Chapter 1

The C Programming Language

In this chapter we will learn how to

- write simple computer programs using the C programming language;
- perform basic mathematical calculations;
- manage data stored in the computer memory and disk;
- generate meaningful output on the screen or into a computer file.

The C programming language was developed in the early 1970's by Ken Thompson and Dennis Ritchie at the Bell Telephone Laboratories. It was designed and implemented in parallel with the operating system Unix and was aimed mostly as a system implementation language[1]. It, nevertheless, evolved into one of the most flexible and widely used computer programming languages today.

In 1989, the American National Standards Institute (ANSI) adopted a document that standardized the C language. The particular version of the language that it describes is widely referred to as *ANSI C* or *C89*. This document was adopted in 1990 by the International Organization for Standardization[1] (ISO) as *C90* and was later expanded to the current standard, which is often referred to as *C99*.

The C language consists of a small set of core commands and a large number of library functions that can be incorporated in a program, if necessary. One can find many excellent books that discuss the C language in detail, starting from the first book *The C Programming Language* by B. W. Kernighan and D. M. Ritchie[2]. This book describes ANSI C and remains today one of the easiest texts on the subject. In this chapter, we will cover the most basic of the core commands of the language, as well as those library functions that are useful in developing programs that involve scientific computations.

1.1 The first program

It is a tradition in the culture of C programming that the first program compiled and executed by a new student of the language is one that prints on the screen the happy message "Hello World!"[2]. This program, which is shown bellow, demonstrates the

Developing the C Programming Language

The developers of the C programming language, Ken Thompson (sitting) and Dennis Ritchie, in front of a PDP-11/20 computer at the Bell Labs, in 1972 (Scientific American, March 1999). The PDP-11/20 computer could manage 64 Kbytes of memory at any given time

basic structure of all programs written in C.

```
#include <stdio.h>              // incorporates libraries
                                // for input/output
int main(void)                  // begin main program
{
   printf(‘‘Hello World!\n’’);  // print on screen Hello World!

   return 0;                    // normal end of program
}
```

The first command that starts with the '#' sign is not an actual C command but rather an instruction to the compiler. Such instructions are called **preprocessor directives**. In this particular case, the directive

`#include <stdio.h>`

instructs the compiler to incorporate all those commands that are necessary for input and output of data. In C lingo, this directive instructs the compiler to incorporate the library of functions for the **st**an**d**ard **i**nput/**o**utput. Almost all C programs start with this directive, since they will require some input and will produce some output that will need to be communicated to the user!

The second command,

`int main(void)`

identifies the beginning of the main program. We will postpone the discussion of the syntax of this command until later, when we will study the definition of functions in C. For now, it suffices to say that the main program is the set of commands that appear between two braces (symbols '{' and '}') immediately following this identification.

The first command of the main program,

`printf(‘‘Hello World!\n’’);`

prints on the screen the message

`Hello World!`

The command name is `printf` and stands for *print formatted*. The command `printf` takes a number of **arguments**, which are enclosed in parentheses. In this particular example, the only argument is the **string of characters** "Hello

World \n", which is printed on the screen. All character strings in C are enclosed in quotes, which are not part of the string themselves. They denote its beginning and end and are not printed on the screen. The last part of the character string is the **control character `\n`**, which stands for *newline*. It also does not appear on the screen, but is there to instruct the program to begin the next output in the following line on the screen.

The final command on the program,

```
return 0;
```

identifies the point where the program reaches its normal end and control is returned to the operating system. The number `0` signifies the fact that the program is finishing normally, without any error messages.

Most lines in this example end with text in plain English that is preceded by the symbols `//`. These are **comments** to help the programmer understand the structure of the program and are not commands of the language. In fact, the compiler ignores anything from the symbols `//` to the end of the line. This construction, which is actually borrowed from C++, is one of the ways that allows a C programmer to insert explanatory comments in the program. In a different construction, comments are inserted between the symbols `\*` and `*\`, as in the following example:

```
\* This is a comment ...
   ... and can continue to a second line *\
```

This second construction allows for comments to occupy more than one lines.

Contrary to several other languages, there are very few and flexible rules in C that dictate the way a program needs to be typed. For example, empty lines or spaces are completely ignored by the compiler. The example program discussed in this paragraph can also be typed as

```
#include <stdio.h>
int
main(void) {printf(‘‘Hello World!\n’’)
;return;}
```

with the same result. This gives a programmer the flexibility to format the program in a way that elucidates its structure, flow, and sequence of commands. However, this flexibility also necessitates a method to instruct the compiler that the current command ended and that a new command is about to start. This is achieved by the semi-colon ‘`;`’, which appears at the end of the two commands in the main program in our example. Note that preprocessor directives (such as the `#include` command) do not end with semi-colons. Moreover, there is no semi-colon after the command `int main(void)` because the following set of lines that are enclosed in braces is considered part of the same command. Finally, there is no need for a semi-colon at the end of a block of commands enclosed in braces, because it is implicitly assumed to be there.

Compiling and Executing a C Program

Compiling and executing this program depends on the operating system and the C compiler used. Let us assume, as an example, that we have used an editor to type the program and save it in a file called `hello.c`. We use the suffix `.c` to denote that this file contains the **source** of a C program, i.e., the set of C commands that need to be compiled. In order to convert this program into an **executable** file we will invoke the GCC compiler of the GNU project[3] running under the LINUX operating system. In this case, we will type the command

```
gcc hello.c -o hello
```

The International Obfuscated C Code Contest

The C language offers an unprecedented flexibility both in the construction of a computer program and in the presentation of its source code. Since 1984, the International Obfuscate C Code Contest[5] has been rewarding the most "obscure/obfuscated C program, which shows the importance of programming style, in an ironic way". The program below, which was one of the winning entries in 1995, asks the user for an integer and calculates its factorial (you should try it!). Albeit legitimate, this is clearly not the way to write a computer program that is easy to understand or that allows for easily spotting potential mistakes.

```
#include <stdio.h>

#define llll 0xFFFF
#define lll for
#define lllll if
#define llll unsigned
#define llll struct
#define lllll short
#define lllll long
#define lllll putchar
#define lllll(l) l=malloc(sizeof(llll lllll));l->lllll=1-1;l->lllll=1-1;
#define lllll *lllll++=llll%10000;llll/=10000;
#define lllll lllll(!ll->lllll){lllll(ll->lllll);ll->lllll->lllll=ll;}\
lllll=(ll=ll->lllll)->lll;ll=1-1;
#define llll 1000



                                                     llll lllll {
                                                     llll lllll *
      lllll,*lllll        ;llll                      lllll lll [
      llll];};main      (){llll lllll                *llll,*lll,*
      ll, *llll, *    malloc ( ) ; llll              lllll llll ;
      lllll lll,ll  ,l;llll lllll *llll,*            lllll; lll(l
      =1-1 ;l< 14; lllll("\t\"8)>l\"9!.)>vl"         [l]^'L'),++l
      );scanf("%d",&l);lllll(lll) lllll(llll         ) (ll=lll)->
      lll[lll->lll[1-1]     =1]=llll;lll(lll         =1+1;lll<=l;
      ++lll){ll=llll;         llll = (llll=(         llll=lll))->
      lll; lllll =(            lll=ll)->lll;         ll=(llll=1-1
      );lll(;llll->             lllll||llll!=        *llll;){llll
      +=lll**llll++             ;lllll lllll         (++ll>llll){
      lllll llll=(              llll =llll->         lllll)->lll;
      }}lll(;llll;              ){lllll lllll        (++ll>=llll)
      { lllll} } *              lllll=llll;}
      lll(l=(ll=1-              1);(l<llll)&&
      (ll->lll[ l]              !=llll);++l);        lll (;ll;ll=
      ll->lllll,l=              llll){lll(--l        ;l>=1-1;--l,
      ++ll)printf(              (ll)?((ll%19)        ?"%04d":(ll=
      19,"\n%04d")              ):"%4d",ll->         lll[l] ) ; }
                                                     lllll(10); }
```

This invokes the application `gcc` to compile the C program that is stored in the file `hello.c` and stores the output of the operation in the file `hello`. Note that these two filename have to be distinct. Had we typed

```
gcc hello.c -o hello.c
```

the result of the compilation would have overwritten the C program and we would not be able to make any changes to it. If, on the other hand, we had omitted the last part of the command, i.e., if we had typed

```
gcc hello.c
```

the result of the compilation would have been stored in the default file `a.out`.

In order to execute the compiled program, we need to simply type

```
./hello
```

where the two symbols `./` preceding the name of the executable file simply instruct the operating system that the file exists in the current directory. The result of this command is

```
Hello World!
```

1.2 Managing Simple Data with C

One of the most important operations performed by a computer is the storage and manipulation of data. For archival purposes, data are stored in external devices, such as magnetic disks and laser disks, which retain the information even after the power of the computer has been turned off. However, in order for the computer to manipulate the data, they need to be stored in its random-access memory (RAM), which the microprocessor has a direct access of.

Data Types

Within the C language, different types of data are stored and manipulated in different ways, so that the minimum amount of memory is utilized with the maximum efficiency. Of all the **data types** recognized by the compiler, we will consider here only the four that are the most useful for mathematical computations. They are:

Type	`int`	integer numbers; ANSI C requires that they cover at least the range -32767 to 32767
	`float`	fractional numbers; ANSI C requires that they have at least 6 significant digits and they cover the range from 10^{-37} to 10^{37}, with both signs
	`double`	fractional numbers with at least 10 significant digits and a larger possible range of values
	`char`	single characters

Data of type `int` can be any integer number within the allowed range that does not include a decimal point. For example, the numbers

```
32, -18, 0, 13756
```

are all type `int`. However, the numbers

```
12.8, 32.0, 0.0
```

are not, even though, mathematically speaking, the last two numbers are integers! Data of type `float` and `double` can be any number within the allowed range that includes a decimal point. For example, the last three numbers can all be type `float` or `double`. We will discuss in more detail the first three data types in Chapter 2, where we will study the ways in which a computer stores numbers in its memory and performs numerical calculations.

Programming Tip

Choosing whether to use the type `float` or `double` in a program with scientific computations depends on the number of data and calculations involved as well as on the desired accuracy of the result. Data of type `float` are less accurate and cover a smaller range of values, but require half the amount of memory to be stored and calculations performed with them are typically faster than with data of type `double`.

Data of type `char` can be any single character. For example, the following

```
'A' 'B' 'y' 'z'
```

are all valid data of this type. Note that the single characters are enclosed in single quotes. This is necessary for the language to distinguish between data of type `char` and functions defined by the user that may have a name consisting of a single character. Characters are stored in the computer memory as integer numbers using a correspondence that was standardized in 1967 under the name of *ASCII*, which stands for the American Standard Code for Information Interchange. The ASCII table of characters and the integer number they correspond to is shown in Appendix A. Characters 0–33 are not printed on the screen and mostly control the way text is printed. For example, the control character `\n` that we saw in the previous section corresponds to the integer number 13 in the ASCII table.

Variables

In a high-level programming language, such as C, we do not store data directly to particular places in the computer memory. Instead, we define **variables** to which we assign the values we want to store and leave the nasty job of manipulating the computer memory to the compiler and the operating system.

A variable name can consist of any combination of the letters of the English language, the digits, and the underscore '`_`' but it cannot start with a digit. As an example, all the following are valid variable names

```
area, Mass_of_Electron, v_1
```

but

```
21_cross_section, +sign, a*
```

are not. Variable names cannot be any of the words reserved for C commands and functions. For example, `printf` is not a valid variable name, because it is a C command. Variable names are also case specific, which means that the variable `mass` is different than the variable `Mass`. Finally, only the first few characters of a variable name are recognized by the compiler and the remaining are ignored. In ANSI C, only the first eight characters specify uniquely a variable. For example, the variables `mass_of_electron` and `mass_of_proton` are indistinguishable in ANSI C, because they share the same beginning eight characters. In later versions of the C standard, a variable is uniquely specified by the first 31 characters of its name.

Programming Tip

Howlong the name of a variable is affects neither the amount of memory occupied by the compiled program nor the speed of execution. The variable name appears only in the source code and is there to make the program easy to understand and debug. It is, therefore, advantageous to use variable names that are self explanatory rather than ones that are generic or obscure. For example, an appropriate name for a variable to store the value of Planck's constant is `h_planck` and not simply `h`, since the latter can be easily misinterpreted to mean "height". Also, a good name for a variable to store the root of an algebraic equation is `root` and not simply `x`, since the latter can be misinterpreted to mean the coordinate of a point along the x-axis.

After choosing the name of a variable, we need to define the type of data it will carry, i.e., `int`, `float`, `double`, or `char`, and let the language assign a unique place in the computer memory where it will be stored. We achieve both functions by declaring the names and types of variables to the compiler in the beginning of the program. In the body of the program we can then assign data of the proper type to the variables we have declared. The example program shown in the following page, which calculates the area of a triangle, demonstrates the use of variable **declarations** and **assignments**.

Declarations and Assignments

The first command in the main program

```
float area;
```

declares to the compiler that a place in memory should be reserved for data of type `float` and the program will refer to this memory place with the variable name `area`.

Declarations of variables of the same type can be combined into a single command, as demonstrated by the second command in the main program

```
float height, base;
```

which declares two additional variables of type `float`. Finally, when declaring a variable, we have the option of assigning its initial value, as in the following command

```
int sides=3;
```

This command declares that variable `sides` is of type `int` and assigns to it (i.e., it stores in the corresponding memory space) the value 3.

```
#include <stdio.h>

\* Program to calculate the area of a triangle *\
int main(void)
{
   float area;                   \\ declare float variable area
   float height, base;           \\ declare more float variables
   int sides=3;                  \\ declare and initialize integer
                                 \\    variable

   height=2.5;                   \\ assign number 2.5 to height
   base=3.5;                     \\ assign number 3.5 to base

   area=0.5*height*base;         \\ calculate the product
                                 \\ 0.5*height*base
                                 \\ and assign it to variable area

                                 \\ print a message with the result
   printf(``The area of this shape with %d sides\n'', sides);
   printf(``is %f\n'', area);

  return 0;
}
```

Because we have not initialized the values of the other two variables, `height`, and `base`, it is important that we do so before we use them for the first time in the program. We achieve this with the following two assignment commands

```
height=2.5;
base=3.5;
```

The equal sign ('`=`') in these two commands simply means *assign to* and does not carry the implications of the equal sign in mathematics. It might be easier to think of the last two commands as the equivalent of

```
height←2.5;
base←3.5;
```

This apparently small distinction becomes really important in understanding assignments of the form

```
x=x+1.0;
```

where `x` is a variable of type `float`. In mathematics, this last command leads to a contradiction, if viewed as an equation, because cancellation of `x` leaves `0=1`, which is never satisfied. However, if we view this command as

```
x←x+1.0;
```

then we can easily understand its use. It takes the current value stored in the variable `x`, it increases it by one, and then stores the result again in the same variable `x`.

Assignments are used to perform numerical calculations. This is shown in the following command

```
area=0.5*height*base;
```

which calculates the area of the triangle as one-half times the product of its height to its base and stores it in the variable `area`. Note the self-explanatory names of the variables.

The next two commands in the main program print the result of the calculation on the screen. In the first occasion,

```
printf(``The area of this shape with %d sides\n'', sides);
```

we find the **specifier `%d`**, which instructs the compiler to print at this point in the output the variable `sides`, which is of type `int` and follows the double quotes. Similarly, in the second occasion,

```
    printf(``is %f\n'',area);
```

the specifier `%f` instructs the compiler to print at this point the variable `area`, which is of type `float` and follows the double quotes. The output of these two commands is

```
    The area of this shape with 3 sides
    is 8.75
```

Very large or very small numbers can be assigned to a variable in a compact form using the scientific notation. In C, as in most computer languages, the syntax of the scientific notation is a little different from the corresponding syntax in mathematics. For example, Avogadro's constant $N_\mathrm{A} \equiv 6.022 \times 10^{23}$ mol^{-1} can be assigned to a C variable with the line of code

```
    float N_Avogadro=6.022e23;              \\ in mol^{-1}
```

Note that the symbol `e`, which stands for *exponent*, takes the place of the symbols $\times 10$ in the usual scientific notation in mathematics. For very small numbers that require a negative power of ten, the syntax is very similar, with a negative sign preceding the exponent. For example, Planck's constant $h = 6.627\times10^{-34}$ m^2 kg s^{-1} can be assigned to a C variable with the command

```
    float h_Planck=6.627e-34;               \\ in m^2 kgr s^{-1}
```

Constants

In a program that involves scientific computations, we often wish to store a physical constant in a place in the computer memory and use it throughout the algorithm. For example, in a computer program that deals with the radioactive decay of ^{14}C to ^{14}N we might want to store the halftime of this reaction, which is approximately equal to 5730 years. We can achieve this by declaring a variable and assigning the value of the halftime to it, e.g.,

```
    double halftime_C14=5730.0;           \\ halftime in years
```

However, the value of this variable will not change throughout the program. C allows for a different declaration of such **constants**, which improves the speed of execution. For the example discussed above, the declaration would be

```
    const double halftime_C14=5730.0;     \\halftime in years
```

We can achieve the same result also using, in the beginning of the program, the preprocessor directive

```
    #define halftime_C14 5730.0           \\ halftime in years
```

Note that there is no equal sign between the name of the constant and its value. Moreover, there is no semicolon at the end of this line, because this is not a C command but rather an instruction to the compiler. During compilation, the compiler literally replaces all occurrences of `halftime_C14` in the source code with `5730.0`.

Programming Tip

A second advantageous use of constants in a program that involves scientific computations is in identifying various parameters of the numerical algorithm that may be different between different applications. These may include the limits of the domain of solution of an equation, the number of equations solved, or the accuracy of the solution. For example, we can write a general algorithm that solves a system of `Neq` linear equations and precede the C program by a compiler directive such as

```
    #define Neq 3
```

which specifies that in this particular occasion the system involves only 3 linear equations. If, in a different application, we need to solve a system of 5 linear equations, we will only need to change the directive to

```
    #define Neq 5
```

and leave the rest of the program unchanged. This technique improves the readability of the program and reduces the chances of introducing inadvertently mistakes to an algorithm that was borrowed from a different application.

1.3 Formatted Input and Output

Output on the Screen

As we saw in the previous section, the command `printf` controls the output of a C program on the computer screen. The general syntax of the command is

```
printf(‘‘control string’’, variable1, variable2, ...)
```

The control string is enclosed in double quotes and consists of printable characters, such as `Hello World!`, of specifiers, such as `%d` and `%f`, and of control characters, such as `\n`. For each specifier, there is a variable of the corresponding type following the control string.

Specifiers

The specifiers in the control string determine the position and format of printing variables of different types. The most useful ones for programs that involve scientific computations and the data types they correspond to are

Specifier	`%d`	data of type `int`
	`%f`	data of type `float` or `double` in decimal notation
	`%e`	data of type `float` or `double` in scientific notation
	`%g`	equivalent to `%f` or `%g` depending on the value of the number
	`%c`	data of type `char`

We have already discussed the use of the specifiers `%d` and `%f` in the previous section. As another example, the lines of code

```
int Neq=3;
float pi=3.1415927;
printf(‘‘The number of equations is %d\n’’,Neq);
printf(‘‘and the value of pi is %f\n’’,pi);
```

produce the output

```
The number of equations is 3
and the value of pi is 3.141593
```

More than one specifiers can be combined in a single `printf` statement, as long as they match the number and type of the variables that follow the control string. For example, the line

```
printf(‘‘Equations: %d; pi=%f\n’’,Neq,pi);
```

produces the output

```
Equations: 3; pi=3.141593
```

For very large or very small numbers, we often obtain more meaningful results if we use the `%e` specifier. For example, if we assign the mass of the electron to a variable of type `double` as in

```
double Mass_electron=9.11e-31;             \\ kgr
```

then printing it with the command

```
printf(‘‘Mass of electron=%f kgr\n’’,Mass_electron);
```

generates the output

```
Mass of electron=0.000000 kgr
```

Because the first six digits after the decimal point are zero for a number as small as the mass of the electron, the output is not meaningful. If, on the other hand, we use the command

```
printf(‘‘Mass of electron=%e\n’’,Mass_electron);
```

the output will be

```
Mass of electron=9.110000e-31 kgr
```

Programming Tip

It is important to emphasize that the specifier required for printing a variable is determined by the *type* of the variable, e.g., whether it is of type `int` or `float`, and not by the value assigned to the variable. For example, the following two lines of code

```
float days_in_year=365.0;
printf(``The number of days in a year is %d\n'',days_in_year);
```

are not correct. The variable `days_in_year` is of type `float`, even though its value is an integer number, whereas the specifier `%d` used in the `printf` statement is for variables of type `int`. The output of these lines of code is something similar to

```
The number of days in a year is 1081528320
```

which is clearly not what was intended.

There are a number of ways in which the output of variables of type `int` can be modified. The following few lines of code demonstrate the most useful modifications

```
int days_in_year=365;
printf(``#%d#\n',days_in_year);
printf(``#%5d#\n',days_in_year);
printf(``#%-5d#\n',days_in_year);
```

The output of these commands is

```
#365#
#  365#
#365  #
```

In the first `printf` statement, only the three digits of the number stored in the variable `days_in_year` are being printed on the screen, with no leading or trailing spaces. In the second `printf` statement, the integer number `5` between the symbols `%` and `d` specifies the minimum number of columns being allocated for the output of the variable. Because the value assigned to this variable has three digits, there are two additional spaces to the left of the number `365`. Finally, in the last `printf` statement, the minus sign after the symbol `%` generates an output of the value assigned to the variable `days_in_year` that is left justified.

The specifiers `%f` and `%e` can be modified in a very similar way to control the format of the output of floating numbers. As an example, the following lines of code

```
float pi=3.1415927;
printf (``#%f#\n'',pi);
printf (``#%e#\n'',pi);
printf (``#%9.3f#\n'',pi);
printf (``#%9.3e#\n'',pi);
```

generate the output

```
#3.141593#
#3.141593e+00#
#    3.142#
#3.142e+00#
```

In the last two `printf` statement, the number `9` after the symbol `%` specifies the minimum number of columns alocated for the output of the variable `pi`, whereas the number `3` after the decimal point specifies the number of digits that will be printed to the right of the decimal point. Note that in the case of the `%e` specifier, the total number of columns includes the column needed for printing the symbol `e` as well as the exponent.

Control Characters

The control string in a `printf` statement can also incorporate a number of control characters that provide additional flexibility in formating the printing of text and variables. One of these characters is `\n`, which instructs the program to continue printing in a new line. Some other useful control characters are listed below

Control Character	`\b`	backspace
	`\t`	horizontal tab
	`\\`	backslash (\)
	`\'`	single quote (')
	`\"`	double quote ('')

Note that the last three control characters allow us to print the symbols \, ', and '' without confusing them with other control characters of the C language.

Input from the keyboard

Many programs require input from the user to specify, for example, initial values or other parameters of the calculation. This task is achieved with the command `scanf`. Its general syntax is similar to that of the command `printf`, i.e.,

```
scanf(‘‘control string’’, &variable1, &variable2, ...)
```

with the control string consisting of printable characters, of specifiers, and of control characters. Note that, contrary to the command `printf`, the variable names in this case are preceded by the symbol `&`. We will discuss the reason for this in the section about pointers.

The main specifiers for the command `scanf` that are useful in computational physics programs are

Specifier	`%d`	Input will be interpreted as type `int`
	`%f`, `%e`	Input will be interpreted as type `float`
	`%lf`, `%le`	Input will be interpreted as type `double`
	`%c`	Input will be interpreted as type `char`

The use of the command `scanf` is illustrated with the program in the following page, which converts a temperature value from degrees Fahrenheit to degrees Celsius. The first two lines of the main program declare two variables of type `float` in which to store the value of the temperature expressed in the two temperature scales. Note the use of the comments following each declaration to explain the use of the variables.

The following command

```
printf("Degrees Fahrenheit? ");
```

prints the message

```
Degrees Fahrenheit?
```

and the command

```
scanf("%f",&degrees_F);          // Input from user Degrees F
```

waits for input from the user and stores it in the variable named `degrees_F`. The program then computes the equivalent value of the temperature in the Celsius scale using the assignment

```
degrees_C=(degrees_F-32.0)*5.0/9.0;
```

and outputs the result on the screen.

1.4 Evaluating Mathematical Expressions with C

The C programming language offers a wide variety of mathematical functions that we can use in performing numerical calculations. The basic algebraic manipulations, i.e., addition, subtraction, multiplication, and division, are performed with the symbols +, -, *, and \, respectively. For example, the command

```
a=b+c;
```

```
#include <stdio.h>

/* Program to convert degrees F to degrees C */
int main(void)
{
  float degrees_F;                // Degrees Fahrenheit
  float degrees_C;                // Degrees Celsius

  printf("Degrees Fahrenheit? ");
  scanf("%f",&degrees_F);         // Input from user Degrees F
                                  // Convert to Degrees C
  degrees_C=(degrees_F-32.0)*5.0/9.0;
                                  // Output result
  printf("%f degrees F are equal to ",degrees_F);
  printf("%f degrees C\n",degrees_C);

  return 0;
}
```

adds the values of the variables `b` and `c` and assigns the result to the variable `a`, whereas the command

```
a=b/c;
```

divides the value of the variable `b` by the value of the variable `c` and stores the result in the variable `a`.

There are a few shortcuts for common mathematical expressions that are incorporated in the syntax of the C language and not only result in a compact source code but often affect the speed of computations as well. For example, the commands

```
f++;
```

and

```
f--;
```

are equivalent to

```
f=f+1;
```

and

```
f=f-1;
```

respectively. The symbols `++` and `--` are called the increment and decrement operators, respectively. Moreover, the command

```
f+=a;
```

is equivalent to

```
f=f+a;
```

Similarly, the commands

```
f-=a;
```

and

```
f*=a;
```

are equivalent to the commands

```
f=f-a;
```

and

```
f=f*a;
```

respectively.

Operator Precedence

The C language follows a well defined set of rules regarding the order of evaluation of different operators that appear in a complicated mathematical expression. For example, among the operators that we discuss here, the increment and decrement operators (`++` and `--`) have the highest precedence, followed by the operators for multiplication and division (`*` and `/`), and then by the operators for addition and subtraction (`+` and `-`). When two or more operators of the same precedence appear in an expression, then the C language evaluates them in the order they appear, from left to right. As an example, in evaluating the expression

```
    c=2.0+3.0*5.0;
```

the multiplication `3.0*5.0` is performed first and the product is then added to `2.0` for a final result of `17.0`.

As in mathematics, we can change the ordering with which various operators in an expression are evaluated by grouping together complicated expressions using parentheses. For example, the command

```
    c=(2.0+3.0)*5.0;
```

assigns to the variable `c` the value `25.0`, because the parentheses force the addition to be performed before the multiplication. Parentheses can be nested at different levels to allow for more flexibility in the ordering of evaluation of a mathematical expression. For example, the command

```
    c=(0.8+0.2)/(2.0*(1.0+1.0)+1.0);
```

assigns to the variable `c` the value `0.2`.

Mathematical Functions

For more complicated mathematical operations, the C language makes use of an external library of mathematical functions. In order to employ them, we need to add to the beginning of the source code the preprocessor directive

```
    #include <math.h>
```

In many implementations of the C compiler, we also need to alter the command with which we compile a program whenever we are using this library of mathematical functions. For example, if we saved the source code of a program that uses mathematical functions in the file `math_calc.c`, then we would compile it with the command

```
    gcc math_calc.c -o math_calc -lm
```

The option `-lm` at the end of this command links the library with mathematical functions to the compiler.

The following table summarizes the most common of the functions in the mathematical library. In all cases, the arguments of the functions are variables of type `float` or `double` and the result should also be stored in variables of type `float` or `double`.

Function	`fabs(a)`	absolute value of a
	`sqrt(a)`	square root of a
	`pow(a,b)`	a to the b-th power (i.e., a^b)
	`sin(a)`	sine of a (in radians)
	`cos(a)`	cosine of a (a in radians)
	`tan(a)`	tangent of a (a in radians)
	`atan(a)`	arc (in radians) with tangent a
	`log(a)`	natural logarithm of a (i.e., $\ln a$)
	`log10(a)`	logarithm base 10 of a (i.e., $\log_{10} a$)

Note in particular that the arguments of the trigonometric functions `sin`, `cos`, and `tan` are in radians and not in degrees.

Programming Tip

It is possible to mix variables or constants of different data types in a single expression; the C compiler typically makes the appropriate conversion and calculates the correct result. There is, however, the potential of introducing in this way mistakes in the program that are very difficult to spot and correct. This is especially true when data of type `int` are mixed with data of type `float` or `double`. Consider for example the two lines of code

```
    float c,d=0.1;
    c=(1/2)*d;
```

The second command will assign zero to the variable `c` and not 0.05 as we might have thought. This happens because the ratio `(1/2)` is written as a ratio of integer numbers and C evaluates the result as an integer number before multiplying it to the value of the variable `d`. The integer part of the ratio `(1/2)` is zero and hence the result of this expression is zero. Had we written

```
    float c,d=0.1;
    c=(1.0/2.0)*d;
```

then the second command would have assigned the correct value 0.05 to the variable `c`.

1.5 Branching

A numerical calculation often follows different paths depending on the values of quantities that are evaluated as part of the calculation itself. For example, the roots of the quadratic equation

$$ax^2 + bx + c = 0 \tag{1.1}$$

can be real or complex depending on the value of the discriminant

$$\Delta \equiv b^2 - 4ac\ . \tag{1.2}$$

If $\Delta \geq 0$, then the equation has two real roots that are not necessarily distinct,

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}\ , \tag{1.3}$$

whereas if $\Delta < 0$, the equation has two complex roots

$$x_{1,2} = -\frac{b}{2a} \pm i\frac{\sqrt{\Delta}}{2a}\ . \tag{1.4}$$

A computer program that solves such a quadratic equation requires, therefore, two different ways of displaying the solution, depending on the value of the discriminant. In the C language, this type of branching is achieved with the `if` statement, as illustrated with the program that appears in the next page.

The program starts by asking the user to input the values of the three parameters of the quadratic, a, b, and c and then uses them to evaluate the discriminant `Delta`. At this point, the flow of the calculation depends on the value of the discriminant. For positive discriminants, i.e., when `Delta>=0`, the program evaluates the two real roots and prints the result. On the other hand, for negative discriminants, the program calculates the real parts of the two complex roots as well as the absolute values of their imaginary parts and prints the result.

Conditions

The general syntax of the `if` statement is

```
    if (condition)
        command 1
    else
        command 2
```

If the condition in the parenthesis that follows the `if` statement is satisfied, then `command 1` is executed, otherwise `command 2` is executed.

For the programs that we will be considering in this book, the condition is any logical expression that can be either true or false. This often involves the validity of

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

/* Program to solve the quadratic equation
       a*x*x+b*x+c=0 */
int main(void)
{
  float a,b,c;                    // Parameters of quadratic
  float Delta;                    // Discriminant
  float root1,root2;              // Two real roots
  float root_real;                // Real part of complex roots
  float root_imag;                // Abs value of imaginary part
                                  //     ... of complex roots

  printf("a, b, c? ");
  scanf("%f %f %f",&a, &b, &c);   // Input the parameters

  Delta=b*b-4.0*a*c;              // Evaluate Discriminant

  if (Delta>=0)                   // Two real roots
    {
      root1=0.5*(-b+sqrt(Delta))/a;
      root2=0.5*(-b-sqrt(Delta))/a;
      printf("The quadratic has the real roots: \n");
      printf("%f and %f\n",root1,root2);
    }
  else                            // Two complex roots
    {
      root_real=-0.5*b/a;
      root_imag=0.5*sqrt(-Delta)/a;
      printf("The quadratic has the complex roots: \n");
      printf("%f + i%f\n",root_real,root_imag);
      printf("%f - i%f\n",root_real,root_imag);
    }

  return 0;
}
```

mathematical inequalities, as in the example discussed above. Additional examples of conditions that involve mathematical expressions are given in the following table:

`x<a`	x is less than a
`x>a`	x is greater than a
`x<=a`	x is less than or equal to a
`x>=a`	x is greater than or equal to a
`x==a`	x is equal to a
`x!=a`	x is not equal to a

Note that the condition for the equality of two numbers involves the double-equal sign (`==`), which is different than the assignment operator (`=`) that we discussed before.

Logical Operators

We can create more complicated conditions by combining simple expressions with a number of logical operators. If we use `A` and `B` to denote simple logical conditions, then some useful logical operators in order of decreasing precedence are

`!A`	True if `A` is false (Logical NOT)
`A && B`	True if both `A` and `B` are true (Logical AND)
`A || B`	True if either `A` and `B` are true (Logical OR)

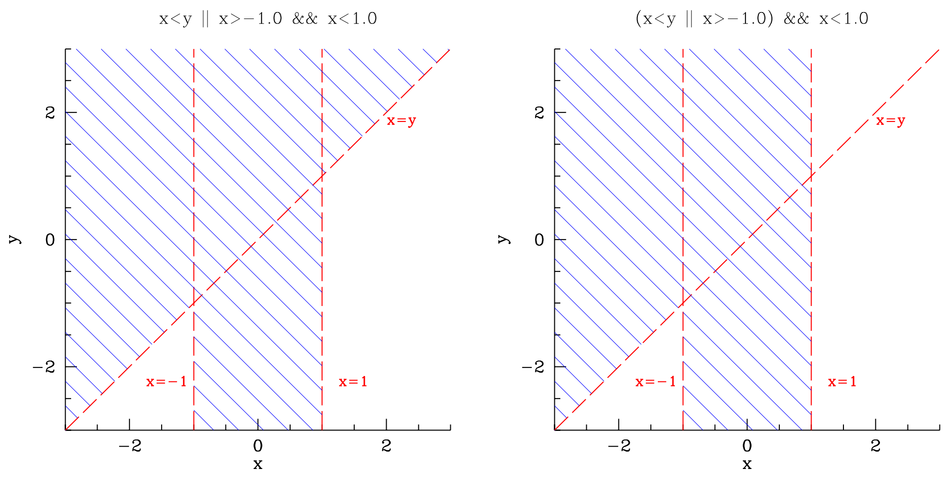

Figure 1.1: An example of logical operator precedence in the C language. The hatch-filled areas in the two graphs show the regions of the parameter space where the two shown conditions are true.

For example, the condition

```
x>-1.0 && x<1.0
```

is true only if `x` has a value in the range $-1 < x < 1$. On the other hand, the condition

```
x<y || x>-1.0 && x<1.0
```

is true if either `x` has a value in the range $-1 < x < 1$ or if it is smaller than `y`. As in the case of mathematical expressions, we can change the order with which different logical conditions are evaluated by using parentheses. For example, we can change the previous condition by adding parentheses as

```
(x>y || x>-1.0) && x<1.0
```

This last condition will always be false if $x \geq 1$, it will always be true if $-1 < x < 1$, but will also be true if $x \leq -1$, as long as $x < y$. Figure 1.1 illustrates the difference between the two conditions discussed in this example.

1.6 Loops

One of the most useful aspects of a computer is its ability to repeat many times a simple task. For example, calculating the factorial of a number N using the equation

$$N! = \prod_{i=1}^{N} i \tag{1.5}$$

requires the evaluation of N products, which can be a daunting task for a human if N acquires large values. The procedure of repeating a task multiple times is called *looping*.

Loops

The C language offers three different ways for performing loops. If the number of repetitions is known *a priori*, then we can use the command `for` that has the following general syntax:

```
for (initialization; condition; update)
    command
```

In the beginning of the loop, the *initialization* is evaluated. Then the `command` is repeated until the *condition* is met, with the *update* being evaluated at the end of each repetition.

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

/* Program to calculate the factorial of a number */
int main(void)
{
  int number;                      // number to calculate factorial of
  int factorial;                   // the factorial
  int index;                       // index for looping

  printf("Number? ");
  scanf("%d",&number);             // input the number

  if (number<0)                    // no factorial for negatives
    {
      printf("The number %d is less than zero \n",number);
    }
  else                             // normal case
    {
      factorial=1;                 // initialize product
                                   // and multiply integers<=number
      for (index=1;index<=number;index++)
        factorial*=index;
                                   // output the result
      printf("The factorial of number %d ",number);
      printf("is %d\n",factorial);
    }

  return 0;
}
```

The example program shown above illustrates the use of the command `for` in calculating the factorial of a number using equation (1.5). The command

```
for (index=1;index<=number;index++)
```

causes the program first to initialize the variable `index` to unity. It then evaluates the first term of the product with the commands

```
factorial*=index;
```

that sets the value of the variable `factorial` to 1 since `index=1`. The program then continues by updating the value of the variable `index` to 2, because of the last argument in the command `for`, i.e.,

```
index++
```

It then repeats the command

```
factorial*=index;
```

to set the value of the variable `factorial` to 2. Because of the second argument in the command `for`, i.e.,

```
index<=number
```

this process is repeated for as long as the value of `index` is smaller than the value of `number`. After each repetition, the value of `index` is increased by one. At the end of the loop, the variable `factorial` will have the factorial of the number stored in the variable `number`.

Programming Tip

Note that there is no semicolon after the closing parenthesis in the `for` statement, because the command that follows is considered as an integral part of the `for` statement itself. Had we put a semicolon there, the program would assume that there is no command to be repeated and would simply update the looping variable `index` without using it in any calculation. This is a common mistake in C programs and is very hard to spot.

If each repetition requires the execution of more than one commands, then we enclose the set of commands that need to be repeated in curly brackets. For example, we could change the loop in the previous program by adding an extra command that verbalizes to the user the progress of the algorithm, e.g.,

```
for (index=1;index<=number;index++)
  {
    factorial*=index;
    printf("Done with %d multiplications\n",index);
  }
```

In this case, both commands within the curly brackets will be repeated during the loop.

If the number of repetitions in a loop is not known *a priori* but a set of commands needs to be repeated while a condition is true, then we can use the `while` command. The general syntax of the command is

```
while (condition)
      command
```

This causes the program to repeat the `command` while the *condition* is true. As in the case of the `for` loop, the `command` can be either a single command or a set of commands enclosed in curly brackets.

For example, the following lines of code calculate the remainder of the integer division between the numbers `Num1` and `Num2` by subtracting the latter from the former until the result becomes less than zero

```
scanf("%d %d",&Num1,&Num2);
while (Num1>=0)
   Num1-=Num2;
printf("The remainder is %d\n",Num1+Num2);
```

Note again the absence of a semicolon following the closing parenthesis in the `while` command.

In both the `for` and the `while` commands, the condition that determines whether the looping commands continue to be repeated is checked in the beginning of each repetition. As a result, if the condition is false initially, then the looping commands are never executed. In several situations, however, it is useful for the condition to be executed at the end of each repetition. This will be necessary, for example, if the condition depends on the calculation being performed during each iteration of the loop. We can achieve this by using the `do - while` command. The general syntax of this command is

```
do
  command
while (condition);
```

In this case, the `command` is executed as long as the *condition* is true. However, because the *condition* is checked at the end of its repetition, the `command` will be executed at least once, even if the *condition* is false initially. Note the semicolon following the closing parenthesis that terminates the `while` command.

The program in the next page uses the `do - while` command to calculate the sum

$$\sum_{n=1}^{\infty} = 1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n} + ... \tag{1.6}$$

Because this is an infinite but converging sum, we choose to add all the terms until

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

/* Program to calculate the sum of the series
               1+1/2+1/4+...+1/2^n+...
   to a required accuracy               */

#define ACCURACY 1.e-6             // level of required accuracy

int main(void)
{
  float n=1;                       // index of sum
  float term;                      // each term in sum
  float sum=0.0;                   // current value of sum

  do                               // keep on adding terms
    {
      term=pow(2.0,-n);            // calculate current term
      sum+=term;                   // add to the sum
      n++;                         // and go to next term
    }                              // as long as haven't reached
                                   //      accuracy
  while (fabs(term/sum)>ACCURACY);
                                   // print result
  printf("The result is %f\n",sum);
  return;
}
```

the one that makes a fractional contribution to the sum that is smaller than the constant `ACCURACY`. In this case, it is to our advantage to have the condition checked at the end of each repetition, because at least one term needs to be calculated and the condition involves that calculated term.

Programming Tip

Note in this program the use of the constant `ACCURACY` that we introduced using the preprocessor directive

```
    #define ACCURACY 1.e-6          // level of required accuracy
```

which clarifies in plain English the condition in the `do - while` loop.

Nesting

The set of commands that are repeated in a loop may incorporate a second loop, which may incorporate a third loop, and so on. This is called loop **nesting** and can extend for several levels. The only requirement is that each inner loop must terminate before an outer loop does. This is illustrated with the example in the following page.

Programming Tip

Keeping track of the beginning and end of each loop, which is crucial for nesting loops properly, is very easy if the source code has proper indentation. In the program shown in the next page, the opening and closing braces for each loop (as well as for the main program) are placed on the same column of the text editor. This is called the *ANSI indentation style*, because it has been used in the documents that standardized ANSI C, but is not the only one. Different styles have different advantages and disadvantages. Although the choice is mostly a matter of esthetics, a well indented program will be very readable and mistakes will be spotted more easily. Most modern text editors perform the indentation automatically.

```
#include<stdio.h>

/* Prints the multiplication tables of the numbers between
   1 and 10 to illustrate the use of nested loops */
int main(void)
{
  int i,j;
  for (i=1;i<=10;i++)                           //     i-loop
    {                                           // --------------
                                                //  j-loop      |
      for (j=1;j<=10;j++)                       // --------|    |
        {                                       //         |    |
          printf("%2d x %2d = %3d\n",i,j,i*j);  //         |    |
        }                                       // ---------    |
                                                //              |
    }                                           // --------------
  return 0;
}
```

1.7 Arrays

Earlier in this chapter, we discussed variables of different type that can be used in storing data in the computer memory. Their only limitation is the fact that each variable can store only a single piece of data. There are many occasions in scientific computing, however, when the amount of information we wish to store is so vast that it is impractical for us to define a new variable for each piece of data. Consider for example, an X-ray detector that has been measuring the X-ray photons from a source for a total of 10 hours but has recorded the number of photons it detected in intervals of one second. If we wish to calculate the average rate with which photons were detected, then we will need to store all this information in the computer memory. Since there are 36,000 seconds in 10 hours, this will require defining 36,000 different variables!

The C language offers several ways of handling complicated and large data structures. Here we will discuss only the **arrays**, which are the simplest and most commonly used ones.

Declaration and use

An array is a data structure that allows us to store in the computer memory a large amount of data *of the same type*. As with all other variables, we have to declare an array before we can use it. Consider, for example, a program in which we wish to store the energy it takes to remove an electron from a neutral atom for the first six elements; this is called the ionization energy. We will need an array of type `float` with six elements in order to store the six different values of the ionization energy. We can declare this array with the command

```
    float ioniz_energy[6];            // ionization energy in eV
```

Note that, following the name of the array, we declared within the square brackets the number of its **elements**.

When declaring an array, we may initialize its elements by enclosing a list of values in curly brackets separated by commas. For the above example, we could write

```
    float ioniz_energy[6]={13.6, 24.6, 5.4, 9.3, 8.3, 11.3};
```

in order to initialize the array with the ionization energies (in eV) of the elements from hydrogen to carbon.

Following its declaration, we can use each element of the array in the same way

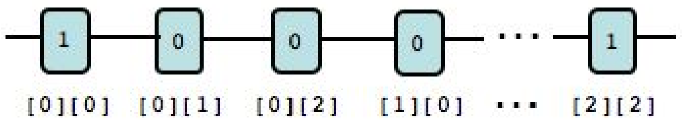

Figure 1.2: A visual representation of the way in which the two dimensional array of equation (1.7) is stored sequentially in the computer memory.

we would have used any variable of the same type. The only potentially confusing issue with arrays in the C language is the fact that we use the index [0] to refer to the first element of the array, the index [1] to refer to the second element, and so on. In the previous example, the command

```
    printf("%f %f\n",ioniz_energy[0],ioniz_energy[5]);
```

generates the output

```
    13.600000 11.300000
```

In general, if an index has N elements, then its first element will always correspond to index [0] and its last element to index [N-1].

Multidimensional Arrays

We can declare an array with more than one dimensions by adding to its name the number of elements in each dimension enclosed in square brackets. For example, we can store the 3×3 identity matrix

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.7}$$

in a two dimensional array that we declare using the command

```
    float identity[3][3];
```

As in the case of one dimensional arrays, the top left element of the identity matrix is identity[0][0] and the bottom right element is identity[2][2].

A multi-dimensional array is stored in the computer memory in a sequential way, as shown in figure 1.2. The first element identity[0][0] is stored first, followed by all the other elements of the first row. Then the elements of the second row are stored, and the same procedure continues until the last element of the array occupies the last allocated memory space.

We can make use of the sequential storage of a multi-dimensional array when we initialize the values of its elements. For example, the command

```
    float identity[3][3]={1.0, 0.0, 0.0,
                          0.0, 1.0, 0.0,
                          0.0, 0.0, 1.0};
```

declares the 3×3 array identity and initializes its elements to those of the identity matrix.

It is important to emphasize here that the standard libraries of the C language do not incorporate commands that perform operations between arrays. For example, the following lines of code

```
    float A[3][3],B[3][3],C[3][3];
    C=A+B;                          \\ does not work!
```

do not store in array C the sum of the arrays A and B. In order to perform a matrix addition, we need to add one by one all the elements of the arrays, as in the example shown in the following page.

```
#include<stdio.h>
#define Nelem 3

/* Calculates the sum of the 2-dimensional square matrices A and B
   and stores it in matrix C. The size of each matrix is controlled
   by the parameter Nelem */
int main(void)
{
  float A[Nelem][Nelem],B[Nelem][Nelem],C[Nelem][Nelem];
  int row, column;

  /* insert here code to initalize arrays A and B */

  for (row=0;row<=Nelem;row++)                // for all rows
    {
      for (column=0;column<=Nelem;column++)   // and for all columns
        {                                     // add each element
          C[row][column]=A[row][column]+B[row][column];
        }
    }
  return 0;
}
```

1.8 Functions

As we discussed in the beginning of this chapter, a key feature of the C language is the small number of its core commands and the extensive use of external **functions**. We are already familiar with the various mathematical functions that are part of the external **library** that we invoke with the preprocessor directive

```
#include <math.h>
```

For example, the command

```
b=sqrt(a);
```

calls the mathematical function `sqrt(a)` that calculates the square root of the variable `a`. Even the statements `printf` and `scanf` are functions that only perform an operation, such as print on the screen or request an input from the user, but do not return any specific value. (Speaking more precisely, they do return a value, but we do not need to use it for now.)

A function, in general, is any piece of computer code that may require some input (called the **argument** of the function), performs an operation, and may return a result. We may think of a function as a black box, for which we only need to provide a particular set of input parameters and expect a specific output. All modern computer languages give us the possibility of defining and using our own functions in a computer program. This is a fundamental ingredient of **structured progamming**.

The example in the following page illustrates how we can define and use a new function that calculates the factorial of an integer number. We choose to call this function `Nfactorial` (compare it to the program in Page 1–17).

Function Prototyping

The first command after the preprocessor directive informs the compiler that a new function will be introduced somewhere later in the program, it declares the number and type of its arguments, and finally specifies the type of data that the function returns. This is called **function prototyping**. In this case, the function will be called `Nfactorial`, it will take one integer value as an argument, and it will return an integer number as its result. Note the semicolon at the end of the

```
#include <stdio.h>

/* Function Prototypes */
int Nfactorial(int number);

/* Main program to calculate the factorial of a number */
int main(void)
{
  int number;                      // number to calculate factorial of
  int factorial;                   // the factorial

  printf("Number? ");
  scanf("%d",&number);             // input the number
  factorial=Nfactorial(number);    // calculate factorial
  if (factorial>0.0)               // if calculation is possible
    {                              // output the result
      printf("The factorial of number %d ",number);
      printf("is %d\n",factorial);
    }

  return 0;
}                                  // end of main program

int Nfactorial(int number)
/* Function that calculates the factorial of its
   argument. If number<0 it returns -1 */
{
  int factorial;                   // the factorial
  int index;                       // index for looping

  if (number<0)                    // no factorial for negatives
    {
      factorial=-1;                // result is the error code
    }
  else                             // normal case
    {
      factorial=1;                 // initialize product
                                   // and multiply integers<=number
      for (index=1;index<=number;index++)
        factorial*=index;
    }
  return factorial;               // return the result
}
```

command, which indicates that this is just the prototype and the function itself will be defined elsewhere.

The main program, which follows after the function prototype, is practically identical to that of Page 1–17, with one important difference. Instead of calculating the factorial of a number explicitly, the command

```
factorial=Nfactorial(number);   // calculate factorial
```

assumes that the function `Nfactorial` exists and performs that calculation. Although we have not explicitly indicated yet what this function does, the compiler will not complain because of the function prototype that we indicated before the main program.

From Spaghetti Code to Structured Programming

Early versions of computer languages that were developed in the 50's and 60's, such as IBM's FORTRAN (FORmula TRANslating system) and Dartmouth College's BASIC (Beginners All-purpose Symbolic Instruction Code), did not incorporate user-specified functions or procedures. In such languages, the flow of the program could be changed through GOTO commands, which caused the execution to jump unconditionally to a different part of the code. This unstructured programming practice is said to lead to *spaghetti* code, because the flow chart of such a code will look as tangled as a plate of spaghetti.
In the late 60's, marked by an influential paper by Edsger Dijkstra titled "GOTO Statement Considered Harmful", a new programming practice was developed under the name **structured programming**. In this approach, a computer program is divided into a large number of small, independent pieces and each of these pieces is implemented as an individual user-defined function. This practice minimizes (but does not eliminate) the use of GOTO statements and invariably leads to a clean and structured flow chart of the code. PASCAL was the first computer language that was specifically designed in 1970 for the teaching of structured programming but it was the C language that exploited this paradigm. Today, almost all computer languages, including the latest versions of FORTRAN and BASIC, comply with the philosophy of structured programming.

As we will see below, our new function returns the factorial of its argument, unless the argument is negative, in which case it returns the value `-1`. Because the factorial is defined only for positive integer numbers and is always larger than zero, the value `-1` is clearly an indication that something has gone wrong in the function we defined. This is called **error handling** and is an important component of all functions. If the result of the function is positive, then no error has occurred and the program prints out the value of the factorial.

The body of the function `Nfactorial` follows that of the main program. The first command

```
int Nfactorial(int number)
```

is the same as that used for the prototype, without the final semicolon. The command starts with the data type of the function's result, which in this case is of type `int`. It is then followed by the name of the function and, finally, by the data types and names of the arguments enclosed within parentheses. In this case, the function takes only one argument, a variable of type `int` that will be called `number`.

The Main Program

Comparing the initial statement of the function with the initial statement of the main program, i.e.,

```
int main(void)
```

it becomes obvious that the main program is just another function for the C compiler, with the distinction that it is the one that is executed first. Note that the main program does not take in this example any arguments, although it may in principle. For this reason, its argument list within the parentheses is replaced by the type `void`. Moreover, the main program is expected to return a result of type `int`, which may be used by the operating system for error handling, for example.

Returning to the main body of the function `Nfactorial` in the previous example, we notice that it starts with the declaration of two local variables and continues with the evaluation of the factorial of the integer stored in the variable `number`. At the end of the calculation, the result is stored in the variable `factorial`. The last statement in the body of the function, i.e.,

```
        return factorial;              // return value of factorial
```
instructs the compiler to return this value as the result of the function. For obvious reasons, the data type of the variable that follows the statement `return` should be the same as the data type of the function stated in the prototype and in the declaration. In this example, both the function `Nfactorial` and the variable `factorial` are of type `int`. In the case of the main program, which in the C language is a function of type `int`, we emphasize the fact that the program finished without errors by returning the integer value zero with the command
```
      return 0;
```
Note that a function can return only one result. In §1.10, where we will introduce the concept of pointers, we will discuss an alternate way in which we can program a function to affect the value of more than one variables.

A function can have more than one arguments of different types. The most general syntax of a function declaration is

`type` *function_name*`(type` *variable1*`, type` *variable2*`, ...)`

where each variable in the argument list is preceded by its type. For example, the following lines of code implement a function that calculates the logarithm of a number in an arbitrary base, using the expression

$$\log_b a = \frac{\ln a}{\ln b}\ . \tag{1.8}$$

```
   float logbase(float number, float base)
   /* Function to calculate the logarithm of the variable number in
      an arbitrary base. If either argument is negative it returns
      the value -9999, which would not have occurred even for the
      smallest allowed positive number of type float. */
   {
      if (number>0 && base>0)    // logarithms defined only for
        {                        // positive numbers
          return log(number)/log(base);
        }
      else                       // error handling
        {
          return -9999.0;
        }
   }
```

There are three points to note in this function. First, both the variable `number` and the variable `base` in the argument list are preceded by their types, even though both are of type `float`. This is contrary to the usual variable declarations, where variables of the same type can be collected together in one decleration statement. Second, the variables in the argument list are not declared again in the main program. Finally, a function may have more than one `return` statements, although this practice is susceptible to introducing logical mistakes in the code, since it corresponds to a function that has multiple exit points.

1.9 Local and Global Variables

Every function, including the main program, starts with the decleration of the variables that will be used