Consider the following statements:

```
int i = 3;
double a;
a = (double) i / 2;
```

Without the `(double)` *cast* operator, the variable `a` would take the value 0. Instead, by including the *cast* operator the integer variable `i` is converted into a `double` only in the context of the expression where `(double)` operates, while the integer variable `i` itself remains equal to `3`. The expression is promoted to a `double` and `a` takes the value 1.5.

3.4 Input/Output for beginners

Let us compile the Listing 3.1. The compilation does not give any errors and the program is executed without any problems. Nonetheless, we do not observe any effect of this execution. The reason is simple: the Listing 3.1 does not contain any *input/output* statements, (I/O), i.e., statements which receiving external data or return data from the program to an external device. The most obvious extension of the program Listing 3.1 consists in asking the user to supply a value of T_C for the variable `tc` and print the computed value of T_F (`tf`) as this is simply the aim of a measurement unit conversion program!

Before entering the details of the syntax of I/O statements, we briefly discuss what receiving input or producing an output means in practice. In today's computers an interactive program can receive input from any device that has been designed for this aim, for instance, from the keyboard or a file on a disk. In both cases, the program receives data through a communication channel which is managed by the operating system. Programming languages supply *native* functions which operate as an interface between I/O services of the system. The same holds for producing output. A program may write on devices, such as a terminal or a file, by using output functions of the language which depend on the operating system services.

The C language uses the `scanf` function to receive input from the keyboard and the `fscanf` function to receive input from a file. The output functions are `printf` to write to a terminal and `fprintf` to write to a file. These functions have a very well-constructed syntax which allows to use a variety of formats. In the remainder of this section, we illustrate just a part of the existing options and refer the reader to a C language manual for a complete overview [Kernighan and Ritchie (1988); Kelley and Pohl (1996)]. We anticipate that when using the `scanf` function a symbol of the C language is included whose meaning shall be clarified in Chapter 5.6.3. For now it suffices to follow the syntax of the command.

```
1 main() {
2    double tc, tf, offset, conv;
3    offset = 32.;
4    conv = 5./ 9.;
5    printf("Value in degrees Fahrenheit = ");
6    scanf("%lf", &tf);
7    tc = (tf - offset) * conv;
8    printf("Value in degrees Celsius = %f", tc);
9 }
```

Listing 3.4 Measurement unit conversion with I/O.

Observe the Listing 3.1 with the I/O statements. and examine the `printf` statement on line 5. Between the parentheses `( )` there is a string of characters which is delimited by double quotes on each side `"···"`. On line 5, the string is `Value in degrees Fahrenheit =`, and this string is printed on the output device (for instance the terminal). The aim is to be able to write informative messages that indicate what the program is computing, or what operations it is performing and similar information. On line 8 the syntax of the string included between the parentheses `( )` is slightly different: besides the message to be printed, it also includes a term `%f`. This term is called a *format specifier* as it serves to *specify the format* of the variable which follows the message. In this case the format specifier `%f` informs the compiler that the variable `tc` is of the `double` type, such that it can be correctly printed out onto the output. The `printf` function allows a variable number of arguments, separated from each other by a comma `,`. Its generic format is

`printf("`*`character string`*`"`*`[,exp1, exp2,...]`*`);`

The *`character string`* is always enclosed between double quotes `"···"`, and, in case it is followed by one or more expressions which need to be printed out, it contains a format specifier for each expression which follows: the format specifier has the form `%characters` and reflects the type of the expression it corresponds to. The expressions *`exp1, exp2,...`* are optional and their number may vary. We use the term *expressions* as *`exp1`* could be either a simple variable or a complex expression since we could have written `printf("Value in degrees Celsius = %f",(tf - offset) * conv);` i.e., we could have computed the value of `tc` directly in the arguments of `printf`. The format specifier `%f` refers to a quantity which is represented in floating-point format. In Table 3.4 the most frequently used format specifiers are listed for `printf`.

Format specifier	*Type*
`%f,%e,%g`	`float,double`
`%d,%i`	`int`
`%c`	`char` (single)
`%s`	`char` (string)

The `scanf` function, on line 6 of Listing 3.4, serves to read the value of one or more input variables (for instance from the keyboard). Its format is similar to that of the `printf` function. However, there are some important differences between the two. A first difference is that in the `scanf` function only format specifiers of the variables which are to be acquired can be included between the double quotes `"···"`. A second difference is the presence of the `&` symbol in the `scanf` function which must preceed every such variable.
The general syntax of the `scanf` statement is

```
scanf("character string",&var1[,&var2,...]);
```

The *`character string`* enclosed between double quotes `"···"` contains all format specifiers of the variables which are to be acquired in `var1, var2,...`and only those. Also in this case, the number of variables which follow the *`character string`* are not fixed and each variable must be preceded by the `&` character. On line 6 of Listing 3.4 the format specifier `%lf` for the `double` type variable `tf` is used. Note that if a format specifier `%f` corresponding to the `float` type were used, the program would behave differently. In Table 3.4 we list the most frequently used format specifiers for reading purposes.
The I/O functions for files are discussed in Chapter 5.6.3.

Finally, let us underline that the program Listing 3.4 actually does not compile correctly[2]; The reason is connected to the considerations made in Section 2.4, i.e. the necessity to *link* the code written by the pro-

[2]As explained in the next section there do exist compilers configured in such a way that they do not give any errors. However, in general, this affirmation is true.

Format specifier	*Type*
`%f`	`float`
`%lf`	`double`
`%Lf, %llf`	`long double`
`%d, %i`	`int`
`%u`	`unsigned int`
`%Lu`	`unsigned long long int`
`%c`	`char` (single)
`%s`	`char` (string)

grammer (Listing 3.4) to the already existing code, which in this case concerns the code for the `printf` and `scanf` functions. Indeed, the I/O functions `printf` and `scanf` are not written by the person who encodes a program to solve a given problem; rather they are already available from the C compiler, which is why they are generally called *system functions*.

Exercise 2 - I/O statements

? Write down `printf` and `scanf` statements to read the input from a `double` type variable `a` and a `long int` type variable `k`, based on a message that is printed by the program. Next, print out the values of `a` and `k` multiplied by the constant `3.14`, preceded by an explanatory message.

However, the fact that these functions are available from the compiler does not imply they are automatically linked to the remainder of the code. In order to achieve this link between the "personal" code and the "system" code, we at least need *to include* the file with all the references to external code in the "personal" code. This file is a first example of a *header* file, i.e., a file consisting of generally used declarations and definitions which must be included in the program to complete it. In Chapter 6.7.1 we discuss the technical details of the functions in the C language, which clarifies the necessity to include header files. There exist both library *header* files for system functionalities or user *header* files for functionalities which are specific to the application under consideration. The library *header* files are defined by a name enclosed between angular brackets as in `<stdio.h>`; for the user *header* files we need to specify their complete path between double quotes as in `"/home/include/myInclude.h"`[3]. Thus, the program of Listing 3.4 needs to be corrected as in Listing 3.5.

```
 1 #include <stdio.h>
 2 
 3 main() {
 4    double tc, tf, offset, conv;
 5 
 6    offset = 32.;
 7    conv = 5./ 9.;
 8    printf("Value in degrees Fahrenheit= ");
 9    scanf("%lf",&tf);
10    tc = (tf - offset) * conv;
11    printf("Value in degrees Celsius= %f",tc);
12 }
```

Listing 3.5 Measurement unit conversion with I/O.

The statement `#include <stdio.h>` on line 1 includes a system *header* file which makes all I/O functions of the C language available to the program. The format of this statement is discussed in Section 3.5. Note that in some cases the inclusion of a header file is not always sufficient to link the user's code to the system code. In these cases specific options must be included in the compilation command, such as the one given in Section 2.8.2 for the Linux operating system, `gcc -o program.exe program.c`, following the syntax of the operating system employed.

3.5 Preprocessor directives

Let us now discuss the statement `#include <stdio.h>` of Listing 3.5 in more detail. It has quite a different format compared to the lexical rules of the other C statements we have encountered so far since it starts with the character `#` and does not end with the character `;`. Indeed, as all statements which start with the character `#`, it is not a C statement, but rather a *preprocessor directive*.

The *preprocessor* is a software component which elaborates the C code before it is compiled. The preprocessor directives are usually included for two reasons:

- to include generally, though not necessarily, system declarations or definitions; they are synthetically expressed in one line of code and

[3]The compiler searches for the *header* files enclosed between angular brackets in either the predefined or the optionally specified *directory*.