below, or via reading in files, which we discuss next.

Input files are either parameter files, which tell the simulation program what the system to be simulated looks like, how many iterations are to be performed, or at which temperature your system is simulated. These files may also describe the complete (initial) state of a simulation, like the coordinates of all particles or, in general, the degrees of freedom. Input files may have been generated by previous simulation runs, which you would like to continue. Furthermore, files where the simulation results are contained in may be used as input files for further analysis. In case you do not use standard tools like *gnuplot* (see Secs. 7.4 and 7.6.2), you have to teach your self-written analysis program how to read in data. To summarize, reading data files is an important task. How this is performed in C is explained next.

Similar to writing to a file, one also has to *open* it for reading. This is again done with the `fopen()` function, but now the access mode should be `"r"`. This will again provide a file pointer (a stream in general), which can be used to actually read the file. The file pointer points to a memory area where all necessary internal information about the file is stored, and also to a *current position*, which indicates where the reading continues. Just after opening the file, the current position is the beginning of the file. There are several possibilities to actually read in and process the data. Here, we will describe one approach in detail, probably the most general one. For this purpose, we use the function `fgets()` ("file get string") , which exactly reads the next unread line of the file, i.e. till the next new line '`\n`' character. The format of the function is as follows:

`fgets(` ⟨*file name string*⟩, ⟨*maximum length*⟩, ⟨*access mode string*⟩`)`

Consequently, you have to pass a string `s`, where the line is stored to, the maximum number $n_{\max}$ of allowed bytes, and a file pointer to `fgets()`. Note that the string will be terminated by a `0`. Also, no more than $n_{\max} - 1$ bytes (without the closing 0) will be transferred to the string. This prevents you from writing beyond the reserved range of memory. `fgets()` will return the string which is passed (i.e. a pointer to the first character of the string), if something was read in. If the end of the file is already reached, i.e. no additional line could be read in, then a `NULL` pointer is returned.

Now, once the string is stored in memory, it can be further processed in many different ways, e.g., by using string-processing functions. There are many ways to treat the input lines, which basically depends on your chosen file format. For example, one can directly test the values of the string elements. Hence, if you want to filter out comment lines, which start by a '`#`' character, you can test whether `s[0] == #`. If yes, then one can continue with the next line.

A very convenient way to process strings is available if they follow some format, e.g., in case they were generated using `fprintf()` or similar functions. In this case one can use `sscanf()` which is basically the inversion of `sprintf()`: One has to supply as parameters an input string, which is to be analyzed, and a format string which may contain printable characters, escape sequences, and conversion specifiers. Depending on the conversion specifiers, *pointers* to variables also have to be supplied, one for each conversion specifier indicating a value to be read in.[10] The string is analyzed by comparing it to the format string. Whenever a conversion specifier in the format string is encountered, the corresponding value is extracted from the input string and stored in the given variable. This processing continues until the full input string is analyzed, or until *the first* mismatch between format string and input string is detected. For example, if the string `s` contains `"particles: 20 runs: 100"`, then the call to

```
int num_read;
int num_part = 0, num_runs = 0;

num_read= sscanf(s, "particles: %d runs: %d", &num_part, &num_runs)
```

will assign the value 20 to `num_part` and 100 to `num_runs`, and will return the value 2, which is assigned to `num_read`. If the format string contains a mismatch, then the processing stops. Thus

```
num_read= sscanf(s, "particles: %d Runs: %d", &num_part, &num_runs)
```

will only assign the first variable `num_part`, while `num_runs` remains at its initial value 0, and `num_read` will be 1. Therefore, if you are only interested in the number of particles anyway, you may also use

[10] There may also appear the conversion specifier modification '`*`' right behind the '`%`' character which results in skipping the corresponding value; hence, *no* pointer should be supplied for this item.

```
num_read = sscanf(s, "particles: %d", &num_part)
```

The conversion specifiers are the same as for `printf()`. There is one important point: When you print a variable of type `float` or `double`, you can use a conversion specifier `%f` in both cases. But when reading in a floating point number, which you want to assign to a variable of type `double`, you *must* use the conversion specifier `%lf`, otherwise the value will not be assigned correctly! Furthermore, there are also the functions `scanf()` and `fscanf()` which enable standard (keyboard) input or file input to be scanned directly. Nevertheless, this often generates problems. Hence, it is better to first read in a line via `fgets()` (or `gets()` for standard input) and then use `sscanf()`.

As we have seen, `fgets()` reports if the end of the file has been reached. This can also be tested directly via `feof()`, which takes a file pointer as argument. It will return true (value 1), if the end of the file has been reached. Note that it will not report *true* if just the last line has been read in, i.e., an attempt to read in the next, non-existing line is necessary.

GET SOURCE CODE
DIR: `c-programming` FILE(S): `file_in.c`

As a complete toy example, we next present a source code which reads in the four-column file which we have printed above (page 43). The program ignores all comment lines starting with a '`#`' character, and it just prints for each line the value in the first column and the sum of the values in the other three columns, a task which may occur for some data analysis problems[11]

```
1  int main(int argc, char**argv)
2  {
3    char line[1000];    /* string where one line of file is stored */
4    double val1, val2, val3, val4;             /* values from file */
5    int num_got;     /* how many where obtained from current line? */
6    FILE *file_p;                                  /* file pointer */
7
8    file_p = fopen("funcs.dat", "r");     /* open file for reading */
9
10   while(!feof(file_p))      /* read until end of file is reached */
11   {
12     if(fgets(line, 999, file_p) == NULL)         /* read in line */
13       continue;                             /* nothing was read in */
14     if(line[0] == '#')                           /* comment line? */
15       continue;                                        /* ignore */
16     num_got = sscanf(line, "%lf %lf %lf %lf",      /* get values */
17                      &val1, &val2, &val3, &val4);
18     if(num_got != 4)                           /* everything OK ? */
19     {
20       fprintf(stderr, "wrong line format in line: %s ", line);
21       continue;
22     }
```

[11] For this simple purpose, one should not write a program but use the *awk* tool. The example is just used for pedagogical purposes here.

```
23      printf("%f result= %f\n", val1,
24             val2+val3+val4); /* process */
25    }
26    fclose(file_p);                                  /* close file */
27
28    return(0);
29  }
```

First, the file has to be opened in line 8. Note that here the filename is hard-coded in the program. In general, one will need analysis programs which work for any files; hence, the file name has to be passed to the program. This is discussed below. Here, for the purpose of the example, the filename `"funcs.dat"` is sufficient. The file is processed in the main loop (lines 10–25), until the end of the file has been reached. First, the current line is read (line 12). Comment lines are ignored in lines 14 and 15. In lines 16–22, the content of the line is analyzed, and finally processed in lines 23–24.

Finally, note that a very useful string processing function for analyzing input files is `strstr()`, which is able to locate given string patterns inside other strings. This is useful for reading in poorly structured input files, where different values are identified by key words at arbitrary positions.

So far, we have just linearly read in a file. Sometimes it is necessary to read a file several times. This can happen when you first want to count how many input lines the file contains, e.g. to set up enough local storage space dynamically (see Sec. 1.4), and then actually read in the data in a second sweep.[12] For this purpose, the `rewind()` function can be used, which sets the internal position pointer back to the beginning of the file. One can even navigate completely freely inside a file. For this purpose, functions like `fseek()` and `ftell()` are available. For more details on these functions, please refer to the documentation.

As mentioned above, one can use files to pass simulation parameters to a program. This is in particular useful if many parameters are available, and if one wants to archive the values of these parameters for the different runs. This is helpful if one has to perform many different runs without getting lost in all these data. Note that organizing large-scale simulations in a useful way is an active area of research called *Computational Provenance* [Comp. Sci. Eng. (2008)]. A possible parameter file, e.g. for the Molecular Dynamics simulation[13] of a system of gas particles, could look like this

```
num_particles     = 512
temperature       = 37.3
number_steps      = 10000
step_size (fs)    = 1.2
box_size          = 10
```

[12] Alternatively, one could read in the data in one sweep, but dynamically extend the local storage space, if necessary.

[13] For a Molecular Dynamics simulation, you have formulas describing the forces between different particles (*force fields*). Using the forces you integrate the Newton's equation of motions to study the dynamic evolution of such a system. Molecular Dynamics simulations are widespread, such as to study the dynamics of proteins in cells.

```
appendix          = A67
save_config       = no
```

In real applications one could have many more simulation parameters, which for example describe different particle types and the coefficients for different force fields. Such a parameter file can be read in, as mentioned above, most conveniently using the `strstr()` function; we do not go into details here. Nevertheless, for many applications it is sufficient to pass simulation parameters as program arguments when invoking the program, e.g. like

```
arguments 512 37.3
```

where it is assumed that `arguments` is the name of the program, the first parameter (here 512) determines, say, the number of particles, and the second (37.3) determines the temperature of the system. Different arguments always have to be separated by spaces. For other parameters of `arguments`, the default values are taken, unless they are changed via options, as described below.

To be able to read the program arguments, one has to define the `main()` function as follows:

```
int main(int argc, char *argv[])
```

During the execution of `main()`, `argc` will contain the number of arguments including the program name, i.e. three in the above example. The array `argv` of strings contains in `argv[0]` the program name, in `argv[1]` the first argument (here 512), in `argv[2]` the second argument (here 37.7) and so on. Note that the arguments are stored in strings; hence, if they are to be interpreted as numbers, they have to be converted, e.g. using `atoi()`, which converts a string into an integer, or using `sscanf()` as described above.

For our example, we use an additional counter `argz`, to treat one argument after the other. This is useful in particular in case the program has additional options, see below. Here, the program arguments are treated via:[14]

GET SOURCE CODE
DIR: `c-programming` FILE(S): `arguments.c`

```
int N;                                  /* number of particles */
double temp;                                    /* temperature */
int argz = 1;                     /* counter to treat arguments */

N = atoi(argv[argz++]);
sscanf(argv[argz++], "%lf", &temp);
```

Some simulation parameters are usually kept at their default values and they do not have to be passed as arguments each time. In this case, it is convenient to use program options to change the default values. Program options are usually indicated by a '-' character at the beginning, followed by some name, and maybe some additional values. A call to the example program including some options could look like

[14] For this program, no line numbers are given here because we discuss different parts of the example program in non-linear order.

```
arguments -size 10.3 -appendix XX 100 3.7
```

Options and their accompanying values are stored in the `argv[]` strings like any other argument. Consequently, they can be processed using standard string manipulation. For instance, one can implement a loop, that is executed before the non-optional arguments are read, which is iterated as long as the "current" argument starts with a '-' character. Then, one can compare the "current" argument via `strcmp` to the different available options. If the option matches, one can take suitable actions, like setting a flag or assigning some parameter value. Finally, one should implement printing an error message in case the option passed as argument is not known to the program. For the example program, this could look like:

```
char appendix[1000];            /* to identify output file */
int do_save;                 /* save files at end of output */
int print_help;                    /* print help message ? */
double l;                         /* lateral size of system */
int num_steps;          /* how many MD steps are performed */

/** default values **/
l = 10; do_save = 0; appendix[0] = 0;
num_steps = 10000; print_help = 0;

/** treat command line arguments **/
while((argz<argc)&&(argv[argz][0] == '-'))
{
  if(strcmp(argv[argz], "-steps") == 0)
    num_steps = atoi(argv[++argz]);
  else if(strcmp(argv[argz], "-save") == 0)
    do_save = 1;
  else if(strcmp(argv[argz], "-size") == 0)
    sscanf(argv[++argz], "%lf", &l);
  else if(strcmp(argv[argz], "-appendix") == 0)
    strcpy(appendix, argv[++argz]);
  else
  {
    fprintf(stderr, "unkown option: %s\n", argv[argz]);
    print_help = 1;
    break;
  }
  argz++;
}
```

Note that different types of arguments have to be treated independently. For example, for `-save`, one just sets a flag variable, while for the other options additional values have to be read in, i.e. an integer value (`-steps`), a floating point value, (`-size`) or a string (`-appendix`).

It is also recommendable to print the calling sequence of your program and maybe some additional important simulation parameters at the beginning of each line to your output (or log) file, each line preceded by the '`#`' comment

symbol. This helps to reconstruct later on how the output file was generated, i.e. supports Computational Provenance (see page 47). For the current example it reads

```
char name_outfile[1000];                /* name of output file */
FILE *file_out;                  /* file pointer to output file */
   ...

sprintf(name_outfile, "md_N%d_T%3.2f%s.out",       /* file name */
        N, temp, appendix);
file_out = fopen(name_outfile, "w");

fprintf(file_out, "# calling sequence: ");        /* print args */
for(t=0; t<argc; t++)           /* print command line arguments */
  fprintf(file_out,"%s ", argv[t]);
fprintf(file_out, "\n");
fprintf(file_out, "# params: num_steps=%d, size=%f, save=%d\n",
        num_steps, l, do_save);
```

Before treating the non-optional program arguments, you should always test whether the number of supplied arguments is as expected. In case this is not true, your program can conveniently print out a simple error message which also explains the usage of your program. For the example above it looks as follows (this code appears just before the values `N` and `temp` are assigned):

```
if( print_help || (argc-argz) != 2)   /* not enough arguments ? */
{                                    /* print error/usage message */
  fprintf(stderr, "USAGE: %s {<options>} <num_part> <temp>\n",
          argv[0]);
  fprintf(stderr, "   options: -steps <n> : num. MD steps "
          "(d:%d)\n",  num_steps);
  fprintf(stderr, "     -size <l>: system size (d:%d)\n",
          l);
  fprintf(stderr, "     -save: save config at end\n");
  fprintf(stderr, "     -appendix <s>: for output file name\n");
  exit(1);
}
```

Finally, when your simulation is finished, you can close the output file. Quite often, simulation programs write out huge amounts of information. In this case, to save hard-disk space, you should use a compression tool like *gzip* to compress your output files. This can be done most conveniently right in your program. For this purpose you can use the `system()` library function, which takes as argument a string which may contain any command line, e.g. a shell command line under UNIX/Linux. For our example, we can compress the output file simply in the following way:

```
char command[1000];                  /* for system() calls */

     ...

fclose(file_out);                              /* close file */

sprintf(command, "gzip %s", name_outfile);     /* zip file */
system(command);
```

Using this compression, you also do not have to think much about how to save data very efficiently, since the zipping program takes care of this automatically. To unzip a file before reading it into your program is also quite simple, see exercise (3). The names of your compression tools, and the behavior of the `system()` function depend heavily on the programming environment and on the operating system. Hence, the application of this zipping might make your program less portable. Note also that the output file is only compressed after the simulation has terminated, hence when the output file is written completely. Therefore, it may occupy a lot of disk space before being compressed. If this poses a problem for you, you can also use on-the-fly zipping libraries like *zlib* [zlib], which works for all standard operating systems.

1.4 Pointers and dynamic memory management

More complex data types involve references between different objects. Consider, for example, a simulation of a social system, where you simulate a set of individuals, and you want to store for each individual the other individuals he/she knows. You could for example generate a big array with one entry for each individual. The entry for each individual carries a small array, which contains the indices of the other individuals he/she knows. On the other hand, if you assume that you have two big arrays, one for the males and one for the females, then you already have to distinguish whether an index refers to a female or a male. As you see, for more complex simulations, this might become even more cumbersome.

A more general and elegant approach is to use *pointers* to the objects you want to refer to. In this case it does not matter whether the objects are stored in the same or in different arrays, because pointers are basically memory addresses, as introduced in Sec. 1.1.1. Here, it is explained how dynamically changing data structures can be implemented via pointers and via memory management. We also show how pointers and arrays are related to each other. Using these basic ingredients, quite complex data structures as *lists*, *trees*, or *graphs* can be built. These advanced applications are discussed in Secs. 4.6 to 4.8.

First, the previously given information about pointers is summarized. A pointer `p` to a memory position, where a variable of type ⟨*type*⟩ is stored, is defined via

⟨*type*⟩ *p;

Initially, there is no value assigned to `p` (maybe zero for some compilers). To make `p` pointing to some relevant location, there are two possibilities: First, if `var` is a variable of type ⟨*type*⟩, one can write

```
p = &var;
```

Now, as you know already from page 11, `p` contains the address where `var` is stored. Assigning a value to `var` will not change `p`. Nevertheless, one can access or change the content of `var` via `*p`.

The second possibility is that one acquires some available memory, and let `p` point to it. This works via the `malloc()` ("memory allocation") function, which requires as argument the number of bytes to be allocated. It returns a pointer to "something" (i.e. of type `void *`), pointing to the reserved memory area:

void *malloc(⟨*size*⟩);

If the operating system cannot provide the requested memory, the special pointer value `NULL` is returned. Since the size of a type is determined using `sizeof()`, one can reserve a so far unused memory location where `p` can point to via:

`p` = (⟨*type*⟩ *) `malloc(sizeof(`⟨*type*⟩`))`;

Note the cast in front of the `malloc()` statement. This memory allocation does *not* take place during compile time or right when the program is started. It just happens, when the `malloc()` function is executed in the program. For this reason, the processes connected to `malloc()` are called *dynamic memory management*.

More interesting usages of pointers are possible. You could for instance have some `init()` function in your simulation (taking, say, an integer as argument) which also reserves some memory (where integers are stored) and returns the pointer to this memory. The function prototype might look like:

```
int *init(int);
```

On the other hand, functions can be referenced through pointers as well. To define a pointer `fct_p`, which points to a function taking an integer as argument and returning an integer, one writes

```
int (* fct_p)(int);
```

which looks very similar to the prototype above; it just differs by the brackets. Here, the usage of `typedef` is convenient. If pointers to functions of this form are needed frequently in the program, one can define a new type name and use it like e.g. here:

```
typedef int (* fctptr_t)(int);

fctptr_t fct_p;
```

Note that the name of the new type does not come at the end of the statement. An example where pointers to functions are needed is given in exercise (4), where a simple integration subroutine for an arbitrary function is given. Another example you find in Sec. 7.3.4.

With the above use of `malloc`, memory is allocated that can store exactly one value of the type ⟨*type*⟩. One can reserve a chunk of memory for several elements by just asking for more bytes. For example, one can reserve a chunk to store 10 persons of the `person_t` (see page 30) type via

```
person_t *pp;

pp = (person_t *) malloc(10*sizeof(person_t));
```

This is essentially the same as reserving an array of size 10. Here, one can access the `i`'th element via `*(pp+i)`, e.g. like in

```
for(i=0; i<10; i++)
  (*(pp+i)).age = 2*i;
```

Note that `pp+i` means that to the address `pp` one adds `i` times the number of bytes required to store `person_t`, i.e., increasing `p` by one here means to add `sizeof(person_t)` bytes to the address. Conveniently, it is also possible to write `pp[i]` to access the `i`'th element; hence, one can write `pp[i].age` to access the `age` member. For pointers to structures, there is an alternative syntax to access the members. Instead of `(*pp).age`, one can write `pp->age`. Similarly, to access the member `age` of the `i`'th element, the loop above can also read

```
for(i=0; i<10; i++)
    (pp+i)->age = 2*i;
```

Consequently, one-dimensional arrays and pointers, together with dynamic memory allocation, are very similar. One important difference is that for an array the size of the array cannot be changed, after it has been defined, while this is possible when using dynamic memory management. For this purpose, the `realloc()` function exists, which expects two arguments: 1. the pointer to the memory area to be extended and 2. the new size of the array:

void *realloc(⟨*pointer*⟩, ⟨*size*⟩);

The function returns a pointer to the new start of the memory chunk. This can be the same position as before the call. Nevertheless, it may sometimes be necessary to extend the memory chunk considerably. In this case it might happen that the operating system allocates a completely different part of the memory and copies the memory content from the old chunk to the beginning of the new chunk. In this case, the execution of the command might take some time, depending on the size of the memory area to be copied.

Note that one should pass only pointers to `realloc()` which have been allocated via `malloc()`, otherwise the behavior of the function is undefined.

Also, it might happen that the available memory is not sufficient. In this case the special `NULL` pointer is returned.

If one wants to allocate a matrix `mat` dynamically, one has to do this in two steps. First, one has to allocate one array which will contain pointers to the beginning of each row, respectively. Now, within a loop, the memory areas for the different rows can be allocated. If `n_rows` and `n_cols` are the numbers of rows and columns, respectively, the code to allocate the matrix `mat` could look like

```
double **mat;
int n_rows = 10, n_cols = 10;
int i, j;

mat = (double **) malloc(n_rows*sizeof(double *));
for(i=0; i<n_rows; i++)
  mat[i] = (double *) malloc(n_cols*sizeof(double));
```

The access can be performed in the same way as for standard matrices; hence, one can write, e.g. to initialize all entries to 0:

```
for(i=0; i<n_rows; i++)
  for(j=0; j<n_cols; j++)
    mat[i][j] = 0;
```

Note that due to the way the dynamically allocated matrix `mat` is stored, the actual access is different from a standard matrix `smat`, which is defined via `double smat[n_rows][n_cols]`: A standard matrix is stored in one large chunk of memory, where each row follows the next. On the other hand, the different rows for the dynamically allocated matrix `mat` are usually stored in different places. Therefore, a direct access could look like `*(*(mat+i)+j)`. This also means that a dynamically allocated matrix requires slightly more memory, because in addition to the actual matrix elements one needs an array with pointers storing the addresses of the different rows. Nevertheless, dynamically allocated matrices are more flexible, because the size can be changed during runtime. It is also possible that different rows contain a different number of elements, which is not possible for standard arrays either. This is useful, for example, when one wants to store an array of strings with strings of different lengths.

Finally, all memory which has been reserved inside the program should be released, one says *freed* again. For this purpose, the `free()` function can be used, which takes as argument a pointer to a previously allocated chunk of memory, e.g. like in:

```
free(p);
```

Freeing the memory which has been allocated for a matrix is performed in two steps: First, all rows are freed, then the array containing the pointers to the rows is freed. For the matrix `mat` this could look like:

```
for(i=0; i<n_rows; i++)
  free(mat[i]);
free(mat);
```

Freeing of memory is particularly important, if the memory is used only inside some function and will not be accessed again after the execution of the function has terminated. If it is not freed, and if the function is called several times, then more and more memory is allocated during the execution of the program, which may cause the operating system to run out of memory. Such a bug is called a *memory leak* and should be avoided under all circumstances. A useful tool to detect memory leaks and other bugs connected with dynamic memory management is introduced in Sec. 5.3. These bugs are often hard to spot by hand.

Using dynamic memory management, arbitrarily complex data structures can be generated. More advanced examples like *lists*, *trees*, and *graphs* are discussed in Secs. 4.6 to 4.8.

1.5 Important C compiler options

Throughout this book, many C compiler options are explained which are useful or even necessary for some tasks. Here, they are summarized:

`-o`

sets the name of the output file like e.g. in `cc -o first first.c`

`-c`

Just the compile process is performed, no final executable is linked. Each `.c` file will result in a corresponding `.o` file. This is useful in case the source code contains some functions which are used in several different programs. This process can be made automatic using *make* files, see Sec. 1.7.

`-l`

states the name of a library where some precompiled functions can be found, e.g. for mathematical functions (`m`) in
`cc -o mathtest mathtest.c -lm`

`-Wall`

Switches on (almost) all compiler warnings, e.g. indicates if one uses '`=`' inside a condition instead of '`==`'.

`-Wextra`

Switches on additional compiler warnings, e.g. indicates if one compares variables of different types.

`-Wshadow`

When using this option, the compiler will warn if a local definition shadows a global definition, e.g., in case there are two variables of the same name (see page 40).

`-g`

Switches on support for debuggers, see Sec. 5.1.

`-pg`

Switches on support for profiling, i.e. for measuring where the program spends how much running time, see Sec. 5.4.

`-O`

Switches on optimization of the code. This means it will run faster. E.g. the complier may "unroll" loops where the number of iterations is fixed. Also, it writes the code of a function inline instead of calling the function (can be forced via the `inline` directive at the beginning of a function definition). It can get rid of intermediate expressions/ find common subexpression. Furthermore, it can decide to put variables, which are used frequently, into registers (which can also be forced in the source by the `register` storage class specifier, which has to be printed in front of the type name of a variable).

Different levels of optimization are available like `-O0` (no optimization), `-O1`, `-O2` and `-O3`. The higher the level, the more optimized your code is. The highest level may even alter the meaning of your code under some circumstances; hence, to use `-O3` is dangerous. Here, `-O2` is recommended.

The effect on the running time of the different optimization options can be tested in exercise (5).

`-I`

Normally, the compiler looks for include files in standard search paths as indicated by the operating system, and in the current directory. Using this option, additional search paths can be stated where include files of your own libraries (see Sec. 6.4) can be found, e.g.
`cc -o prog prog.c -I$HOME/include`.

`-L`

Normally, the compiler (in fact the linker) looks for library (`.a`) files in standard search paths as indicated by the operating system, and in the current directory. Using this option, additional search paths can be stated, where your own ond other local libraries (see Sec. 6.4) can be found, for instance `cc -o prog prog.c -L$HOME/lib`.

`-D`

Defines a macro (see Sec. 1.6), as if it was defined via the `#define` directive in the source code, e.g. `cc -o program program.c -DUNIX=1`.

C compilers have many more options, some of them are machine-dependent. You should have at least a quick look once at the documentation of your compiler to see what options are available in principle.

1.6 Preprocessor directives and macros

The compile process can be controlled via *preprocessor directives*. The most important one is the `#include` directive, which makes the compiler read in the given file at the current position. In principle, arbitrary files can be included, but it is most useful to include header files which contain declarations of external functions, variables, and global data types, but no actual code. Note that there are two variants for including files:

- `#include <header.h>`

  This will include a standard header file `header.h` (which is to be replaced by a real file name) as provided by the operating system. Hence, the search for such files will usually take place in directories provided by the operating system. Basically, every program has to have a `#include <stdio.h>` at least. Other important standard header files to be included are `stdlib.h` and `math.h`.

  When using the `-I` compiling option, one can specify additional directories, where the compiler searches for header files.

- `#include "header.h"`

  This is used for your private and local header files. This means, the compiler will first look in the current directory for the header files. Only if they are not found there, it will look in the standard directories.

  More information on what you have to take into account when writing your own header files is given below.

The second important type of compiler directive is the definition of a macro. Macros are shortcuts for code sequences in programming languages. Their primary purpose is to allow computer programs to be written more quickly. But the main benefit comes from the fact that a more flexible software development becomes possible. By using macros appropriately, programs become better structured, more generally applicable, and less error-prone. Here, it is explained how macros are defined and used in C; a detailed introduction can be found in C textbooks such as Ref. [Kernighan and Ritchie (1988)]. Other high-level programming languages exhibit similar features.

In C a macro is constructed via the `#define` directive. Macros are processed in the preprocessing stage of the compiler. This directive has the form

`#define` ⟨*name*⟩ ⟨*definition*⟩

Each definition must be on one line, without other definitions or directives. If the definition exceeds one line, each line except the last one has to be ended with the backslash '\' symbol. The simplest form of a macro is a constant, e.g.

```
#define  PI  3.1415926536
```

You can use the same sort of names for macros as for variables. It is convention to use only upper-case letters for macros. A macro can be deleted via the `#undef` directive.

When scanning the code, the preprocessor just replaces literally every occurrence of a macro by its definition. If you have, for example, the expression `2.0*PI*omega` in your code, the preprocessor will convert it into `2.0*3.1415926536*omega`. You can use macros also in the definition of other macros. But macros are not replaced in strings, i.e. `printf("PI");` will print `PI` and not 3.1415926536 when the program is run.

It is possible to test for the (non)existence of macros using the `#ifdef` and `#ifndef` directives. This allows for conditional compiling or for platform-independent code, such as e.g. in

```
#ifdef UNIX
  ...
#endif
#ifdef MSDOS
  ...
#endif
```

Please note that it is possible to supply definitions of macros to the compiler via the `-D` option, e.g. `cc -o program program.c -DUNIX=1`. If a macro is used only for conditional #ifdef/#ifndef statements, an assignment like `=1` can be omitted, i.e. `-DUNIX` is sufficient.

When programs are divided into several modules, or when library functions are used, the definitions of data types and functions are provided in header files (`.h` files), as mentioned above. It is important that each header file should be read by the compiler only once for each source code file, otherwise declarations will appear twice and the compiler will complain. When projects become more complex, many header files have to be managed, and it may become difficult to avoid multiple scanning of some header files. This can be prevented automatically by this simple construction using macros:

```
/** example .h file: myfile.h  **/

#ifndef _MYFILE_H_
#define _MYFILE_H_

   .... (rest of .h file)
   (may contain other #include directives)

#endif /* _MYFILE_H_ */
```

After the body of the header file has been read the first time during a compilation process, the macro `_MYFILE_H_` is defined, thus the body will never be read again.

So far, macros are just constants. You will benefit from their full power when using macros with arguments. They are given in braces after the name of the macro, such as e.g. in

```
#define  MIN(x,y)  ( (x)<(y) ? (x):(y) )
```

You do not have to worry more than usual about the names you choose for the arguments, there cannot be a conflict with other variables of the same name, because they are replaced by the expression you provide when a macro is used, e.g. `MIN(4*a, b-32)` will be expanded to `(4*a)<(b-32) ? (4*a):(b-32)`.

The arguments are used in `( )` braces in the macro, because the comparison $<$ must have the lowest priority, regardless of which operators are included in the expressions that are supplied as actual arguments. Furthermore, you should take care of unexpected side effects. Macros do not behave like functions. For example, when calling `MIN(a++,b++)` the variable `a` or `b` may be increased twice when the program is executed. It is usually better to use inline functions (or sometimes templates in C++) in such cases. But there are many applications of macros, which cannot be replaced by inline functions, like in the following example, which closes this section.

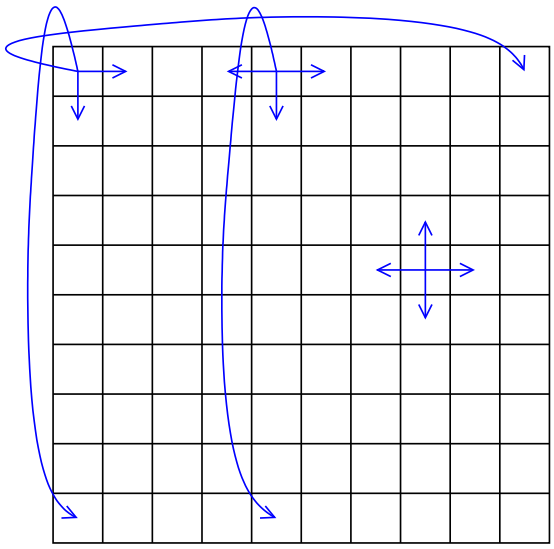

Figure 1.1: A square lattice of size 10×10 with periodic boundary conditions. The arrows indicate the neighbors of the spins.

The example illustrates how a program can be written in a clear way using macros, making the program less error-prone, and furthermore allowing for a broad applicability. A system of Ising spins is considered, i.e., a lattice where at each site i a particle σ_i is placed. Each particle can have only two states $\sigma_i = \pm 1$. It is assumed that all lattice sites are numbered from 1 to N. This

is different from C arrays, which start at index 0. The benefit of starting with index 1 for the sites is that, for many simulations of Ising systems, one needs the site 0 for additional technical reasons, see below. For the simplest version of the model only neighbors of spins are interacting. With a two-dimensional square lattice of size $N = L \times L$ a spin i, which is not at the boundary, interacts with spins $i+1$ ($+x$-direction), $i-1$ ($-x$-direction), $i+L$ ($+y$-direction), and $i-L$ ($-y$-direction). A spin at the boundary may interact with fewer neighbors when free boundary conditions are assumed. With *periodic boundary conditions* (pbc), all spins have exactly 4 neighbors. In this case, a spin at the boundary interacts also with the nearest mirror images, i.e. with the sites that are neighbors, if you consider the system repeated in each direction. For a 10×10 system, spin 5, which is in the first row, interacts with spins $5+1=6$, $5-1=4$, $5+10=15$ and through the pbc with spin 95, see Fig. 1.1. The spin in the upper left corner, spin 1, interacts with spins $2, 11, 10$ and 91. In a program pbc can be realized by performing all calculations *modulo* L (for the $\pm x$-directions) and modulo L^2 (for the $\pm y$-directions).

This way of realizing the neighbor relations in a program has several disadvantages:

- You have to write the code for realization of the pbc everywhere where the neighbor of spins are accessed. This makes the source code larger and less clearly structured.
- When switching to free boundary conditions, you have to include further code to check whether a spin is at the boundary.
- Your code works only for one lattice type. If you want to extend the program to lattices of higher dimensions you have to rewrite the code or provide extra tests/calculations.
- Even more complicated would be an extension to different lattice structures such as triangle or face-center cubic. This would make the program look even more confusing.

An alternative is to write the program directly in a way it can cope with almost arbitrary lattice types. This can be achieved by setting up the neighbor relation in one special initialization subroutine (not discussed here) and storing it in an array `next[]`. Then, the code outside the subroutine remains the same for all lattice types and dimensions. Since the code should work for all possible lattice dimensions, the array `next` is one-dimensional. It is assumed that each site has `num_n` neighbors. Then the neighbors of site `i` can be stored in `next[i*num_n]`, `next[i*num_n+1]`, ..., `next[i*num_n+num_n-1]`. Please note that the sites are numbered beginning with 1. This means, a system with `N` spins needs an array `next` of size `(N+1)*num_n`. When using free boundary conditions, missing neighbors can be set to 0. The access to the array can be made easier using a macro `NEXT`:

```
#define  NEXT(i,r)  next[(i)*num_n + r]
```

`NEXT(i,r)` contains the neighbor of spin `i` in direction `r`. For e.g. a quadratic system, `r=0` is the $+x$-direction, `r=1` the $-x$-direction, `r=2` the $+y$-direction and `r=3` the $-y$-direction. However, which convention you use depends on you, but you should make sure that they are consistent. For the case of a quadratic lattice, it is `num_n=4`. Please note that whenever the macro `NEXT` is used, there must be a variable `num_n` defined, which stores the number of neighbors. You could include `num_n` as a third parameter of the macro, but in this case a call of the macro looks slightly more confusing. Nevertheless, the way you define such a macro depends on your personal preferences.

The `NEXT` macro cannot be realized by an inline function, in case you want to set values directly like in `NEXT(i,0)=i+1`. Also, when using an inline function, you would have to include all parameters explicitly, i.e. `num_n` in the example. The last requirement could be circumvented by using global variables, but this is bad programming style as well.

When the system is an Ising spin glass, the sign and magnitude of the interaction may be different for each pair of spins. The interaction strengths can be stored in a way similar to the neighbor relation, e.g. in an array `j[]`. The access can be simplified via the macro `J`:

```
#define  J(i,r)  j[(i)*num_n + r]
```

A subroutine for calculating the energy $H = \sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j$ may look as follows (please note that the parameter `N` denotes the number of spins and the values of the spins are stored in the array `sigma[]`):

```
double spinglass_energy(int N, int num_n, int *next, int *j,
                        short int *sigma)
{
  double energy = 0.0;
  int i, r;                                          /* counters */

  for(i=1; i<=N; i++)              /* loop over all lattice sites */
    for(r=0; r<num_n; r++)             /* loop over all neighbors */
      energy += J(i,r)*sigma[i]*sigma[NEXT(i,r)];

  return(energy/2);    /* each pair has appeared twice in the sum */
}
```

The code for `spinglass_energy()` is very short and clear. It works for all kinds of lattices. Only the subroutine where the array `next[]` is set up has to be rewritten when implementing a different type of lattice. This is true for all kinds of code realizing e.g. a Monte Carlo scheme or the calculation of a physical quantity. For free boundary conditions, additionally `sigma[0]=0` must be assigned to be consistent with the convention that missing neighbors have the id 0. This is the reason why the spin site numbering starts with index 1 while C arrays start with index 0.

1.7 *Make* Files

If your software project grows larger, it will consist of several source-code files. Usually, there are many dependencies between the different files, e.g., a data type defined in one header file can be used in several modules. Consequently, when changing one of your source files, it may be necessary to recompile several parts of the program. In case you do not want to recompile your files every time by hand, you can transfer this task to the *make* tool which can be found on UNIX operating systems. A complete description of the abilities of *make* can be found in Ref. [Oram and Talbott (1991)]. You should look on the *man* page (type `man make`) or in the texinfo file [Texinfo] as well. Similar tools exists for other operating systems or software development environments. Please consult the manuals in case you are not working with a UNIX type of operating system.

The basic idea of *make* is that you keep a file which contains all dependencies between your source code files. Furthermore, it contains commands (e.g. the compiler command) which generate the resulting files called *targets*, i.e. the final program and/or object (`.o`) files. Each pair of dependencies and commands is called *rule*. The file containing all rules of a project is called *makefile*, usually it is named `Makefile` and should be placed in the directory where the source files are stored.

A rule can be coded by two lines of the form

⟨*target*⟩ : ⟨*sources*⟩
`<tab>` ⟨*command(s)*⟩

The first line contains the dependencies and the second one the commands. The command line must begin with a tabulator symbol `<tab>`. It is allowed to have several targets depending on the same sources. You can extend the lines with the backslash "\" at the end of each line. The command line is allowed to be left empty. An example of a dependency/command pair is

```
simulation.o: simulation.c simulation.h
<tab>   cc -c simulation.c
```

This means that the file `simulation.o` has to be compiled if either `simulation.c` or `simulation.h` have been changed. The *make* program is called by typing `make` on the command line of a UNIX shell. It uses the date and time of the last changes performed on each file, which is stored along with each file, to determine whether a rebuild of some targets is necessary. Each time at least one of the source files is newer than the corresponding target files, the commands given after the `<tab>` are executed. Specifically, the command is executed if the target file does not exist at all. In this special case, no source files have to be given after the colon in the first line of the rule.

It is also possible to generate meta rules which, e.g., tell how to treat all files which have a specific suffix. Standard rules, which tell how to treat files ending for example with `.c`, exist already, but can be changed for each file by stating a different rule. This subject is covered in the *man* page of *make*.

The make tool always tries to build only the first object of your *makefile*, unless enforced by the dependencies. Consequently, if you have to build several independent object files `object1, object2, object3`, the whole compiling must be toggled by the first rule, thus your *makefile* should read like this:

```
all: object1 object2 object3

object1:  <sources of object1>
<tab>    <command to generate object1>

object2: ...
<tab>    <command to generate object2>

object3: ...
<tab>    <command to generate object3>
```

Note that no command is given for the target `all`, because no commands to be executed are connected to it. It is just used to make sure that all objects on which it "depends" (artificially) are up to date. It is not necessary to separate different rules by blank lines. Here, they are just used for better readability. If you just want to rebuild e.g. `object3`, you can call `make object3`. This allows several independent targets to be combined into one *makefile*. When compiling programs via `make`, it is common to include the target "clean" in the *makefile* such that all objects files are removed when `make clean` is called. Thus, the next call of `make` (without further arguments) compiles the whole program again from scratch. The rule for 'clean' reads like

```
clean:
<tab>   rm -f *.o
```

Also, iterated dependencies are allowed, for example

```
object1: object2

object2: object3
<tab> ...

object3: ...
<tab> ...
```

The order of the rules is not important, except that *make* always starts with the first target. Please note that the *make* tool is not just intended to manage the software development process and toggle compile commands. Any project where some output files depend on some input files in an arbitrary way can be controlled. For example, you could control the setting of a book, where you have text-files, figures, a bibliography and an index as input files. The different chapters and finally the whole book are the target files.

Furthermore, it is possible to define variables, sometimes also called macros. They have the format

variable=definition

Also, variables belonging to your environment, like `$HOME`, can be referenced in the *makefile*. The value of a variable can be used, similar to shell variables, by placing a $ sign in front of the name of the variable, but you have to embrace the name by (...) or {...}. There are some special variables, e.g., $@ holds the name of the target in each corresponding command line; here no braces are necessary. The variable `CC` is predefined to hold the compiling command. You can change it by including for example

```
CC=gcc
```

in the *makefile*. In the command part of a rule the compiler is called via `$(CC)`. Thus, you can change your compiler for the whole project very quickly by altering just one line of the *makefile*.

Finally, it will be shown what a typical *makefile* for a small software project might look like. The resulting program is called `simulation`. There are two additional modules `init.c`, `run.c` and the corresponding header `.h` files. In `datatypes.h` types are defined which are used in all modules. Additionally, an external precompiled object file `analysis.o` in the directory `$HOME/lib` is to be linked whose corresponding header file is assumed to be stored in `$HOME/include`. For `init.o` and `run.o` no commands are given. In this case `make` applies the predefined standard command for files having `.o` as suffix, which reads like

```
<tab>   $(CC) $(CFLAGS) -c $@
```

where the variable `CFLAGS` may contain options passed to the compiler and is initially empty. The *makefile* looks like this (please note that lines beginning with '`#`' are comments:

```
#
# sample make file
#
OBJECTS=simulation.o init.o run.o
OBJECTSEXT=$(HOME)/lib/analysis.o
CC=gcc
CFLAGS=-g -Wall -I$(HOME)/include
LIBS=-lm

simulation: $(OBJECTS) $(OBJECTSEXT)
<tab>   $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(OBJECTSEXT) $(LIBS)

$(OBJECTS): datatypes.h

clean:
<tab>   rm -f *.o
```

The first three lines are comments, then five variables `OBJECTS`, `OBJECTSEXT`, `CC`, `CFLAGS`, and `LIBS` are assigned. The final part of the *makefile* are the rules.

Please note that sometimes bugs are introduced if the *makefile* is incomplete. For example, consider a header file which is included in several code files, but this dependency is not mentioned in the *makefile*. Then, if you change e.g. a data type in the header file, some of the code files might not be compiled again, especially those you did not change. Thus, the same object files can be treated with different formats in your program, yielding bugs which seem hard to explain. Hence, in case you encounter mysterious bugs, a `make clean` might help. However, most of the time, bugs which are hard to explain are due to errors in your memory management. How to track down those bugs is explained in Chap. 5.

The *make* tool exhibits many other features. For additional details, please consult the references given above.

1.8 Scripts

Scripts are even more general tools than *make* files. They are in fact small programs, but they are usually not compiled, i.e. they are quickly written but they run slowly. Scripts can be used to perform many administration tasks like backing up data, installing software, or running simulation programs for many different parameters. Here, only an example concerning the last task is presented. For a general introduction to scripts, please refer to a book on your operating system like UNIX/Linux.

Assume that you have a simulation program called `coversim21` which calculates vertex covers of graphs, which are graph-theoretical objects.[15]

Assume that you want to run the program for a fixed graph size `L`, for a fixed concentration `c` of the edges, average over `num` realizations, and write the results to a file, which contains a string `appendix` in its name to distinguish it from other output files. Furthermore, you want to iterate over different relative sizes `x`. Then you can use the following script `run.scr` (under UNIX/Linux, specifically the *bash* shell is used):

```
#!/bin/bash
L=$1
c=$2
num=$3
appendix=$4
shift
shift
shift
shift
for x
```

[15] For the definition of a graph see Sec. 4.8. A *vertex cover* of a graph $G = (V, E)$ is a subset $V' \subset V$ of vertices, such that each edge $\{i, j\} \in E$ is incident to at least one vertex of V', i.e. $i \in V'$ or $j \in V'$.

```
do
  ${HOME}/cover/coversim21 -mag  $L $c $x $num > \
                       mag_${c}_${x}${appendix}.out
done
```

The first line starting with "#" is a comment line, but it has a special meaning. It tells the operating system the language in which the script is written. In this case it is for the *bash* shell, the absolute pathname of the shell is given. Each UNIX shell is equipped with its own script language. Thus, you can use all commands which are allowed in the shell. There are also more elaborate script languages like *perl* or *python*, but they are not covered here.

Scripts can have command line arguments, which are referred via `$1`, `$2`, `$3` etc., the name of the script itself being stored in `$0`. Thus, in the lines 2 to 5, four variables are assigned. In general, you can use the arguments everywhere in the script directly, i.e., it is not necessary to store them in other variables. This is done here, because in the next four lines the arguments `$1` to `$4` are thrown away by four `shift` commands. Then, the argument which was on position five at the beginning is stored in the first argument, the initially sixth argument is now stored in the second one, and so on. Argument zero, containing the script name, is not affected by the shift.

Next, the script enters a loop, given by "`for x; do ... done`". This construction means that iteratively all remaining arguments are assigned to the variable "`x`" and each time the body of the loop is executed. In this case, the simulation is started via calling the program `coversim21` stored in the directory `${HOME}/cover/`with some parameters (where you do not have to care about their meaning here) and the output directed to a file. Please note that you can state the loop parameters explicitly like in "`for size in 10 20 40 80 160; do ... done`".

The above script can be called, for example, by

```
run.scr 100 0.5 1000 testA 0.20 0.22 0.24 0.26 0.28 0.30
```

which means that the graph size is 100, the fraction of edges is 0.5, the number of realizations per run is 1000, the string `testA` appears in the output file name, and the simulation is performed for the relative sizes 0.20, 0.22, 0.24, 0.26, 0.28, and 0.30.

Exercises

(solutions: see CD enclosed with book)

1. **Pointer juggling**

   Write a program which contains a `double` variable `value`, a pointer `p1` which should contain the address of `value` and a pointer `p2` which should contain the address of `p1`.

SOLUTION SOURCE CODE
DIR: `c-programming` FILE(S): `pointers.c`

   Perform the following steps

   (a) Define `p1` and `p2` using suitable data types.

   (b) Assign the desired values to `value`, `p1`, and `p2`.

   (c) Print `value`, `p1` and `p2` using printf and suitable conversion specifiers. Print furthermore the content of `value` via just using `p1` and via just using `p2`, respectively.

2. **Matrix permutation**

   Write a program which permutes a given matrix `test` such that neighboring rows are exhanged. Use the multiplication of a suitably chosen permutation matrix `perm` (from the left) with `test`.

SOLUTION SOURCE CODE
DIR: `c-programming` FILE(S): `matrix.c`

   The program should

   (a) contain the definition of three matrices, one is a `n`×`n` permutation matrix (`n` is an even number) `perm` and two (`test`, `result`) are `n`×`m` matrices,

   (b) initialize `perm` such that neighboring rows are exchanged,

   (c) initialize `test` in an arbitrary way such that different rows can be distinguished,

   (d) calculate the matrix product `result` = `perm`, × `test`

   (e) print the matrix `result` row by row.

3. **Online unzipping**

   Write a program which prints a file, which is passed as argument. If the file is compressed, visible via the appendinx `".gz"`, the it should be uncompressed first.

SOLUTION SOURCE CODE
DIR: `c-programming` FILE(S): `printzip.c`

   Requirements and hints:

   (a) The filename should be passed as first argument.

   (b) If no argument is given, a small error/usage message should be printed.

   (c) Use `strstr()` to locate the substring `".gz"` in the filename. If it is contained, the file should be decompressed via the help of the `system()` function.

   (d) The file should be printed linme by line.

   (e) If the file was compressed before it was printed, it should be compressed again, before the program stops.

4. **Integration of function**

SOLUTION SOURCE CODE
DIR: `c-programming` FILE(S): `integration.c`

Write a function which integrates a one-dimensional function $f(x)$ over the interval $[x_1, x_2]$ via the rectangle rule, i.e. calculates

$$I = \sum_{t=0}^{t_{\max}-1} \frac{f(x_1 + t\Delta) + f(x_1 + (t+1)\Delta)}{2} \Delta$$

where $t_{\max}$ is the number of integration steps and $\Delta = (x_2 - x_1)/t_{\max}$.
The function prototype reads as follows:

```
/******************** integrate() ********************/
/** Integrates a 1-dim function numerically using    **/
/** the rectangular rule                             **/
/** PARAMETERS: (*)= return-paramter                 **/
/**        from: startpoint of interval              **/
/**          to: endpoint of interval                **/
/**   num_steps: number of integration steps         **/
/**           f: p. to function to be integrated     **/
/** RETURNS:                                         **/
/**     value of integral                            **/
/*****************************************************/
double integrate(double from, double to, int num_steps,
                 double (* f)(double))
```

Test the function in your `main()` function by integrating the `sin()` function defined in `math.h` over different intervals, for example $[0, \pi/2]$, $[0, \pi]$.

5. **Optimizing code by compiler**

GET SOURCE CODE
DIR: `c-programming` FILE(S): `optimization.c`

Get the program `optimization.c` which calculates for a set of numbers for all possible subsets of numbers the sum. This takes exponentially long in the number of numbers, hence it is a good testbed to see the effect of compiler optimization

Compile the program via `cc -o optimization optimization.c -lm -Ox`, where `x` is `0`, `1`, `2`, and `3`. Measure the running time for executing the compile program via `time optimization` (note that the program does not print anything in the standard version).

Bibliography

[Philipps (1987)] Phillips, J. (1987) . *The Nag Library: A Beginner's Guide* (Oxford University Press, Oxford); see also `http://www.nag.com`.

[Kernighan and Ritchie (1988)] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*, (Prentice Hall, London).

[Loukides and Oram (1996)] Loukides, M. and Oram, A. (1996). *Programming with GNU Software*, (O'Reilly, London); see also `http://www.gnu.org/manual`.

[JAVA] *JAVA* programming language, see `http://www.java.com/`.

[Comp. Sci. Eng. (2008)] Computational Provenance, special issue of *Computing in Science & Engineering* **10** (3), pp. 3–52.

[zlib] *zlib* compression library, see `http://www.zlib.net/`.

[Oram and Talbott (1991)] Oram, A. and Talbott, S. (1991). *Managing Projects With Make*, (O'Reilly, London).

[Texinfo] Texinfo system, see `http://www.gnu.org`. For some tools there is a *texinfo file*. To read it, call the editor 'emacs' and type <crtl>+'h' and then 'i' to start the texinfo mode.